# ANALYTICAL MODELING OF SHARED BLOCK CONTENTION IN CACHE COHERENCE PROTOCOLS[1]

Jin-Chin Wang

CENG Technical Report: 91-12

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Electrical Engineering)

October, 1990



[1]This work has been supported by the National Science Foundation under Grant DCCR-8709997

## Abstract

The objective of this dissertation is to understand the program behavior in terms of shared block accesses in cache-based multiprocessor systems. In this direction, we study the following issues: develop an analytical program model which captures the program behavior of shared block accesses, verify the accuracy of the model by comparing model predictions and simulation results for several parallel algorithms, and evaluate the performance of general purpose multiprocessor systems through model implementation.

Cache-based multiprocessor systems with a large number of processors are difficult to evaluate. Prototyping is costly and often impossible and simulations are not flexible enough and are very time consuming and resource intensive. The major difficulty lies in the estimation of the coherence overhead. An approximate and economical technique is to develop an analytical program model for block sharing and then to apply the model to the various architectures for comparison.

In many cache-based multiprocessor systems, private caches are associated with each processor and coherence among caches is maintained in hardware by a cache coherence protocol. Multithreading, or the concurrent execution of the multiple processes forming a task is also often supported in these systems. The efficiency of multiprocessor systems for a parallel algorithm depends to a large extent on the amount of sharing in the algorithm as well as the effectiveness of the cache protocol for shared block accesses.

In this dissertation, we develop a simple program model for block sharing called the *access burst model.* This model is based upon the observation that shared writable data are accessed in critical or semi-critical sections. The program model is applied to the analysis of multiprocessor systems in steady state. We find an analytical closed-form solution for all components of the cache coherence overhead for five protocols and compare the model predictions with the execution-driven simulation results for eight parallel algorithms. The correlation between cache size and cache coherence overheads is also studied. The coherence overheads in the infinite cache system are always upper bounds of those caused in the finite cache system when the same trace of events drive both systems.

# 1 INTRODUCTION

## 1.1 Motivation for this Dissertation

The importance of parallel processing has been recognized as the complexity of attempted problems increases. The goal of parallel processing is to reduce execution time by running concurrent and cooperating processes on multiple processors. Parallel processes communicate and synchronize either through shared writable data or message passing mechanisms. The way in which these communication and synchronization mechanisms are supported has great impact on system performance.

One of the major problems of parallel systems is maximizing performance. As the number of processors increases, the demand for memory bandwidth also increases. Communication among processors is the major bottleneck, especially with today's fast microprocessors. Cache memory serves as a bridge between a processor and main memory, which reduces memory access latency and increases memory bandwidth. Many researchers [2, 4, 10, 30, 40, 41, 47, 57, 80, 81, 89] have searched for techniques which will reduce communication overheads in cache-based multiprocessor systems by establishing a framework for performance evaluation. Solutions to these problems are essential for the success of cache-based multiprocessor systems.

The three approaches generally used to evaluate the performance of multiprocessor systems are measurements, trace-driven simulations, and analytical program model, which vary in cost, flexibility, and accuracy. Measurements have the highest credibility; however, they are limited to existing machines. Some hardware parameters such as cache block size are not adjustable. Trace-driven simulations of real programs are usually time-consuming and the systems being simulated are, in general, limited to small processor configurations. Moreover, the major drawback for measurements and trace-driven simulations is that they fail to explain the observed performance level. The results are only valid for specific benchmarks run on the specific architecture.

A good analytical program model to evaluate multiprocessor systems can be particularly useful because the analytical program model can be used to quickly analyze new ideas in architecture. Such a model has theoretical foundations and can be used to explain experimental data. An objective of this dissertation is to find a representative analytical program model which captures program behavior in terms of shared block accesses, reaches the same level of accuracy as that of trace-driven simulations, and has much higher flexibility. Analytical program models are the most economical way to evaluate multiprocessor systems.

Performance indices, such as speedup and throughput, provide rough estimates of system performance. These indices, however, do not reveal much useful information for performance improvement. We intend studying the effects of indices characterizing both the system hardware (such as number of processors in the system and cache block size) and software behavior (such as write rate and

sharing characteristics), all of which affect the performance of a multiprocessor system. These performance indices lay out the foundation for fine tuning of both software and hardware.

Stochastic analytical models are simple. They capture the general behavior of many applications. Through the derivation of the models, insight is gained into the important parameters affecting performance. In turn, this insight helps programmers optimize hardware utilization and exploit parallelism better by improving the software design. Of course, to be applicable, an analytical program model must be validated against measurements or trace-driven simulations. The major contribution of this dissertation is to define an analytical program model, to validate it against execution-driven simulations [1], and to apply it to various protocols.

## 1.2 Cache Coherence

In multiprocessor systems, caches introduce the *cache coherence* problem. Censier and Feautrier gave a definition of a coherent memory system as follows [17]:

> *A memory scheme is coherent if the value returned on a LOAD instruction is always the value given by the latest STORE instruction with the same address.*

In many cache-based multiprocessor systems (see Figure 1), private caches are associated with each processor and coherence among caches has to be maintained by some mechanisms. The cache coherence problem can be avoided either by using the shared cache or by making shared objects non-cacheable [71]. The shared cache which was adopted in the Univac 1100/82 [15] is not feasible for a system with large number of processors, since the bottleneck of the system is now transferred from shared-bus to the shared cache. In addition, a longer average cache access time is incurred because the shared cache cannot be physically close to all processors. When shared objects are not cached, certain items such as semaphores or job queues can only be accessed from the shared memory. The drawback of non-cacheable data is that the access time of these data is substantially increased. Also, in [24], Darema-Rogers *et al.* pointed out that shared data accesses account for a large portion of memory accesses. For efficiency, shared data must therefore be cached.

When shared data are cached, coherence among caches can be maintained by either software or hardware mechanisms. Cheong and Veidenbaum [18], and Min and Baer [62] proposed three software-based compiler-directed strategies to selectively flush caches. Compiler-directed management of caches implies that a

[1] In trace-driven simulations, traces are collected from a real, existing multiprocessor system which serves as input to a new system being studied. Whereas in execution-driven simulations, this input are generated by simulating the execution behavior of the new system.

processor has to issue explicit instructions to invalidate cache blocks and special hardware is needed to support this scheme. Another alternative uses hardware coherence protocols [10], to maintain the coherence. Cache coherence protocols may suffer from scalability problems; however, they are the most efficient and user-transparent mechanisms to maintain the consistency of shared data in bus-based systems. My research applies to architectures in which coherence is enforced by a hardware protocol.

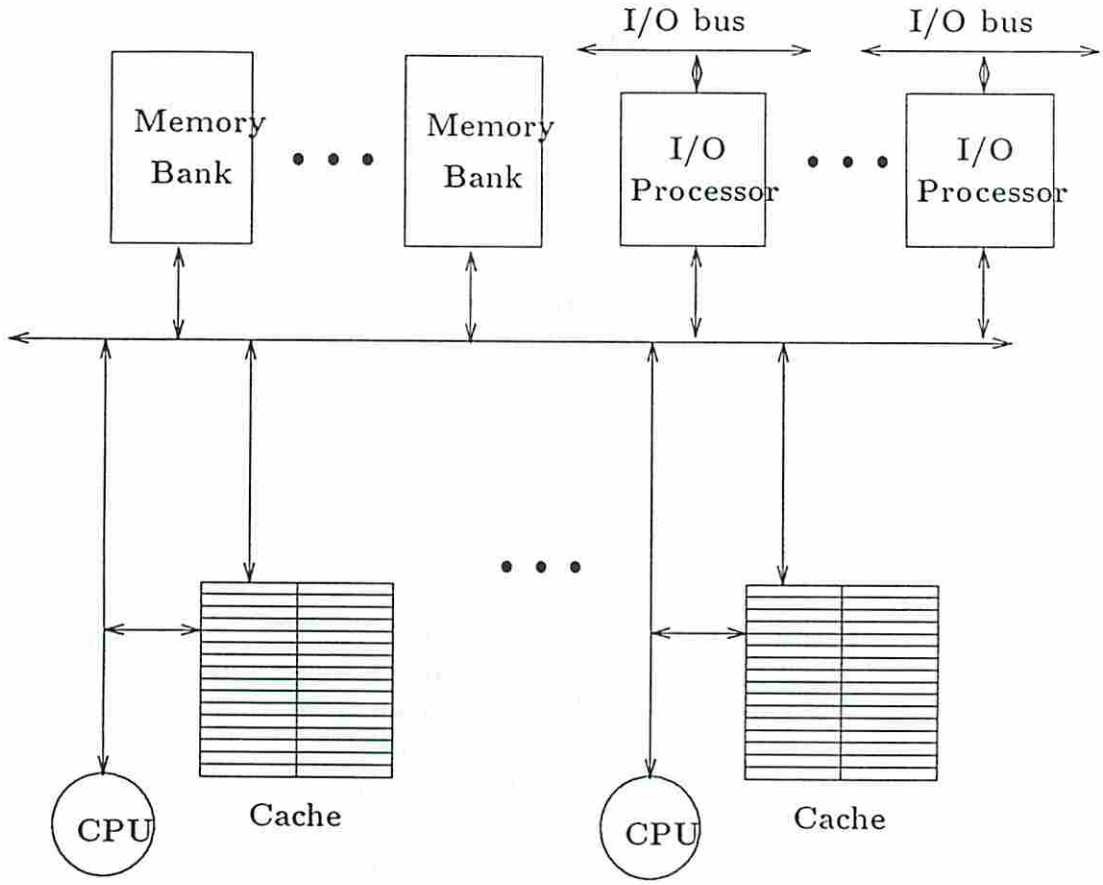



Figure 1: A shared-memory multiprocessor system

### 1.2.1 Cache Coherence Protocols

Cache coherence protocols have been shown to be an important design consideration in shared-bus multiprocessor systems because of the differences in performance among protocols [10]. Two classes of such protocols, write-through protocols and write-back protocols, are identified based upon the actions required to modify data [49, 63]. In write-through protocols, Stores are immediately transmitted to shared memory. However, write-back protocols do not update the shared memory until dirty blocks are replaced in the caches [49]. Archibald and Baer have shown that write-back protocols generate less bus traffic than write-through protocols [10]. Thus, in this dissertation, we concentrate only on write-back protocols.

Write-back protocols can be classified into two categories, namely write- invalidate protocols and write-broadcast protocols. The first type of protocols maintains consistency by invalidating all copies in other caches on a Write. The Basic [30], the Write-Once [44], the Synapse [42], the Berkeley [54] and the Illinois [64] cache coherence protocols fall into this category. On the other hand, in write-broadcast protocols all copies are updated on a Write by distributing the new value to all caches. The Firefly [77, 78] and the Dragon [60] cache coherence protocols belong in this category.

The rest of this subsection describes these coherence protocols in greater detail. A Write to a shared block can be performed locally (private write) or globally (shared write). Unlike private writes which only update local cache blocks, a shared write not only updates the shared block resident in the local cache, but also changes the status of the copies of the same block present in remote caches.

Write-Invalidate Protocols

The Basic Cache Coherence Protocol: The Basic cache coherence protocol [30, 49] is adopted by cache-based multiprocessor systems with no direct cache-to-cache transfers. All transfers between caches must transit through the shared memory. This is typical of high-end main-frame multiprocessors such as the IBM 3081 series [20]. A detailed description for the Basic cache coherence protocol is in Chapter 2.

The Write-Once Cache Coherence Protocol: In [44], Goodman proposed the Write-Once cache coherence protocol. The Write-Once protocol employs write-through on every first shared write to a cache block and write-back on all following writes until a read access by another processor. After the first shared write to a shared writable block, the state of the block is assigned as either Reserved or Dirty. All subsequent accesses to the block are done locally until a miss request is generated by another cache. This scheme needs cache-to-cache transfers on miss requests for dirty blocks. During the transfer, shared memory is updated. This protocol has been implemented in a hierarchical multiprocessor system by Encore Computer Corporation [87].

The Synapse Cache Coherence Protocol: The Synapse N+1 architecture was a tightly-coupled MIMD multiprocessor system for on-line transaction processing [42]. This protocol is developed based on the concept of *block ownership*. An owner bit is associated with each block and both shared memory and caches can be the owner of the block. Ownership by a cache block carries the right to update the block locally without having to inform other caches and shared memory. Upon a miss request, if the owner of the requesting block is a cache, the ownership of the block is transferred to either shared memory or the requesting cache according to the type of the request, a read miss request or a write miss request. If the request is a read miss, the ownership of the block is transferred to shared memory

in the cache and a busy acknowledge signal is sent to the requesting cache to retrieve the block from shared memory. Otherwise, a cache-to-cache transfer is initiated and the ownership is transferred to the requesting cache along with the copy of the block. The Synapse protocol avoids the extra memory update on first shared write incurred by Write-Once protocol; however, it pays a large penalty on read miss requests when the owner of the block is not shared memory.

The Berkeley Cache Coherence Protocol: This protocol was adopted in the SPUR multiprocessor project at Berkeley [54] and is based on the concept of block ownership. Once the ownership for a cache block is obtained, the cache can update the block locally without initiating additional bus operation. The protocol overcomes the disadvantage in the Synapse protocol on read miss requests when the owner of the block is a cache. A miss request is satisfied by the owner of the missing block. All cache-to-cache transfers are done in one single bus transfer and shared memory is not updated until an owned block is replaced. Also, if no cache owns the missing block, shared memory is the owner of the block. This eliminates the need to explicitly represent ownership with an additional bit in memory as in the Synapse protocol.
Illinois Cache Coherence Protocol: Papamarcos and Patel proposed this low-overhead cache coherence protocol in 1984 [64]. In this scheme, a new state called Exclusive-Unmodified is introduced, which eliminates redundant invalidations when a cache identifies that the block is the only cached copy in the system. In the implementation of this protocol, shared memory is updated during a miss request for a dirty block. The simultaneous shared memory update reduces the number of cache-to-memory transfers on dirty block replacement incurred in the Basic, the Write-Once the Synapse and the Berkeley cache coherence protocols. However, to implement this protocol, a special design for cache snooping controller is needed to prevent memory latency from dominating the time of memory-to-cache transfer.

Write-Broadcast Protocols

The Firefly Cache Coherence Protocol: The Firefly protocol has been implemented in the Digital Equipment Corporation Firefly multiprocessor system, which is a shared memory multiprocessor workstation [78]. Under the Firefly protocol, caches broadcast the updated word on every shared write; for private data, a write-back mechanism is adopted. The bus-watching snoopers assert a special control line called shared line to indicate sharing. When a snooper detects an operation for a cache block which resides in its cache, it raises the shared line. In this scheme, a cache can detect whether the retrieved block on a cache miss is private or not, according to the signal on the shared line. If the block is a private block, all following writes to the block can take place without informing other caches. Thus, this protocol has potential performance benefits for accessing both private and shared blocks.

The Dragon Cache Coherence Protocol: The Dragon protocol is adopted in a multiprocessor system designed by Xerox PARC [60]. The protocol is very similar to the Firefly protocol; the only difference between the two protocols is that on every broadcast, the Firefly protocol updates the shared memory and caches at the same time, while the Dragon protocol only updates caches. There is no memory update in the Dragon protocol until a dirty cache block is replaced.

### 1.2.2 Performance Analysis of Cache Coherence Protocols

Different approaches have been adopted to tackle the problem of analyzing cache coherence overhead. In this section, we survey the most important methods that have been used in recent years.

Dubois and Briggs [30] studied the shared data coherence activity in a shared memory multiprocessor system with centralized directory coherence scheme. Each processor in the shared memory multiprocessor system has a private cache with finite size and its organization is fully associative. The reference stream for their simulations resulted from the merging of two reference streams, a stream of private/shared read-only accesses and a stream of shared read-write accesses, where the first stream is generated based on the Least Recently Used stack model (LRU) and the second stream is derived based on the Independent Reference Model (IRM) [75]. The state transitions of shared data in the system are modeled by semi-Markov diagrams. Analytical solutions are derived for those diagrams to evaluate the hit ratio degradation due to the sharing of data. Unfortunately, the IRM model cannot make a precise prediction of shared data behavior in multiprocessor systems because it does not reflect the locality of shared data accesses.

Archibald and Baer [9, 10] used simulations to compare the performance of cache coherence protocols in shared-bus multiprocessor systems. The replacement policy for shared blocks in caches was random. The workload model they used is called Multiple Least Recently Used stack model (MLRU) which was derived from the reference stream developed by Dubois and Briggs [30]; in this model, if the request is to a shared block, the block number of the reference is determined using an LRU stack (one stack for each processor). The metric for protocol comparisons was the system power, which is the number of processors times their average CPU utilization. Their analysis focused on the degree of sharing. They found that the system power for all write-back protocols was comparable when the number of processors is small. When the number of processors increases, the broadcast-based protocols are better than the invalidate-based protocols especially in the case of high contention for shared block. The protocols such as the Illinois, the Firefly and the Dragon, which can detect private and unmodified blocks always perform the best when the degree of sharing is low. With a high degree of sharing, the Berkeley protocol performed better than the Illinois protocol. The rest of the protocols can be ranked in terms of system power as follows: The Write-Once, the Synapse and the Write-Through.

Vernon and Holliday [80] used generalized time Petri-Net to model multiprocessor memory systems and studied the bus utilization and system power for data caches. Again, they used the same workload model which was derived by Dubois and Briggs [30]. A generalized time Petri-Net can be converted into a discrete Markov chain and thus can be solved. The protocol was similar to the Write-Once protocol; however, more features were added to the protocol, such as a shared line to detect a private unmodified block, and an owned shared state to allow cache-to-cache transfer of dirty blocks without updating the shared memory. Their results showed that the shared line increases the system performance dramatically, especially when the level of sharing is intensive. The only problem that they had was that the number of states in the discrete Markov chain explodes when the sum of processors and memory modules were larger than ten in the system.

Leutenegger and Vernon [57, 81] used mean value analysis to evaluate the bus interference for the Wisconsin Multicube cache coherence protocol. The workload derived by Dubois and Briggs [30] was used in this study. The model includes FCFS scheduling at the bus queues with deterministic bus access time, asynchronous broadcast invalidations, and asynchronous memory write-back operations. The study indicated that this simple approximate approach yields comparable results to the generalized time Petri-Net, across a wide range of parameter values. The computation time is a hundred times less than that in the generalized time Petri-Net when large systems are studied.

Eggers and Katz [40] used trace-driven simulations to evaluate the coherence overhead in four parallel CAD applications in infinite cache multiprocessor systems. A model called the Write Run Model is developed for analyzing the program behavior in terms of shared data accesses. A write run is a sequence of references to a shared address by a processor, which begins with the first Write to the address by the processor and ends with an access by another processor. The average difference between model predictions and simulation results is around 200% in the Berkeley cache coherence protocol. Nevertheless, the model can make an accurate prediction, with less than 1% difference between model predictions and simulation results in the case of the Firefly protocol. The reason why their model can make an excellent prediction in a given protocol but fails in another protocol is because the coherence overhead in write-invalidate protocols is a function of the cache block size contrary to write-broadcast protocols. Their model can capture only the effects in systems with cache block size equal to one.

In [41], Eggers and Katz used trace-driven simulations to compare the performance of four types of snooping cache coherence protocols, which are write-invalidate protocols, read-broadcast extension protocols, write-broadcast protocols and competitive snooping protocols. The systems are bus-based multiprocessor systems and the sizes of caches are assumed to be infinite. The read-broadcast extension protocols [45, 68] are enhancement to write-invalidate protocols. Under read-broadcast, snoops update an invalidated shared block with data from shared-bus when they detect a read bus operation for the shared

block. The competitive snooping protocols [52, 53] are hybrid protocols which use both write-invalidate and write-broadcast features to maintain coherence; they are essentially write-broadcast protocols that switch to write-invalidate protocols when the breakeven point in terms of coherence overhead between two types of protocols is met.

Agarwal *et al.* [4] also used trace-driven simulations to evaluate the performance of directory schemes in a small-scale multiprocessor system with infinite cache sizes. The directory schemes remove the major limitation of the snoopy schemes, the reliance on broadcasts, while they provide similar efficiency in handling shared data accesses. The basic bandwidth limitation of the shared memory and of the directory is mitigated by distributing directories on the processor board. This technique allows the bandwidth to both shared memory and to the directory to scale with the number of processors. A conclusion of the study is that the performance of directory schemes and snoopy schemes is comparable, and the simulations show that most shared writable blocks written into are present in only a small number of other caches. This make broadcast invalidation signals inefficient. They suggested that in each entry of the directory, a small number of pointers to caches containing the block is sufficient.

Gupta and Weber [47, 86] also used trace-driven simulations to examine the invalidation pattern of five parallel algorithms in directory-based shared memory multiprocessor systems with infinite cache sizes. The systems being simulated have 4, 8, and 16 processors and the effect of these patterns on a directory-based protocol is investigated. The five parallel algorithms cover a wide range of applications, including two graph algorithms, a 3-dimensional particle simulator, one logic simulator and a VLSI global router. Shared data in the five parallel algorithms are classified into five categories, which are code and read-only data objects, migratory objects, synchronization objects, mostly-read objects and frequently read-written objects. Their results showed that the number of copies which are invalidated per invalidation is small for most objects except for the synchronization objects. The synchronization objects have a very different invalidation pattern than that of other objects. A write to a synchronization object usually causes invalidations in a large number of caches. Their results also showed that it is possible to scale well-written parallel programs to a large number of processors without an explosion in invalidation traffic.

Yang and Bhuyan [88, 89] used queueing network models to study the cache coherence overhead in multiple-bus multiprocessor systems; we [34, 35] used four different program models which are the Independent Reference Model, the Global Least Recently Used Stack Model, the Multiple Least Recently Used Stack Model and Independent Reference Model with Critical Sections to study the miss ratio and system penalty on shared data accesses; Goosen and Cheriton [46] modeled the program behavior in a shared cache for estimating the load placed on the shared bus by such a shared cache; and Agarwal [2] modified the Dubois-Briggs [30] model by adding the processor locality concept to predict the cache miss rate.

## 1.3 Organization of the Dissertation

The remainder of the dissertation is organized as follows. Chapter 2 proposes a new program model called the access burst program model to study the coherence overhead for shared writable blocks in multiprocessor systems. In this chapter, we start by analyzing a baseline system, the infinite cache model. The infinite cache model isolates the traffic incurred in maintaining coherence; the traffic due to replacement is eliminated. Also, the infinite cache model is much simpler to study than the real system with finite caches, primarily because it is described by fewer parameters. Because of this simplicity, a closed-form solution can be found for the model.

Chapter 3 verifies the accuracy of the access burst program model by comparing the model predictions with the execution-driven simulation results of eight parallel algorithms. The eight algorithms are run on a simulated shared-memory multiprocessor system in which each processor has a private data cache of infinite size. The eight parallel algorithms are the Jacobi iterative [90], the S.O.R. iterative [29], the dynamic quicksort [69], the bitonic merge sort [66], the non-shuffling FFT [19], the shuffling FFT [19], the single source shortest path [26] and the image component labeling [67] algorithms.

Chapter 4 applies the access burst program model to compare the performance of five invalidation-based cache coherence protocols. The protocols are the Basic, the Write-once, the Synapse, the Illinois and the Berkeley cache coherence protocols. Protocols are modeled by Markov chains. An analytical closed-form solution is derived for all components of the cache coherence overhead and for all cache coherence protocols in systems with caches of infinite sizes.

Chapter 5 studies the finite cache effects. Replacements are assumed to be uniformly distributed throughout the whole execution. The system is again modeled by a Markov chain. An approximate solution is found for each component of the coherence overheads. Again, the accuracy of the model is verified by comparing the model predictions with execution-driven simulation results for the eight parallel algorithms.

Finally, Chapter 6 summarizes the research results and outlines future research directions.

# 2 THE ACCESS BURST PROGRAM MODEL

In this chapter, a simple program model for data sharing is introduced and an analytical closed-form solution is found for all components of the cache coherence overhead in a cache coherence protocol. The system being studied has caches of infinite sizes and works in steady state.

## 2.1 General Assumptions

Several simplifying assumptions are made in this chapter. These assumptions are listed and their validity is discussed.

**Assumption 1:** The size of all caches is infinite.
**Assumption 2:** The models are in steady-state. Initial transients are not included.

The major motivation for studying the infinite cache model is its simplicity. Most parameters of the cache do not affect the model prediction, including cache size, cache organization and cache replacement policy and resulting models are therefore parsimonious. Another reason why the infinite cache model is a compelling model is the result of present trends in memory chip sizes indicating that fast and large caches are becoming possible. In these caches, most of the misses are due to the initial loading of the data and to coherence invalidations. It is expected that the infinite cache model will become more and more relevant as the level of integration of memory chips increases.

Modeling transient effects in an infinite cache is not difficult, but the models are not very interesting: As the program starts, caches are empty and every block referenced in a parallel algorithm must be loaded into one of the caches. These initial misses are not included in the models. Their number is simply equal to the total number of different blocks accessed during the entire execution of the parallel algorithm.

## 2.2 The Basic Cache Coherence Protocol

The cache coherence protocol considered in this chapter is a write-invalidate protocol called the Basic cache coherence protocol which is described in Hwang and Briggs's book [49, pages 521–525]. The reason that the Basic cache coherence protocol was chosen in this chapter is because it generates the most complicated cache coherence events. The set of all cache coherence events generated by the rest of the write-invalidate cache coherence protocols is included in the set of events generated by the Basic cache coherence protocol. In the following, a program model is proposed and is solved analytically for the Basic cache coherence protocol. The techniques described in this chapter can be applied to any one of the other write-invalidate cache coherence protocols.

In the Basic cache coherence protocol, multiple copies of the same cache block may be present in different caches, if the copies are Read-Only (RO copies), that is, provided no processor has modified any word in the block. If a processor needs to modify a word in a block, it must obtain a Read-Write copy (RW copy), which is a unique copy of the block and this may involve invalidating copies of the block in other caches. Usually a block

containing only instructions, private data or shared constants will be tagged as RO, while blocks containing shared writable data may be tagged as RW. An S-block contains data items accessed and modified by different processors while a P-block contains data items accessed by only one processor or Read-Only data. In the protocol selected for study in this chapter, the following cache events on an S-block may occur in a multiprocessor systems with infinite caches and in steady-state (refer to Figure 2):

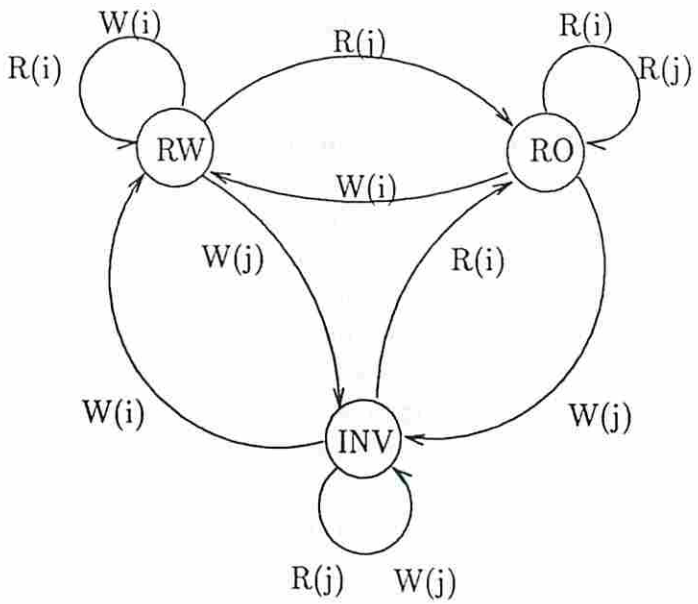



Figure 2: State diagram for a given block in cache i (infinite cache assumption)

R(i) : Read block by processor i.
R(j) : Read block by processor j.
W(i) : Write to block by processor i.
W(j) : Write to block by processor j.

1. *Miss*: this event occurs when the data is referenced and is not present in the cache. We denote this event as $M$ (Miss). All misses occurring as a result of the following events are accounted for as $M$ events. Blocks are always loaded from shared memory on a miss.

2. *Transition from RO to RW*: this event occurs when a processor needs to modify a block already present in another cache as RO; a miss may occur and an invalidation must be sent to the processor(s) possessing the RO copy(ies). We denote this event as $IN\_RO$ (for $IN$validation of $RO$ copy(ies)).

3. *Transition from RW to RO*: this event occurs when a processor reads a block present in another cache as RW. In addition to the occurrence of a miss, a signal must be sent to the cache possessing the RW copy and this cache must write the block back to shared memory. We denote this event as $CS\_RW$ ($C$hange State of a $RW$ copy).

4. *Transition from RW to RW* in a different cache: this event occurs when a cache needs to modify a block which is owned as RW by another cache; it implies a write back to shared memory, a miss and an invalidation. This event is denoted as $IN\_RW$ ($IN$validation of a $RW$ copy).

When writable blocks are actively shared, copies must be transferred among caches and invalidation signals must be sent. As the number of processors actively sharing an S-block increases, the invalidation activity usually increases.

## 2.3 Analytical Models

The access pattern to shared data in multiprocessor systems depends on the algorithm. Synchronization data (such as locks) and other shared operands are two broad classes of shared variables. Synchronization data are used to coordinate process execution or to protect shared operand accesses.

Kung [55] classifies multitasked algorithms into *synchronized* and *asynchronous* algorithms. In *asynchronous* algorithms, accesses to shared operands are not protected and each processor may access the data as it needs them. In *synchronized* algorithms, accesses to shared data are restricted, either by explicit synchronization or simply by structuring the forking and joining of processes. In *synchronized* algorithms, shared writable data are accessed either in **critical sections** [7] (only one process can access the data at a time either to Read or to Write) or in **semi-critical sections** [21] (multiple processors can read a data item at a time, but if a process has to modify the data item, it must do so in mutual exclusion). Figures 3 (a) and (b) illustrate both access patterns. In these figures, only accesses to a specific shared datum are shown.

### 2.3.1 Analytical Model for one S-block

The program model is derived from the model in [75]. We had to extend this model because it did not capture the locality of references to shared writable data. In another paper [28], two additional program models are presented for which the effect of cache coherence can be solved analytically and which take into account the accesses made in critical sections and semi-critical sections. All these program models can be defined as a special case of the following model. The program model that we are about to define assumes that accesses by one processor to a shared writable block are done in uninterrupted bursts. Besides modeling critical section accesses, the access burst model takes into account the locality of references on S-blocks.

The $P$ processors execute independent streams of instructions and generate homogeneous streams of references. S-blocks belong to different *sets*; all S-blocks in a set are accessed with the same pattern, even if they are accessed by different processors. The program model, model parameters and coherence overheads are identical for all the S-blocks in a set.

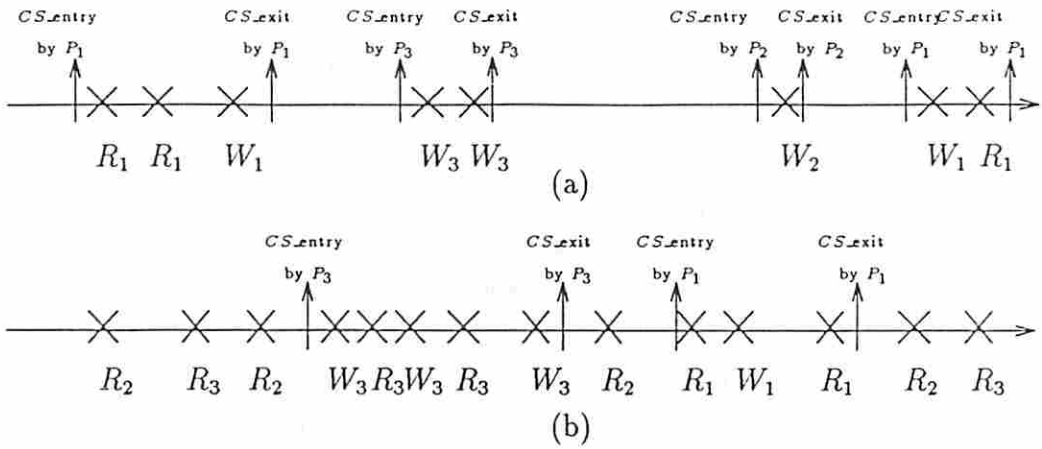



Figure 3: (a) Access pattern to a shared writable datum protected by critical sections. (b) Access pattern to a shared writable datum protected by semi-critical sections.

$R_j$: Read access to shared datum X by processor j
$W_j$: Write access to shared datum X by processor j

Let $q_s$ be the fraction of references to S-blocks. The fraction of S-block accesses that are for a particular S-block $i$ is $p_i$ with $i$=1,...,$N_s$ and $N_s$ is the total number of shared writable blocks. S-block $i$ is shared by $J_i$ processors ($J_i \leq P$, the total number of processors in the system). Processors access an S-block $i$ in bursts. $l_i$ is the average burst size, that is, the average number of accesses to the block during an access burst. An isolated access is counted as a burst of size one. The average burst size can be found by dividing the total number of references by the number of access bursts. For example, for the program fragments of Figure 3 the average burst sizes are $l_i = 2$ (Figure 3(a)) and $l_i = 1.75$ (Figure 3(b)), assuming that one cache block contains only one data element.

The fraction of processor references which start an access burst for a given S-block $i$ is $q_s \cdot p_i/l_i$. The basic approximation of the analytical model is that access bursts are independent from one another. When a processor completes an access burst, all the $J_i$ processors have the same probability of starting the next access burst to the S-block. We designate by $W_i$ the probability that the block is modified during an access burst. Because of the infinite cache assumption, there is no interference among cache accesses to different blocks and the events occurring for one block are independent of the events occurring for any other block; the state transitions of S-block $i$ can be observed in isolation.

The *global state* of an S-block $i$ is described by the number of caches possessing a copy of the block and by the status RO or RW of the block. The global states are denoted by $1\_RW$, $1\_RO$, $2\_RO$,..., $J_i\_RO$. We can ignore the identity of specific processors because the multiprocessor is homogeneous and symmetric.

The Markov chain for the state transitions of S-block $i$ is shown in Figure 4 (we have dropped the index $i$ in the Figure for clarity. Note that all parameters are for a given S-block $i$). The state of the Markov chain is the global state of the block *whenever an access burst is completed* (except for the state $MEM$, which is the state of the block before the first reference to it). It is clear from the Figure 4(a) that states $MEM$ and $1\_RO$ are transient

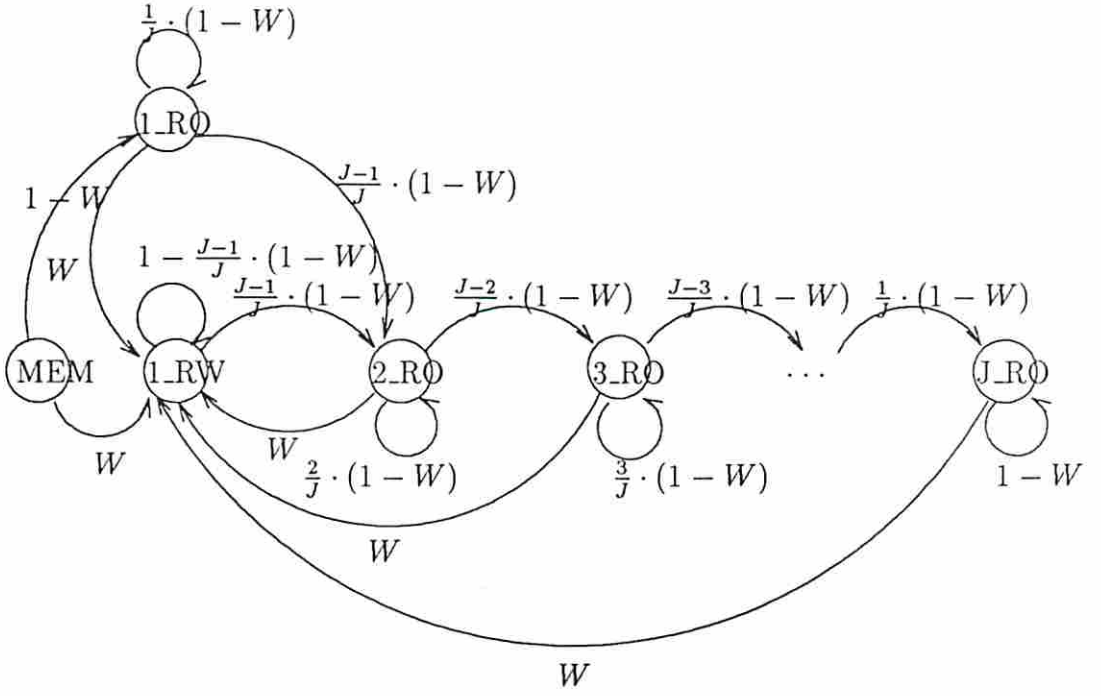



Figure 4: (a)Markov chain for the state transitions of an S-block shared by $J$ processors (including transient states). (b)Markov chain for the state transitions of an S-block shared by $J$ processors (without transient states).

states. Figure 4(b) shows the reduced Markov chain where the transient states have been removed. We will only solve the Markov chain of Figure 4(b). A state transition occurs in this state diagram every time a burst of accesses is completed by one processor. The transition probabilities from state $k\_RO$, $k < J_i$, are found as follows.

1. From state $k\_RO$ to state $k+1\_RO$: The probability of this transition is the product of the probability that the next burst contains only Read accesses, $(1 - W_i)$, and of the probability that the access burst is made in one of the $J_i - k$ other caches, $(\frac{J_i - k}{J_i})$.

2. From state $k\_RO$ to state $k\_RO$: This is the case when the next access burst contains only Read accesses in one of the $k$ caches. The transition probability is $(1 - W_i)\frac{k}{J_i}$.

3. From state $k\_RO$ to state $1\_RW$: This is the case when the next access burst modifies the block. The transition probability is $W_i$.

The transition probabilities from states $1\_RW$ and $J_i\_RO$ are derived from similar arguments.

This finite state Markov chain is aperiodic and irreducible [6]. Denote by $Pr(1)$ and by $Pr(k)$, $k$=2,...,$J_i$, the state probabilities of state $1\_RW$ and states $k\_RO$ respectively. The state probability distribution is given by the set of equations: (see for example [6])

$$Pr(k) = \frac{(J_i - k + 1)(1 - W_i)}{(J_i - k)(1 - W_i) + J_i W_i} Pr(k-1), \quad for\ k = 2, ..., J_i \tag{1}$$

and

$$Pr(1) = \frac{J_i W_i}{(J_i - 1)(1 - W_i) + J_i W_i} \tag{2}$$

With these state probabilities, we can compute the probability of occurrence of each coherence event. When there are $k$ copies in $k$ processor caches a miss occurs at the beginning of a new access burst, that is, at a state transition in Figure 4(b), if the next processor to start an access burst is one of the $(J_i - k)$ processors without a copy in their cache. Therefore, the fraction of references to S-block $i$ which miss in the cache is equal to the fraction of state transitions causing a miss divided by the average burst length $l_i$.

$$Pr(M_i) = \frac{1}{l_i}\left[Pr(1) \cdot \frac{(J_i - 1)}{J_i} + Pr(2) \cdot \frac{(J_i - 2)}{J_i} + ... + Pr(J_i - 1) \cdot \frac{1}{J_i}\right] \tag{3}$$

After some transformations, one finds simply (see Appendix A):

$$Pr(M_i) = \frac{1}{l_i} \cdot \frac{(J_i - 1)W_i}{1 + (J_i - 1)W_i} \tag{4}$$

In the Markov graph of Figure 4(b) a transition from state $1\_RW$ to state $1\_RW$ results from three possible sequences of events:

1. the processor owning the RW copy of the block has started a new burst of accesses for the same block; no event is recorded for S-block $i$ in this case;

2. a different processor has started an access burst for S-block $i$ and its first access to the block is a Write; an event of type $IN\_RW$ must be recorded for S-block $i$;

3. a different processor has started an access burst for S-block $i$ and its first access to the block is a Read followed by a Write; one event of type $CS\_RW$ followed by one event of type $IN\_RO$ must be recorded.

In order to differentiate between the 2nd and the 3rd cases, we have to introduce a new factor $f_i$, which is the fraction of Write bursts[1] so that the first access is a Write. $f_i$ can easily be computed from a string of references. For example, in Figure 3(a), $f_i = 0.75$, and, in Figure 3(b), $f_i = 0.5$. Taking into account this problem, one can derive the fraction of references to S-block $i$ that result in a given event.

An invalidation of RO copies occurs whenever an access burst modifies a block in an RO state. It also occurs in a transition from $1\_RW$ to $1\_RW$, provided the second access burst is executed by a different processor and starts with a Read. Therefore, the fraction of accesses to S-block $i$ invalidating RO copies in other caches is given by:

$$\begin{aligned} Pr(IN\_RO_i) &= \frac{1}{l_i}\left[W_i \cdot (1 - Pr(1)) + W_i \cdot (1 - f_i) \cdot Pr(1) \cdot \frac{J_i - 1}{J_i}\right] \\ &= \frac{1}{l_i} \cdot \frac{(J_i - 1) \cdot W_i \cdot (1 - W_i \cdot f_i)}{J_i - 1 + W_i}. \end{aligned}$$

A change of state from RW to RO occurs whenever a burst leaving the block in state $1\_RW$ is followed by a burst starting with a Read access by a different processor. Therefore, the fraction of references to S-block $i$ changing the state from RW to RO is:

$$\begin{aligned} Pr(CS\_RW_i) &= \frac{1}{l_i}\left[Pr(1) \cdot (1 - W_i) \cdot \frac{J_i - 1}{J_i} + Pr(1) \cdot W_i \cdot (1 - f_i) \cdot \frac{J_i - 1}{J_i}\right] \\ &= \frac{1}{l_i} \cdot \frac{(J_i - 1) \cdot W_i \cdot (1 - W_i \cdot f_i)}{J_i - 1 + W_i}. \end{aligned}$$

An invalidation of a RW copy occurs whenever an access burst leaving the block in state $1\_RW$ is followed by a Write from any other processor. The fraction of references to S-block $i$ causing such an event is therefore:

$$\begin{aligned} Pr(IN\_RW_i) &= \frac{1}{l_i} \cdot Pr(1) \cdot f_i \cdot W_i \cdot \frac{J_i - 1}{J_i} \\ &= \frac{1}{l_i} \cdot \frac{(J_i - 1) \cdot W_i \cdot W_i \cdot f_i}{J_i - 1 + W_i}. \end{aligned}$$

Finally, the number of copies which are invalidated by an invalidation can be computed as follows: Let $X_i$ be the average number of copies for S-block $i$ in the system in steady state, where $X_i = \sum_{k=1}^{k=J_i} k \cdot Pr(k)$. When an invalidation signal for S-block $i$ is broadcast, $X_i - 1$ and $X_i$ copies will be invalidated, in the case of a white hit and of a write miss, respectively. The average number of copies which are invalidated by an invalidation is

$$INV_i = \frac{X_i}{J_i} \cdot (X_i - 1) + \frac{J_i - X_i}{J_i} \cdot X_i$$

[1] By definition, a Write burst is an access burst containing *at least* one Write access.

After some transformations, one finds simply (see [36]):

$$INV_i = \frac{(J_i - 1)}{1 + (J_i - 1) \cdot W_i}.$$

In these equations, $Pr(1)$ is given by equation (2).

### 2.3.2 System Effects

We use the results of the previous section to model the effect of cache coherence on the overall system performance under the assumptions of Section 3.1. Two performance measures are derived: the miss ratio and the average coherence penalty.

Miss ratio

In the infinite cache model, if the transients are not included, the miss rate is given by the miss rate on shared writable blocks:

$$Pr(M) = q_s \cdot \sum_{i=1}^{N_s} p_i \cdot Pr(M_i) \tag{5}$$

where $N_s$ is the total number of shared writable blocks. The value for $\Pr(M_i)$ is obtained by applying equation (4) and depends on different values of the parameters for different S-block $i$. In many cases, the terms in the above sum can be clustered by grouping the shared writable blocks into sets; within a set all blocks are referenced with the same pattern, and therefore have the same value of $\Pr(M_i)$.

To find $\Pr(M)$ from equation (5), we need to specify the model parameters for all sets of blocks. In the studies presented in [30, 10, 80], there is only one set of parameters. Implicitly, it is assumed that the models can be applied to a single *average* set including all the shared writable blocks, and parameters are therefore computed as averages. For the eight examples presented in the next chapter, this approach is shown to be acceptable.

Average Coherence Penalty

A processor runs at maximum speed when no cache misses or coherence events occur. To each coherence event corresponds an average penalty, $\lambda_{EVENT}$. The penalty associated with an event is defined as the average time that a processor is blocked at the occurrence of the event. The average coherence penalty per memory reference to S-block $i$ is:

$$\begin{aligned} \lambda_i = {} & Pr(M_i) \cdot \lambda_M + Pr(IN\_RO_i) \cdot \lambda_{IN\_RO} \\ & + Pr(CS\_RW_i) \cdot \lambda_{CS\_RW} + pr(IN\_RW_i) \cdot \lambda_{IN\_RW}. \end{aligned}$$

If we do not include the transients, the average coherence penalty in the infinite cache system is given by the sum of the coherence penalties on each shared writable block $i$:

$$\lambda_{total} = q_s \cdot \sum_{i=1}^{N_s} p_i \cdot \lambda_i \tag{6}$$

Therefore to compute the system coherence penalty, S-blocks can be clustered into a few sets in which blocks have the same model parameters. The average penalty could also be approximated by the penalty for an *average* block.

The average coherence penalty adversely affects the processor efficiency. In powerful and expensive main frame multiprocessors, any loss of processor efficiency is critical for the performance/cost ratio of the system. If $T_1$ is the mean execution time of an instruction in the uniprocessor system (in microsecond), and if $r$ is the average number of memory references per executed instruction, the average instruction execution time is $T_1 + r\ \lambda_{total}$, and the MIPS rate (Million of Instructions Per Second) per processor is

$$MIPS\ rate = \frac{1}{T_1 + r\lambda_{total}}.$$

The speed-up of the system is therefore equal to

$$Speed - up = \frac{T_1 \cdot P}{T_1 + r\lambda_{total}},$$

where $P$ is the number of processors in the system.

## 2.4 Conclusion

In this chapter, we have presented and solved a simple program model for caching of shared writable blocks in multiprocessor systems. The simplicity and generality of the results stem from the infinite cache hypothesis. The infinite cache model is independent of all cache parameters, such as cache organization and replacement policy. The shared writable block accesses in multiprocessor systems with infinite caches can be modeled by a discrete Markov chain. Analytical closed-form solutions are found for all cache coherence components. In the next chapter, we will validate the model by comparing the model predictions with execution-driven simulation results of several parallel algorithms.

# 3 MULTITASKED ALGORITHMS

In this chapter, the accuracy of the access burst program model is verified by comparing the model predictions with the execution-driven simulations of eight parallel algorithms for shared-memory multiprocessors in which each processor has a private data cache of infinite size.

To simulate the parallel algorithms, a simulation methodology described in [31] was applied. In this methodology, the algorithm is actually executed on a uniprocessor and the multiprocessing effect is obtained by executing the process of each simulated processor in turn. The simulator *switches* from one simulated processor to another on each data access and synchronization primitive execution.

The eight parallel algorithms are Jacobi iterative, Successive Over Relaxation iterative, dynamic quicksort, bitonic merge sort, non-shuffling FFT, shuffling FFT, single source shortest path and image component labeling algorithms. The class of algorithms selected is a class of iterative algorithms which has been the subject of many other papers (for example, see [22]). These algorithms exhibit shared data contention and can be easily mapped to the analytical model of the previous section.

## 3.1 Relaxation Algorithms for Solving Partial Differential Equations

In this section, we consider two iterative schemes [90], the Jacobi and the Successive Over Relaxation (S.O.R.) algorithms, to solve Laplace's equation $\nabla^2 x = 0$ on a rectangular domain of $R^2$.

### 3.1.1 The Jacobi Iterative algorithm

We apply the model to one particular multitasked algorithm, the Jacobi iterative algorithm [90] which is an iterative, compute-intensive algorithm. The algorithm consists of updating the points of a rectangular grid in each iteration. Let $L \times N$ be the size of the rectangular grid where $N$ is the horizontal dimension and $L$ is the vertical dimension. The grid is divided into $P$ partitions, where $P$ is the number of processors. $P = q \cdot p$, where $p$ and $q$ are the number of partitions on the horizontal and the vertical sides of the grid. We assume that $P$ is a power of 2. If $P$ is a square number, we choose $p = q = \sqrt{P}$; otherwise, we choose $q = \sqrt{P/2}$ and $p = P/q = \sqrt{2 \cdot P}$. This partitioning strategy is shown in Figure 5 and 6. The boundary conditions add a total of $2 \cdot (L + N)$ grid points, so that the total number of grid points is actually $L \cdot N + 2 \cdot (L + N)$, and these points are not modified during execution.

The Jacobi iterative algorithm requires maintaining two grids. In each iteration, every point of a grid is updated by using the values of the four neighbors in the other grid and after each iteration, the processors have to synchronize. In general, a processor has to synchronize with no more than four neighbors while at the same time the two grids are interchanged [37]. In the algorithm, the equation for the update of an iterate in the $(K + 1)$th iteration is:

$$x_{i,j}^{(K+1)} = \frac{1}{4} \cdot \left[ x_{i+1,j}^{(K)} + x_{i-1,j}^{(K)} + x_{i,j+1}^{(K)} + x_{i,j-1}^{(K)} \right].$$

In a multiprocessor system, a natural decomposition of this problem is to allocate one partition of the grid points to each processor. Figure 5 illustrates the case of an $8 \times 8$ grid and four processors. In the case of cache block size equal to one ($B$=1)[1], there are $2 \cdot (8+8)$ boundary grid points which are Read-Only data elements and can be treated as P-blocks. The grid points adjacent to these boundary points are called *outer* grid points. The reference pattern to outer grid points is different than the pattern to *inner* grid points. For the inner grid points, the S-blocks can be further divided into two sets according to the number of processors accessing the S-blocks. The sets of S-blocks are circled in Figure 5. There are only three sets of S-blocks in this figure: (1) inner grid points with $J = 2$, (2) outer grid points with $J = 2$, and (3) inner grid points with $J = 3$.

The infinite cache condition is met when the data cache of each processor is large enough to contain all the grid elements accessed by the processor. In this case, steady-state is reached after the first iteration. This important algorithm is therefore a good benchmark to apply the model. Shared writable data are accessed in semi-critical sections in the Jacobi iteration algorithm. In one iteration, the S-blocks in one grid are Read-Only and are accessed by different processors, and in the next iteration they are modified by a single processor in a critical section phase. The parameters of the model are therefore easily derived. Consider, for example, a grid size of $128 \times 128$, $P = 4$ and $B$=1; the parameters of each set can be computed as follows:

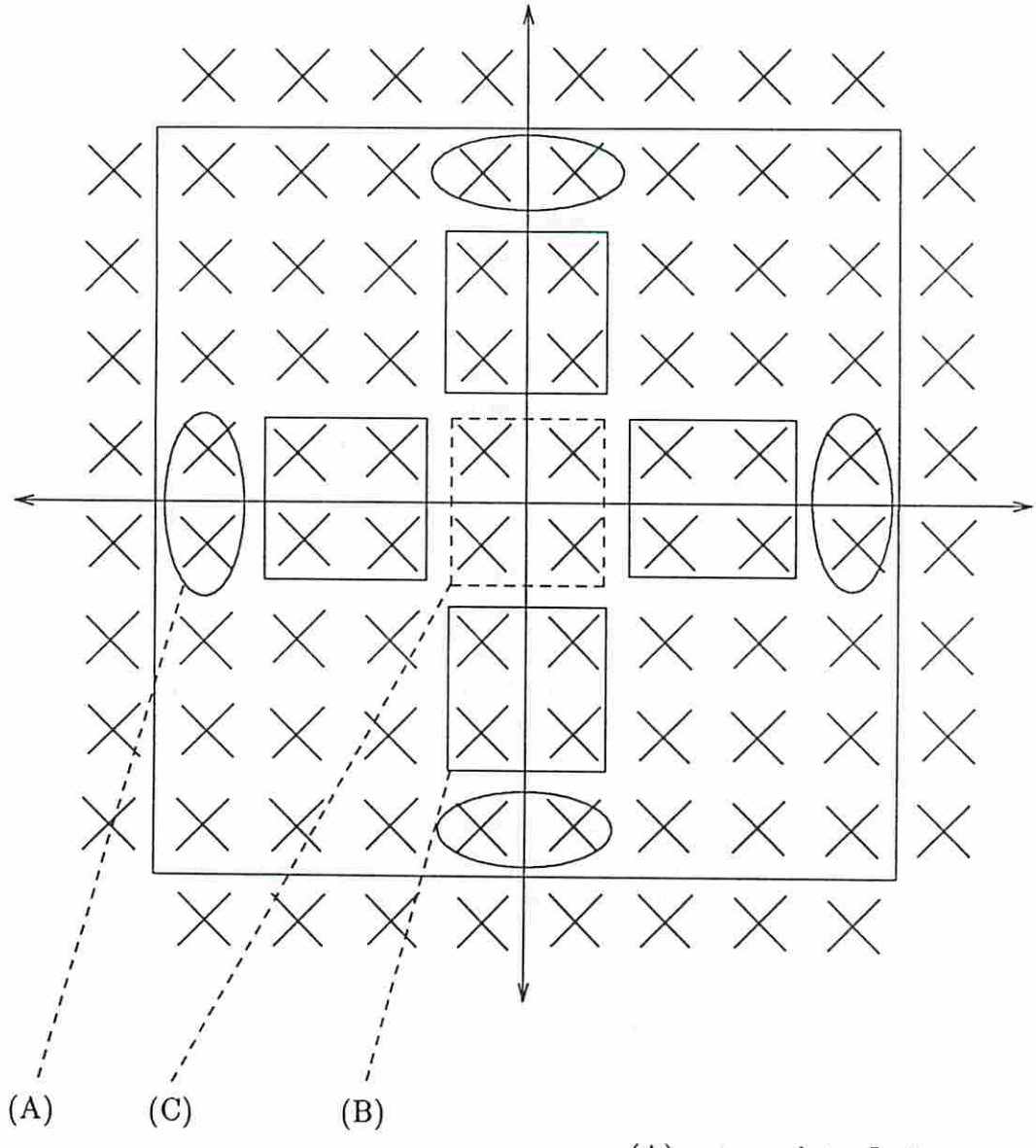
1. For the inner grid points with $J$=2, the total number of accesses in a multiple Read phase is four and in the next iteration (critical section phase), the total number of accesses is one Write. For every two iterations, there are five accesses and five bursts (one of the five bursts is a Write burst); the values of the parameters are $J$=2, $l$=1, and $W=\frac{1}{5}$. Since there is only one access in the Write burst, the value of the parameter $f$ is equal to one.

2. For the outer grid points with $J$=2, the reference pattern is very much like the previous set except that the total number of accesses in the multiple Read phase is three since the boundary points need not be updated during the execution. Hence, the values of the parameters are $J$=2, $l$=1, $W=\frac{1}{4}$ and $f$=1.

3. For the inner grid points with $J$=3, the behavior is exactly the same as that of the set of inner grid points with $J$=2. The values of the parameters are $J$=3, $l$=1, $W=\frac{1}{5}$ and $f$=1.

Table 1 shows the computed value of the model parameters, where the column labeled $p_s$ contains the fraction of processor references to the S-blocks in the set, that is, $p_s = q_s \sum_{set} p_i$; the column labeled $n_s$ contains the total number of S-blocks in the set. The *miss* and *penalty* columns contain the total contribution of the data set to the overall data miss ratio and to the average coherence penalty, that is, miss=$p_s \cdot M_i$ and penalty=$p_s \cdot \lambda_i$, where $i$ refers to any S-block in the set. The unit for the penalties is the average penalty for a miss. We have chosen the following penalties for each event: $\lambda_M = \lambda_{CS\_RW} = \lambda_{IN\_RW} = 0.75 + 0.25 \cdot B$, $\lambda_{IN\_RO} = 0.5$ (remember that events $CS\_RW$ and $IN\_RW$ cause one miss and one write-back access to occur). From Table 1, it appears that the inner grid points shared by two

[1] $B$=1 stands for a cache block containing a data element.




(A) outer points, J=2

(B) inner points, J=2

(C) inner points, J=3

Figure 5: The three data sets in the Jacobi iteration ($B$=1).

processors dominate the overall miss ratio and average coherence penalty. Table 1 shows the correlation between model results and simulated values.

Table 1: Jacobi Iterative Algorithm ($P$=4, 128 × 128 grid, $B$=1)

| Set | $J$ | $W$ | $l$ | $f$ | $p_s$ | $n_s$ | | miss | penalty |
|---|---|---|---|---|---|---|---|---|---|
| Inner | 2 | 0.200 | 1 | 1 | 0.03027 | 992 | model | 0.00505 | 0.01211 |
| | | | | | | | simulation | 0.00605 | 0.01261 |
| Outer | 2 | 0.250 | 1 | 1 | 0.00039 | 16 | model | 0.00008 | 0.00019 |
| | | | | | | | simulation | 0.00010 | 0.00020 |
| Inner | 3 | 0.200 | 1 | 1 | 0.00024 | 8 | model | 0.00007 | 0.00013 |
| | | | | | | | simulation | 0.00010 | 0.00014 |

In the Jacobi iterative algorithm, when the cache block size increases, the number of sets of S-blocks also increases. For instance, there are five sets of S-blocks when $B$=2, eight sets of S-blocks when $B$=4, and ten sets of S-blocks when $B$=8.

Consider the Jacobi iterative algorithm with a cache block size of four data elements and a grid size of 128 × 128. Arrays are stored row-wise. Figure 6 shows eight different types of S-blocks, named type 1 to type 8. There are 248 type 1 S-blocks and each type 1 S-block is referenced 20 times in two iterations (four Reads and one Write per data element). The total number of references in an iteration is 163,840 ($128 \cdot 128 \cdot 5 \cdot 2$) and the value of $p_s$ for type 1 S-blocks is 0.003027. Type 1 S-blocks are shared by two processors, that is $J = 2$. The reference string to a type 1 S-block contains 20 isolated Reads in one iteration (multiple Read phase), and two Write bursts in the next iteration (critical section). The two iterations alternate. Therefore, the total number of access bursts to the S-block for the four elements is 18. The value of $W$ is $2/18 = 0.1111$ and $l$ is $20/18 = 1.1111$. Since the first reference in a Write burst of type 1 S-block is always a Write, the value of $f$ is one. Similarly, we can compute the value of these parameters for type 2 to type 8 S-blocks. Table 2 to 5 lists the values of parameters for $P$=4, grid size of 128 × 128, and $B$=2, $B$=4, $B$=8, and $B$=16, respectively. Table C.1 in Appendix C displays the average values of parameters for different cache block sizes.

Table 2: Values of parameters for the five different sets of S-blocks in the case of the Jacobi iterative algorithm with a grid size of 128 $X$ 128 ($B$=2).

| Set | $p_s$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| Type 1 | 0.01514 | 2.0000 | 0.2000 | 1.0000 | 1.0000 |
| Type 2 | 0.00020 | 2.0000 | 0.2500 | 1.0000 | 1.0000 |
| Type 3 | 0.01514 | 2.0000 | 0.1111 | 1.1111 | 1.0000 |
| Type 4 | 0.00024 | 2.0000 | 0.2000 | 1.0000 | 1.0000 |
| Type 5 | 0.00024 | 4.0000 | 0.2000 | 1.0000 | 1.0000 |

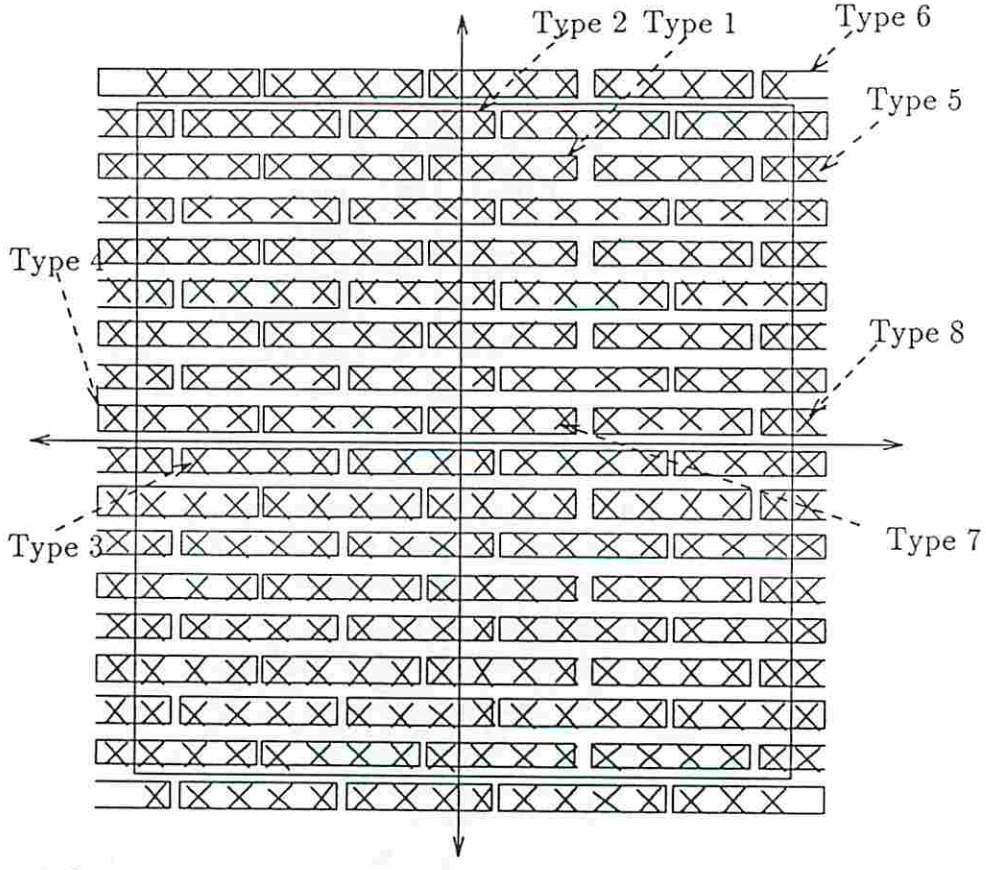



Figure 6: The eight sets of S-blocks in the Jacobi iteration when $B$ is equal to four ($M$=16, $P$=4).

Table 3: Values of parameters for the eight different sets of S-blocks in the case of the Jacobi iterative algorithm with a grid size of 128 $X$ 128 ($B$=4).

| Set | $p_s$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| Type 1 | 0.03027 | 2.0000 | 0.1111 | 1.1111 | 1.0000 |
| Type 2 | 0.00041 | 2.0000 | 0.1429 | 1.1429 | 1.0000 |
| Type 3 | 0.01465 | 2.0000 | 0.0588 | 1.1765 | 1.0000 |
| Type 4 | 0.00037 | 2.0000 | 0.0769 | 1.1538 | 1.0000 |
| Type 5 | 0.00757 | 2.0000 | 0.2000 | 1.0000 | 1.0000 |
| Type 6 | 0.00012 | 2.0000 | 0.2000 | 1.0000 | 1.0000 |
| Type 7 | 0.00049 | 4.0000 | 0.1111 | 1.1111 | 1.0000 |
| Type 8 | 0.00012 | 4.0000 | 0.2000 | 1.0000 | 1.0000 |

Table 4: Values of parameters for the ten different sets of S-blocks in the case of the Jacobi iterative algorithm with a grid size of 128 $X$ 128 ($B$=8).

| Set | $p_s$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| Type 1 | 0.06055 | 2.0000 | 0.0588 | 1.1765 | 1.0000 |
| Type 2 | 0.00078 | 2.0000 | 0.0769 | 1.2308 | 1.0000 |
| Type 3 | 0.01367 | 2.0000 | 0.0303 | 1.2121 | 1.0000 |
| Type 4 | 0.00043 | 2.0000 | 0.0345 | 1.2069 | 1.0000 |
| Type 5 | 0.03406 | 2.0000 | 0.0769 | 1.1538 | 1.0000 |
| Type 6 | 0.00031 | 2.0000 | 0.0476 | 1.1905 | 1.0000 |
| Type 7 | 0.00011 | 2.0000 | 0.1111 | 1.1111 | 1.0000 |
| Type 8 | 0.00037 | 3.0000 | 0.0769 | 1.1538 | 1.0000 |
| Type 9 | 0.00098 | 4.0000 | 0.0588 | 1.1765 | 1.0000 |
| Type 10 | 0.00037 | 4.0000 | 0.0769 | 1.1538 | 1.0000 |

Table 5: Values of parameters for the twelve different sets of S-blocks in the case of the Jacobi iterative algorithm with a grid size of 128 $X$ 128 ($B$=16).

| Set | $p_s$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| Type 1 | 0.12109 | 2.0000 | 0.0303 | 1.2121 | 1.0000 |
| Type 2 | 0.00156 | 2.0000 | 0.0400 | 1.2800 | 1.0000 |
| Type 3 | 0.01172 | 2.0000 | 0.0154 | 1.2308 | 1.0000 |
| Type 4 | 0.00046 | 2.0000 | 0.0164 | 1.2295 | 1.0000 |
| Type 5 | 0.09271 | 2.0000 | 0.0345 | 1.2069 | 1.0000 |
| Type 6 | 0.00040 | 2.0000 | 0.0189 | 1.2264 | 1.0000 |
| Type 7 | 0.00010 | 2.0000 | 0.0588 | 1.0000 | 1.0000 |
| Type 8 | 0.00030 | 2.0000 | 0.0244 | 1.1951 | 1.0000 |
| Type 9 | 0.00020 | 2.0000 | 0.0345 | 1.1379 | 1.0000 |
| Type 10 | 0.00128 | 3.0000 | 0.0345 | 1.2069 | 1.0000 |
| Type 11 | 0.00195 | 4.0000 | 0.0303 | 1.2121 | 1.0000 |
| Type 12 | 0.00085 | 4.0000 | 0.0345 | 1.2069 | 1.0000 |

In the case of an $M \times M$ Jacobi array, when the cache block size exceeds $\frac{M}{\sqrt{P}}+1$, processors update S-blocks alternately, and the number of references $l$ in each burst is equal to one; $J$ can be as high as $2 \cdot \sqrt{P}$ or even $P$, $W = \frac{1}{5}$, and $f$=1 [36].

We have applied the above analysis for block sizes from 1 to 16, $P$=4, and for a grid size of 128 × 128. Figures 16 and 17 show the comparison between model predictions (dotted curve) and simulation results (plain line) for the system miss ratio and the system penalty. These curves are valid for any number of processors $P$ provided $B < \frac{M}{\sqrt{P}} + 1$ [36]. These two Figures show that the analytical program model can make very good predictions for the Jacobi iterative algorithm when the cache block size is greater than two (error is less than 11%).

### 3.1.2 The S.O.R. iterative algorithm

In the S.O.R. iterative algorithm [29, 50], the data decomposition and allocation are the same as those in the Jacobi iterative algorithm. However, only one copy of the grid is needed and iterates are updated according to the red/black ordering: grid elements which are in even positions (the sum of the indexes is even) are tagged as black, others as red and each iteration proceeds in two sweeps. The red elements are updated in the first sweep, and the black elements are updated in the second sweep. After each sweep, each processor has to synchronize with no more than four neighbors. Each processor has the same number of red and of black iterates. The equation for the update of an iterate in the (K+1)th iteration is

$$x_{i,j}^{(K+1)} = (1 - \omega)\, x_{i,j}^{(K)} + \frac{1}{4}\,\omega\,\left[x_{i+1,j}^{(K)} + x_{i-1,j}^{(K)} + x_{i,j+1}^{(K)} + x_{i,j-1}^{(K)}\right],$$

where $\omega$ is the relaxation factor.

In systems with $B$=1, during one sweep of the algorithm, some shared grid points are read by multiple processors, and some others are read and modified by one processor. As for the Jacobi iterative algorithm, there are three sets of shared writable blocks. The values of parameters are derived as follows:

1. For the inner grid points with $J$=2, the total number of accesses in a multiple Read phase is four and in the next sweep (critical section phase), the accesses are a Read followed by one Write. That is, for every iteration, there are six accesses and five bursts (one of the five bursts is a Write burst). Therefore, the values of the parameters are $J$=2, $l$=$\frac{6}{5}$, and $W$=$\frac{1}{5}$. Since the first access in the Write burst is always a Read, the value of the parameter $f$ is equal to zero.

2. For the outer grid points with $J$=2, the reference pattern is very much like the previous set except that the total number of accesses in the multiple Read phase is three since the boundary points need not be updated during the execution. Hence, the values of the parameters are $J$=2, $l$=$\frac{5}{4}$, $W$=$\frac{1}{4}$ and $f$=0.

3. For the inner grid points with $J$=3, the behavior is exactly the same as that of the set of inner grid points with $J$=2. The values of the parameters are $J$=3, $l$=$\frac{6}{5}$, $W$=$\frac{1}{5}$ and $f$=0.

Table 6 displays the comparison between the model predictions and the simulation results of the S.O.R. iterative algorithm. Again, it appears that the set containing the inner grid points shared by two processors dominates the miss and penalty results. There is consensus between the model predictions and the exact values.

Table 6: S.O.R. iterative algorithm ($P$=4, 128 $\times$ 128 grid, $B$=1)

| Set | $J$ | $W$ | $l$ | $f$ | $p_s$ | $n_s$ | | miss | penalty |
|---|---|---|---|---|---|---|---|---|---|
| Inner | 2 | 0.200 | 1.20 | 0 | 0.03027 | 496 | model | 0.00420 | 0.01009 |
| | | | | | | | simulation | 0.00505 | 0.01261 |
| Outer | 2 | 0.250 | 1.25 | 0 | 0.00041 | 8 | model | 0.00007 | 0.00015 |
| | | | | | | | simulation | 0.00008 | 0.00021 |
| Inner | 3 | 0.200 | 1.20 | 0 | 0.00024 | 4 | model | 0.00006 | 0.00011 |
| | | | | | | | simulation | 0.00008 | 0.00014 |

In the S.O.R. algorithm for any cache block size the number of sets of S-blocks are the same as in the Jacobi iterative algorithm. However, the reference patterns to the S-blocks in the sets are different. When the cache block size is greater than one, an S-block is read and written in every sweep because red and black grid elements are mixed in a cache block. Table C.3 in Appendix C shows the average values of parameters for different cache block sizes.

In the case of an $M \times M$ S.O.R. array, when the cache block size exceeds $\frac{M}{\sqrt{P}}+1$, S-blocks are updated alternately by different processors. In this case, $J$ can be as high as $2 \cdot \sqrt{P}$ or even $P$, $W = \frac{1}{6}$, $l$=1, and $f = 1$.

Figures 18 and 19 illustrate the results of the system miss ratios and system penalities for different cache block sizes for a 128 $\times$ 128 grid. These curves are independent of the number of processors provided $B < \frac{M}{\sqrt{P}} + 1$.

## 3.2 Parallel Sorting

### 3.2.1 Dynamic Quicksort

Quicksort [69] is a *divide-and-conquer* algorithm, which sorts a file A[1], A[2], ..., A[$N$] by rearranging it to make the condition that A[1], ..., A[j-1] $\leq$ A[j] $\leq$ A[j+1], ..., A[$N$] hold for some j, and by recursively applying the same procedure to the subfiles A[1], ..., A[j-1] and A[j+1], ..., A[$N$]. At the end of each splitting phase, in a multiprocessor system, the larger subfile is processed by the same processor and a descriptor of the smaller subfile is stored into a global job queue. An idle processor keeps on checking the global job queue and grabs a subfile descriptor when the global job queue is not empty.

In this algorithm, while a processor splits a subfile, no other processor accesses any data item in the subfile and shared data are therefore accessed in critical sections. Every data item in the file may be accessed by all processors.

Table C.5 in Appendix C illustrates the average values of parameters for different cache block sizes. Results are shown for a two processor, four processor and eight processor systems

with data file size of 32,768. Each of the simulated points is an average result of simulation runs for ten independent random files. The values of the parameters are average values and obtained from the simulator.

Figures 20 and 21 show the system miss ratio and the system penalty obtained from the model predictions (dotted line) and from the simulation results (plain line). Even though the relative error in these two Figures is large when the number of processors is two, the model is very good at predicting the general trend.

### 3.2.2 Bitonic Merge Sort

In this section, we present Batcher's bitonic merge sort algorithm [66], which has been the basis for many other parallel sorting algorithm. The fundamental operation of the algorithm is called *compare-exchange* in which two numbers are routed into a comparator, and exchanged according to the sign of the comparator so that they are in the proper order. Given a bitonic sequence[2], a single compare-exchange step divides the sequence into two bitonic sequences of half the length. Applying this step recursively yields a sorted sequence, which can be thought of as half a bitonic sequence of twice the length. In other words, a bitonic merge transforms a bitonic sequence into a set of two bitonic sequences with the property that every element in the first sequence is smaller than any element in the second sequence. Therefore, once the entire set of $N$ elements has been transformed into a single bitonic sequence, the sorting is finished.

Table C.7 in Appendix C illustrates the average values of parameters for different cache block sizes. As for the dynamic quicksort algorithm, results are shown for two processor, four processor and eight processor systems with data file size of 32,768. Each of the simulated point is an average result of simulation runs for ten independent random files. The values of the parameters are obtained from the simulator and are average values.

Figures 22 and 23 show the system miss ratio and the system penalty obtained from the model predictions (dotted line) and from the simulation results (plain line). Again, these two figures show almost total agreement between model predictions and simulation results.

## 3.3 Fast Fourier Transform

In this section, we consider two different algorithms to evaluate the discrete Fourier transform (DFT), the non-shuffling FFT and the shuffling FFT algorithms [19].

### 3.3.1 Non-Shuffling FFT Algorithm

The one-dimensional non-shuffling FFT algorithm [19] for $N$ data items is represented by a butterfly graph with $log_2 N$ stages. A bit-reversal permutation is applied at some point of the algorithm, so that the results are stored in the same order as the initial data items. Let

[2]A bitonic sequence is a sequence of numbers A[0], A[1], ..., A[$N$-1] with the property that (1) there exists an index $i$, $0 \leq i \leq N-1$, such that A[0] through A[$i$] is monotonically increasing and A[$i$] through A[$N-1$] is monotonically decreasing, or else (2) there exists a cyclic shift of indices so that the first condition is satisfied [66].

$s(k)$, $k$=0,1,2,...,$N$-1 be $N$ samples of a time function. The DFT of $s(k)$ is defined to be the discrete function $x(j)$, $j$=0,1,2,...,$N$-1, and

$$x(j) = \sum_{k=0}^{N-1} s(k)\, e^{\frac{2\,\pi\,i\,j\,k}{N}}$$

where $j$=0,1, ..., $N$-1 and $i = \sqrt{-1}$.

In the non-shuffling FFT algorithm, we divide the array of $N$ items into $P$ chunks containing $\frac{N}{P}$ consecutive items. Each processor computes the FFT for its chunk, containing $\frac{N}{P}$ data items. For $N$=16 and $P$=4, the non-shuffling FFT algorithm is illustrated in Figure 7. In general, each S-block is shared by $(log_2 P + 1)$ processors and the algorithm can be divided into two parts. In the first part, that is, in the first $log_2 \frac{N}{P}$ stages of the butterfly, every shared block is accessed by one processor. In the second part, that is, in each of the last $log_2 P$ stages of the butterfly, each shared block is first read by two processors and then modified by a single processor in a critical section. Synchronization is necessary in this algorithm and is denoted by dotted lines in Figure 7. In general, 2 $log_2 P$ synchronization points are needed in the second part (if the algorithm uses two copies of the array and alternates between the copies only $log_2 P$ synchronization points would be needed [36].)

There is only one set of shared writable blocks in this algorithm. Each S-block is shared by $(log_2 P + 1)$ processors, $J = log_2 P + 1$. In the first part of the algorithm, computation can be done locally, where every S-block is accessed in a critical section. In the second part of the algorithm, for every butterfly computation stage, each S-block access can be mapped into the semi-critical section case. In the multiple Read phase, each S-block is read $2 \cdot B$ times, and then followed by a critical section with $B$ Writes. The total number of bursts is therefore $[1 + (2 \cdot B + 1) \cdot log_2 P]$. The values of parameter $W$ is

$$W = \frac{log_2 P + 1}{1 + (2 \cdot B + 1) \cdot log_2 P}.$$

Note that the references of an S-block in the first part of the algorithm are in a Write burst which starts with a Read. The total number of accesses to an S-blocks is $(3 \cdot B \cdot log_2 N)$. Therefore, the value of parameter $l$ is

$$l = \frac{3 \cdot B \cdot log_2 N}{1 + (2 \cdot B + 1) \cdot log_2 P}.$$

The total number of Write bursts is $(log_2 P + 1)$ and all Write bursts in the second part of the algorithm start with a Write. The value of parameter $f$ is thus equal to

$$f = \frac{log_2 P}{log_2 P + 1}.$$

Table C.9 in Appendix C illustrates the average values of the parameters for different cache block sizes. Results are shown for two processor, four processor and eight processor systems with a data file size of 65,536.

Figures 24 and 25 show the system miss ratio and the system penalty obtained from the model predictions (dotted line) and from the simulation results (plain line).

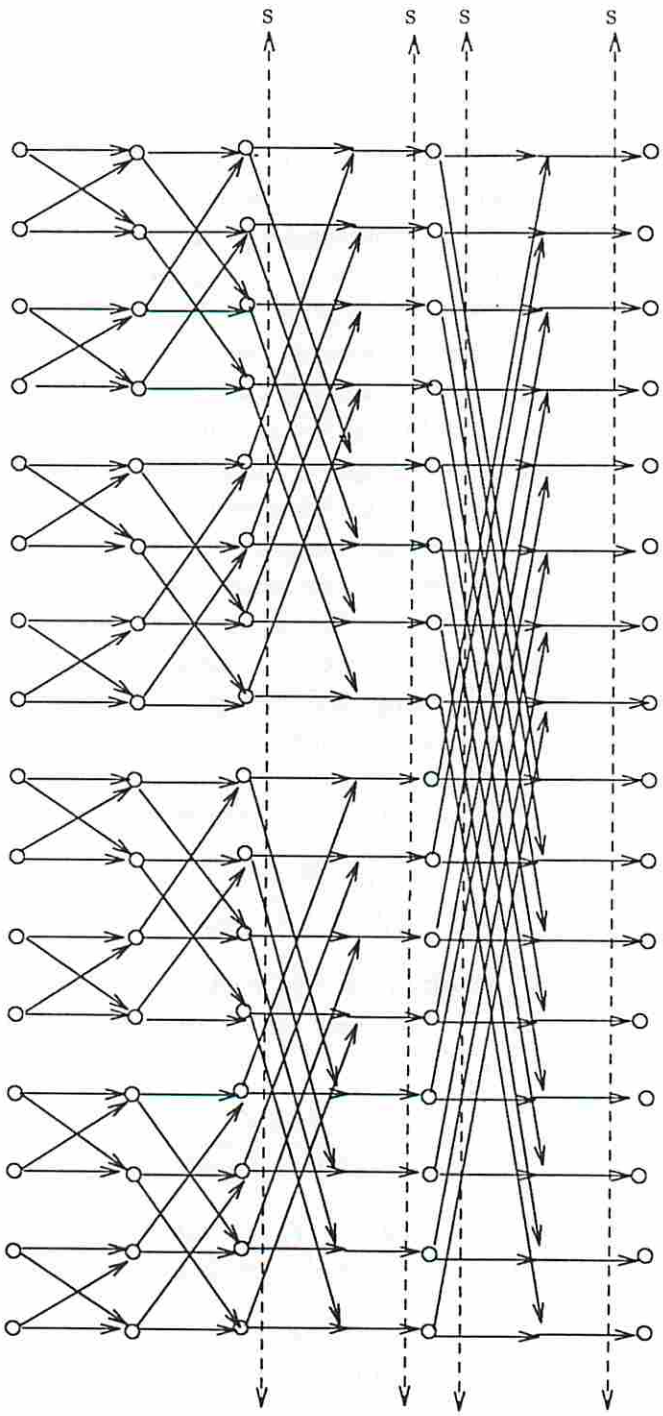



Figure 7: Non-shuffling FFT algorithm for $P$=4 and $N$=16.

### 3.3.2 Shuffling FFT Algorithm

Another algorithm for FFT in multiprocessor systems is the shuffling FFT [19]. In this algorithm, computations of partial FFTs alternate with shuffling stages in which data are passed among processors. Only two processors can share an S-block. Figure 8 presents the shuffling FFT algorithm for an example where $N$=16 and $P$=4. There is much more locality in this algorithm. Coherence activity is reduced significantly, making this algorithm more suitable for cache-based systems.

During each butterfly computation and each shuffling stage, each shared block may be read and updated by only one processor. There is only one set of shared writable blocks in this algorithm and each block is shared by two processors when $B < N/P$ [36], that is, $J$=2. The total number of butterfly/shuffling stages is $2 \lceil \frac{log_2 N}{log_2 \frac{N}{P}} \rceil$. In the algorithm, Writes always occur in the butterfly computation phases and never occur in the shuffling phases. Because of the alternation between Reading and Writing phases, the value of parameter $W$ is

$$W = \frac{\lceil \frac{log_2 N}{log_2 \frac{N}{P}} \rceil + 1}{2 \lceil \frac{log_2 N}{log_2 \frac{N}{P}} \rceil + 1}.$$

where the '1' in the formula corresponds to the Write operation at the end (see Figure 8). In each butterfly computation, there are $[B \cdot (3\ log_2 \frac{N}{P} + 1)]$ accesses to the S-block and in the shuffling phase there are $B$ Reads. The total number of accesses to the S-block is thus equal to $\{B \cdot [(3\ log_2 \frac{N}{P} + 2) \cdot \lceil \frac{log_2 N}{log_2 \frac{N}{P}} \rceil + 1]\}$. The value of parameter $l$ is equal to

$$l = \frac{B \cdot [(3\ log_2 \frac{N}{P} + 2) \cdot \lceil \frac{log_2 N}{log_2 \frac{N}{P}} \rceil + 1]}{2 \lceil \frac{log_2 N}{log_2 \frac{N}{P}} \rceil + 1}.$$

Since the first operation in the butterfly computation stage always starts with a Write except the first butterfly computation stage, the value of parameter $f$ is equal to

$$f = \frac{\lceil \frac{log_2 N}{log_2 \frac{N}{P}} \rceil}{\lceil \frac{log_2 N}{log_2 \frac{N}{P}} \rceil + 1}.$$

Table C.11 in Appendix C shows the average values of parameters for different cache block sizes. As for the Non-shuffling FFT, results are shown for two processor, four processor and eight processor systems with data file size of 65,536.

Figures 26 and 27 show the system miss ratio and the system penalty obtained from the model predictions (dotted line) and from the simulation results (plain line). Again, these two figures show conclusive agreement between model predictions and simulation results.

## 3.4 Single Source Shortest Path Algorithm

The single source shortest path algorithm is the most commonly encounted problem in the study of transportation and communication networks [26]. In this algorithm, a breadth-first search is employed for finding shortest paths from a single vertex (vertex *source*) to all other

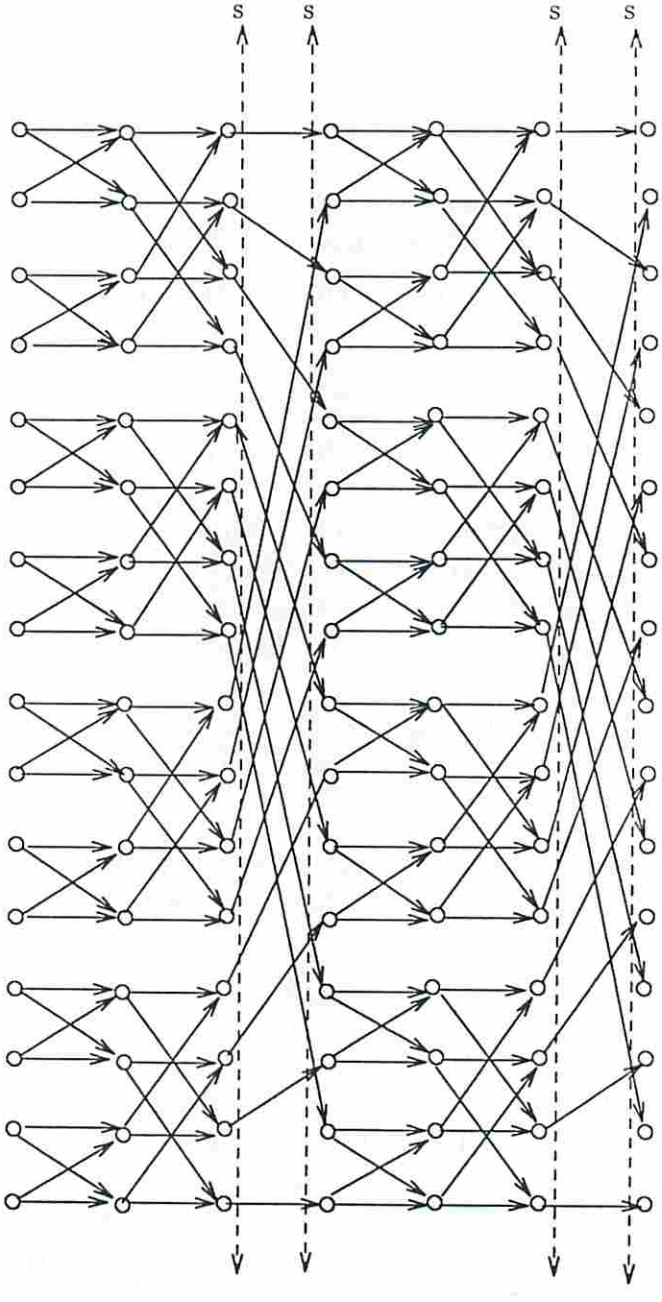



Figure 8: Shuffling FFT algorithm for $P$=4 and $N$=16.

vertices in a directed graph, where each edge $(x,y)$ has a positive length. If vertex $x$ and vertex $y$ do not connect directly, the length of the edge $(x,y)$ is infinite. Also, the length of the edge $(x,x)$ is infinite, that is, there is no self loop in the graph. The length of a path is the sum of the lengths of edges on the path, and the distance of a vertex, $D[v]$ for vertex $v$, is the minimum length of any path to that vertex from vertex *source* and $D[source]=0$.

At the beginning of the algorithm, a MASTER process creates $K-1$ concurrent processes called WORKER processes. During the execution of the algorithm, the $K$ processes share the computation load as long as there are tasks to be performed. In the path finding step, each process repeatedly deletes a node from a queue, $Q$, and updates the distance array (array $D$)[3] and the path array (array $P$)[4]. In addition to a WORKER's tasks, the MASTER process is responsible for finishing the initialization step and for synchronizing the initiation and termination of the algorithm.

Table C.13 in Appendix C illustrates the average values of parameters for different cache block sizes for the distance array (array $D$). Results are shown for two processor, four processor and eight processor systems. Graphs are random graphs of 128 nodes and the average connectivity of each node is 96. Each of the simulation results is an average result of simulation runs for ten independent random files.

Figures 26 and 27 show the system miss ratio and the system penalty for distance array obtained from the model predictions (dotted line) and from the simulation results (plain line).

Every element in the array $D$ has to be accessed in critical sections. Table C.15 in Appendix C illustrates the average values of parameters for the lock array associated with the distance array (lock D). Figures 28 and 29 show the system miss ratio and the system penalty for the lock array obtained from the model predictions (dotted line) and from the simulation results (plain line).

## 3.5 Imaging Component Labeling Algorithm

In this section, we study an intermediate level task in image processing, labeling the connected components of a binary image [67]. The $N \times N$ image pixels are stored in an array which is partitioned and allocated to $P$ processors as in the Jacobi iterative algorithm.

The image component labeling algorithm is also known as the *connected ones algorithm* which consists of associating labels with 1 valued pixels of a binary image such that any two pixels have the same label if and only if they lie in the same connected component, which is a maximal region of 1 valued pixels. In this algorithm, we adopt the *8-connectedness*, which is two pixels are connected if they are adjacent vertically, horizontally and diagonally. After each iteration, the processors have to synchronize. In general, a processor has to synchronize with no more than four neighbors.

The infinite cache condition is met when the data cache of each processor is large enough to contain all the pixels accessed by the processor. In this case, steady state is reached after the first iteration.

[3]The distance array reflects the currently computed shortest distances. At the end of the computation, it stores the shortest distances from the source node to all nodes.

[4]The path array stores those paths which correspond to the distances recorded in the distance array.

Table C.17 in Appendix C illustrates the average values of parameters for different cache block sizes, where 3/4 of pixels are 1 valued pixels. Results are shown for a four processor system with array size of 128 × 128. Each of the simulation results is an average result of simulation runs for ten independent random files.

Figures 30 and 31 show the system miss ratio and the system penalty obtained from the model predictions (dotted line) and from the simulation results (plain line). Again, these two figures show close agreement between model predictions and simulation results.

## 3.6 Discussion of Results

It has been observed that the combined effects of critical sections (for all block sizes) and of the spatial locality [70] of accesses (for block sizes larger than one) to shared writable blocks result in access bursts to those blocks by different processors. Based on this observation, we have extended a previous program model for the sharing of data, and we have tried to match the model predictions and the results from simulations of algorithms in multiprocessors with infinite caches.

It appears that iterative algorithms such as the Jacobi or the S.O.R. and the image component labeling algorithm are very well suited to cache-based systems with large data caches, because shared block contention is low (in realistic cases, the number of processors sharing a given writable block is less than four and the fraction of accesses to shared writable block is low). Figures 14, 15, 16, 17, 30 and 31 show that bigger block sizes do not improve the overall hit rate on shared writable blocks and cause more penalty since *false sharing*[5] effects exist when the cache block size is greater than two in the case of $P$=4 and grid size of 128 × 128. When cache block size changes from one to two, since the average miss rate on each S-block access decreases (i.e., $M_i$ decreases) and the number of accesses to such blocks remains the same (i.e., $q_s$ unchanged), the miss rate decreases. In addition, the probability of a coherence event per access to S-blocks decreases, and the event penalty increases. It does not, however, match with the decrease in the probability of the coherence event. This results in the decrease of the total penalty. When the cache block size is greater than two, the bigger the cache block size, the bigger the system miss ratio and total penalty. This occurs because the average miss rate on each S-block access decreases (i.e., $M_i$ decreases) but the number of accesses to such blocks increases (i.e., $q_s$ increases). The probability of a coherence event per access to S-blocks decreases, but this is more than compensated for by the increase of $q_s$ and of the penalty associated with each coherence event. Therefore, for shared data accesses, the block size should be small (one or two data elements). Note that this conclusion is only valid if the caches are large enough to contain all data across successive iterations. The first iteration causes large number of misses for the initial load of data and instructions and a bigger block size helps these transients.

The results of quicksort and bitonic merge sort are the average values of results of multiple random input files. Bigger block sizes do not improve the performance of the quicksort algorithm when cache block size is greater than eight since large cache blocks have higher

[5]False sharing is induced by the collocation of different data items in the same cache block. For an algorithm with no false sharing, the miss rate can never be increased when cache block size increases in infinite cache-based multiprocessor systems.

probability to be accessed by multiple processors at the same time. This is also the effect of false sharing. Because the input file array is a linear array, if there was no false sharing, the miss ratio would be halved when the cache block size is doubled. There is no false sharing in the bitonic merge sort when the cache block size is less than problem size divided by number of processors (that is, $B \leq \frac{N}{P}$). and the miss ratio and system total penalty decrease when the cache block size increases.

Bigger block sizes also improve the performance of the FFT routines since there is no false sharing in the two FFT algorithms when $B \leq \frac{N}{P}$. While the penalties on individual coherence events increase with the block size, the probability of the coherence events decreases and the number of shared block accesses causing these events remains constant, as the block size increases.

As for the iterative algorithms and the dynamic quicksort algorithm, false sharing effect exists in the single source shortest path algorithm. Bigger cache block sizes cannot improve the performance of the algorithm. The false sharing effect becomes more noticeable when the number of processors in the system is greater than eight or the cache block size is greater than 16.

In all the simulations we ran, there is a maximum block size beyond which the performance drops sharply. This block size depends on the size of the problem and on the decomposition of the algorithm (i.e., the number of processors) [36].

Figures 9 to 13 summarize the accuracy of the access burst program model by comparing model predictions and simulation results. It appears that for the eight algorithms studied in this thesis, the precision of the model based on the idea of access bursts in many cases is good. It appears that the models and their parameters are sufficient to approximate the shared block contention effect for some important parallel algorithms, and in the case of the infinite cache model.

If we look at the comparisons between model and simulations, it appears that the non-shuffling FFT results in the worst predictions. In the case of the non-shuffling FFT, the model predicts that the coherence overhead will increase with $P$, the number of processors, while the simulations predict that it remains constant. A closer look at Figure 7 shows that an S-block is not shared by all $log_2 P + 1$ processors at all times but rather that it is shared by different groups of two processors at different stages of the computation. Applying the model with $J$=2 would yield a much improved prediction of the model.

We never expected the models to fit exactly each case: Because of the large data reduction in the stochastic models, a given model with given parameter values maps on different algorithms with different behaviors. In fact, the stochastic model should represent the *average* algorithm among all algorithms with a given set of parameters. In this context, it is probably more relevant to analyze the correlation between model and simulation than to try to match the simulation for a given algorithm with the model. Figures 9 to 13 correlate the model predictions with the simulation results for five performance metrics, in the case of all the simulations that we have run. In these figures, we have plotted the values predicted by the analytical model as a function of the values predicted by the simulation model. There is a high correlation except for $IN\_RW$. In Figure 12, for some algorithms, the simulation results show that the $IN\_RW$ is zero because a shared writable block can only be modified by one processor. The model predictions, however, may reach up to 0.025 because in the

analytical model every processor has the same probability to start the next new burst for the shared writable block. If the values of parameters $W$ and $f$ for the block are not zero, the probability of event $IN\_RW$ from the model prediction is not equal to zero.

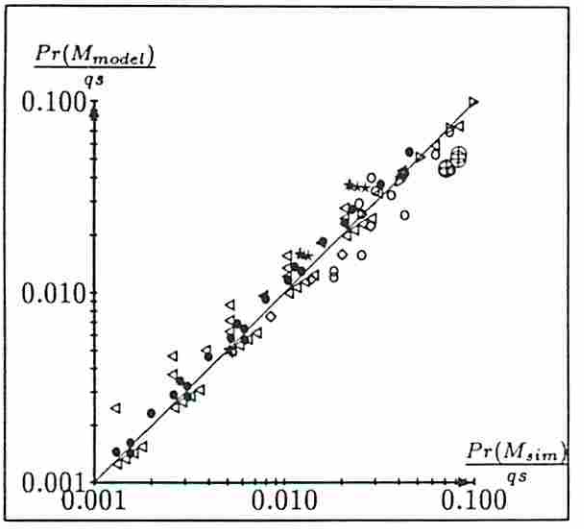



Figure 9: Frequency of occurrence of event $M$ (per data access)

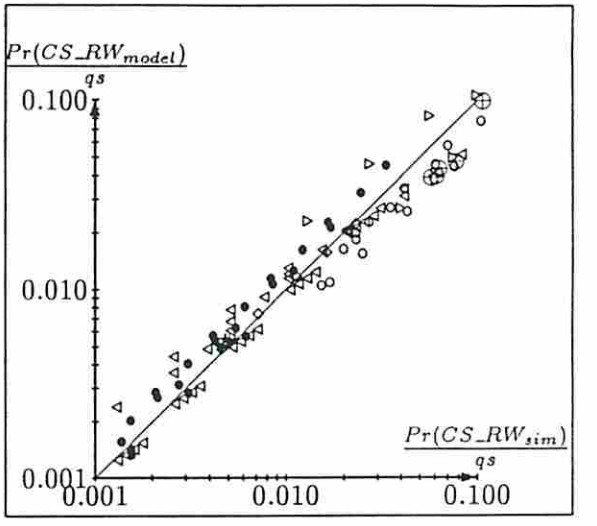



Figure 11: Frequency of occurrence of event $CS\_RW$ (per data access)

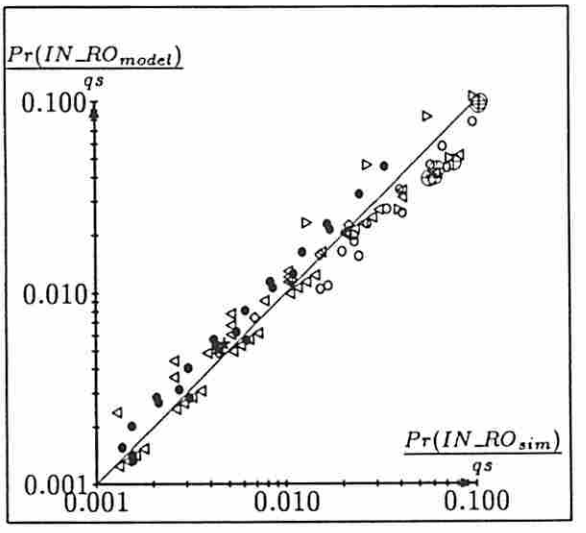



Figure 10: Frequency of occurrence of event $IN\_RO$ (per data access)

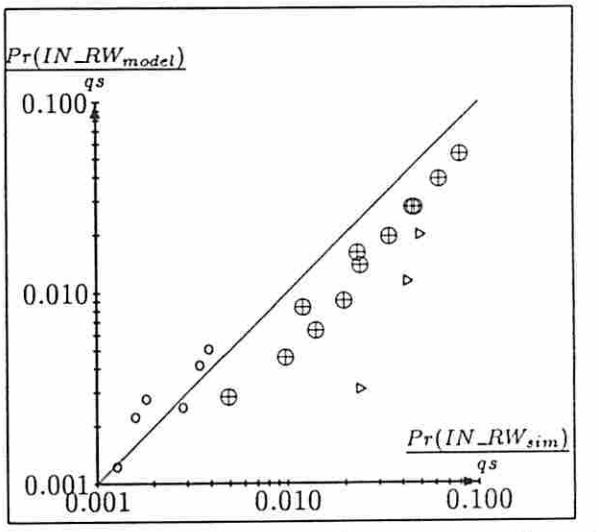



Figure 12: Frequency of occurrence of event $IN\_RW$ (per data access)

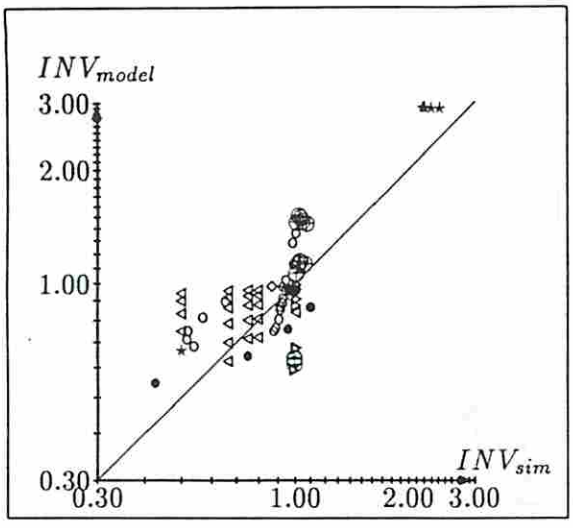



Figure 13: Average number of copies be invalidated (per invalidation)

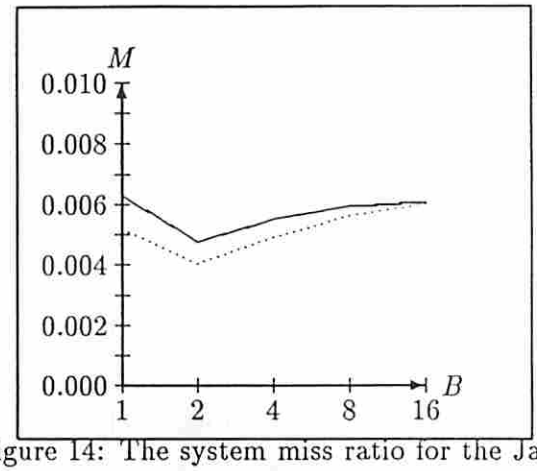
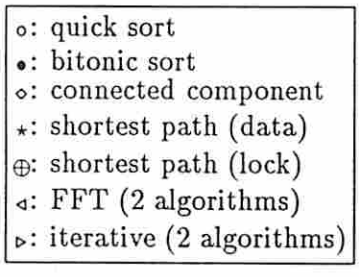



Figure 14: The system miss ratio for the Jacobi iterative algorithm ($P$=4, grid size of 128 × 128) (plain line: simulation results, dotted line: model predictions)



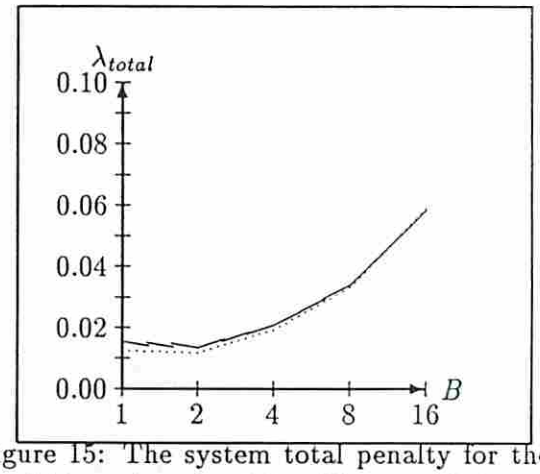




Figure 15: The system total penalty for the Jacobi iterative algorithm ($P$=4, grid size of 128 × 128) (plain line: simulation results, dotted line: model predictions)

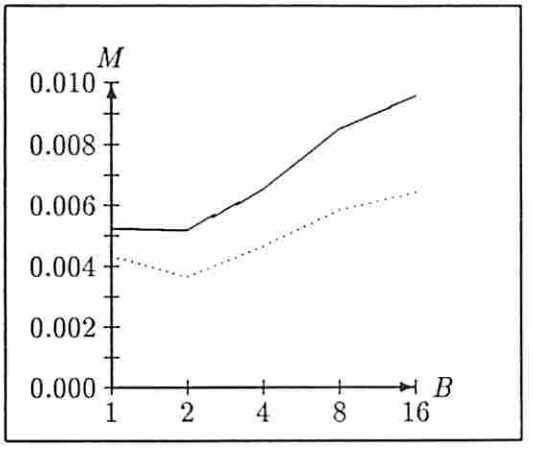



Figure 16: The system miss ratio for the S.O.R. iterative algorithm ($P$=4, grid size of 128 x 128) (plain line: simulation results, dotted line: model predictions)

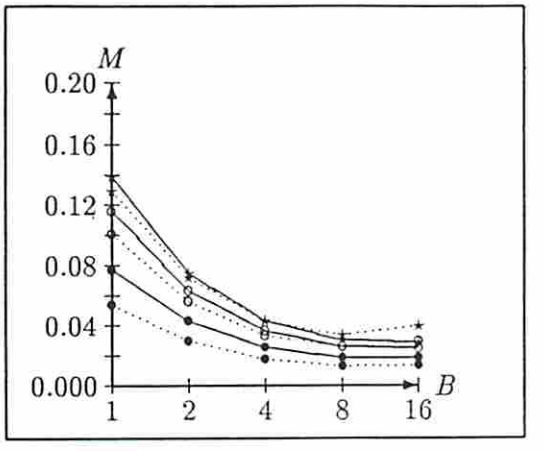



Figure 18: The system miss ratio for the dynamic quicksort ($N$=32768) (plain line: simulation results, dotted line: model predictions) (•: $P$=2, ○: $P$=4, ⋆: $P$=8)

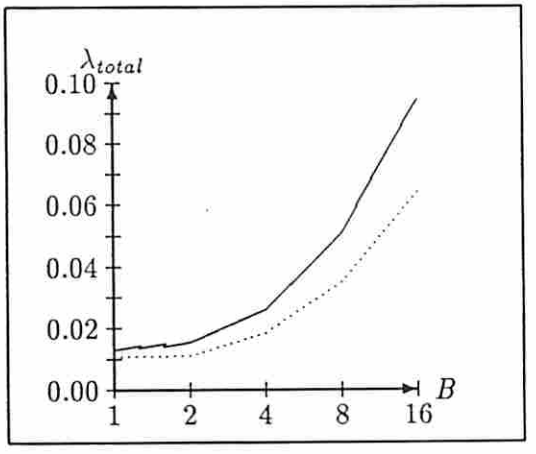



Figure 17: The system total penalty for the S.O.R. iterative algorithm ($P$=4, grid size of 128 x 128) (plain line: simulation results, dotted line: model predictions)

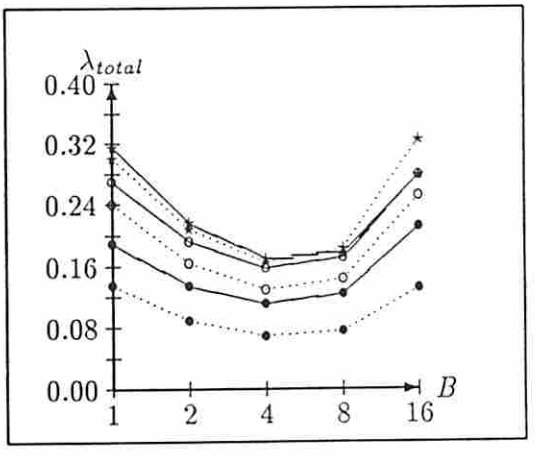



Figure 19: The system total penalty for the dynamic quicksort ($N$=32768) (plain line: simulation results, dotted line: model predictions) (•: $P$=2, ○: $P$=4, ⋆: $P$=8)

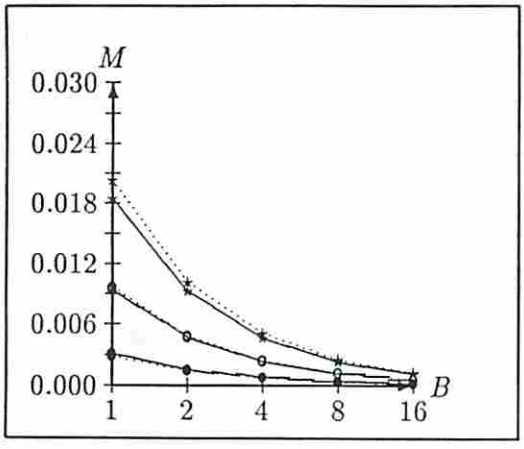



Figure 20: The system miss ratio for the bitonic merge sort ($N$=32768) (plain line: simulation results, dotted line: model predictions) (•: $P$=2, o: $P$=4, ⋆: $P$=8)

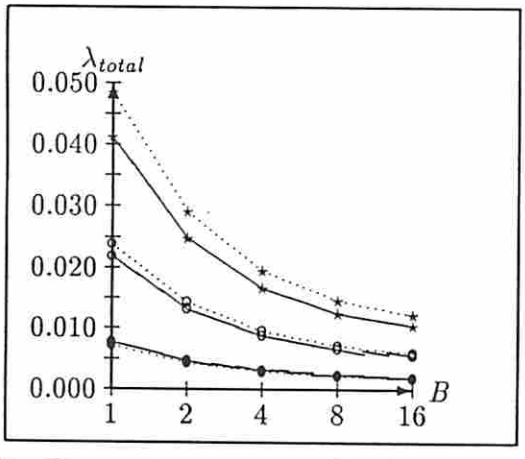



Figure 21: The system total penalty for the bitonic merge sort ($N$=32768) (plain line: simulation results, dotted line: model predictions) (•: $P$=2, o: $P$=4, ⋆: $P$=8)

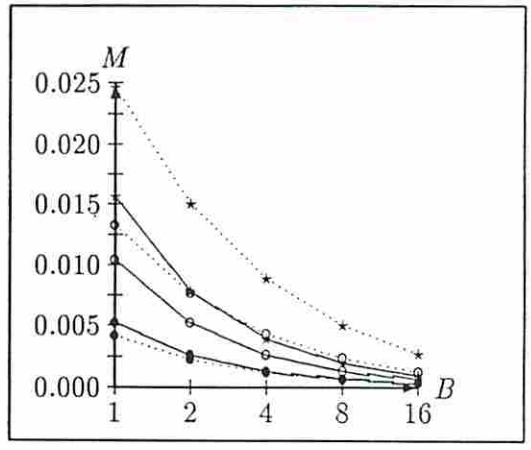



Figure 22: The system miss ratio for the non-shuffling FFT algorithm ($N$=65536) (plain line: simulation results, dotted line: model predictions) (•: $P$=2, o: $P$=4, ⋆: $P$=8)

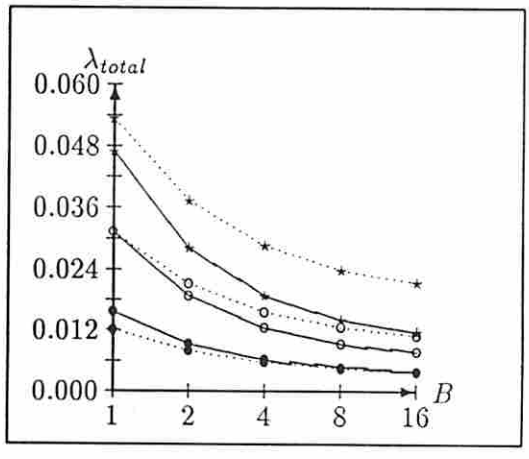



Figure 23: The system total penalty for the non-shuffling FFT algorithm ($N$=65536) (plain line: simulation results, dotted line: model predictions) (•: $P$=2, o: $P$=4, ⋆: $P$=8)

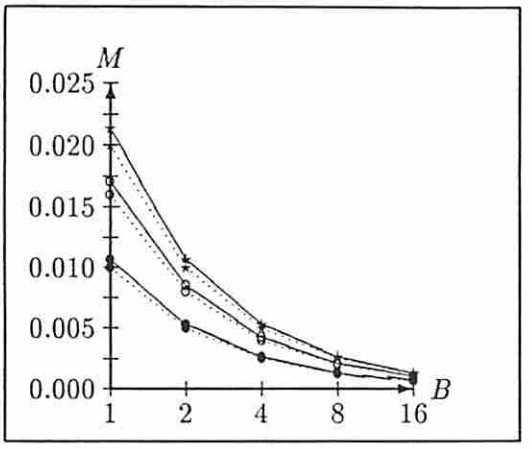



Figure 24: The system miss ratio for the shuffling FFT algorithm ($N$=65536) (plain line: simulation results, dotted line: model predictions) (•: $P$=2, ○: $P$=4, ⋆: $P$=8)

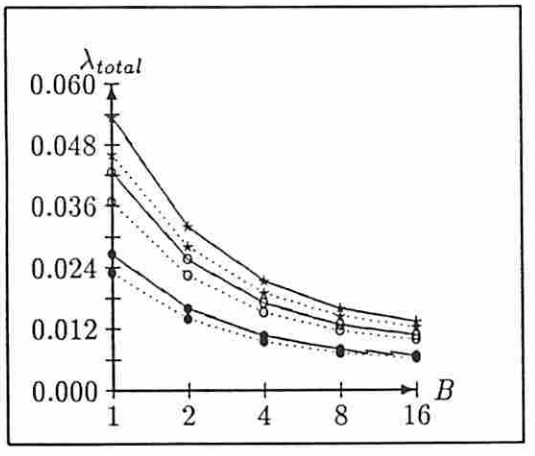



Figure 25: The system total penalty for the shuffling FFT algorithm ($N$=65536) (plain line: simulation results, dotted line: model predictions) (•: $P$=2, ○: $P$=4, ⋆: $P$=8)

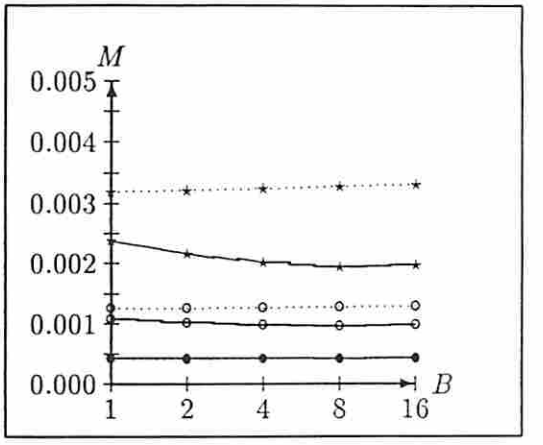



Figure 26: The system miss ratio (data D) for the single source shortest path algorithm (128 nodes) (plain line: simulation results, dotted line: model predictions) (•: $P$=2, ○: $P$=4, ⋆: $P$=8)

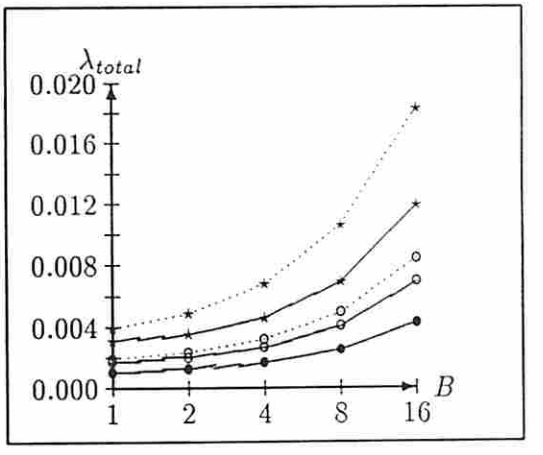



Figure 27: The system total penalty (data D) for the single source shortest path algorithm (128 nodes) (plain line: simulation results, dotted line: model predictions) (•: $P$=2, ○: $P$=4, ⋆: $P$=8)

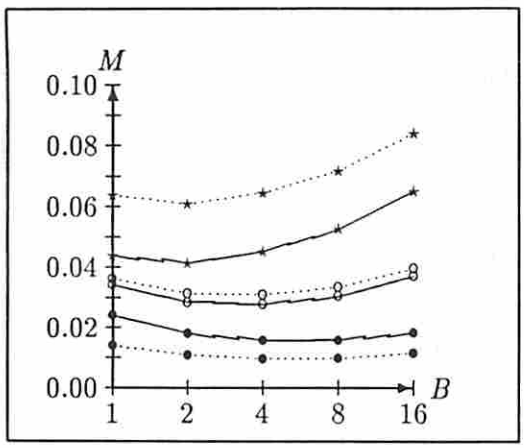



Figure 28: The system miss ratio (lock D) for the single source shortest path algorithm (128 nodes) (plain line: simulation results, dotted line: model predictions) (•: $P$=2, o: $P$=4, ⋆: $P$=8)

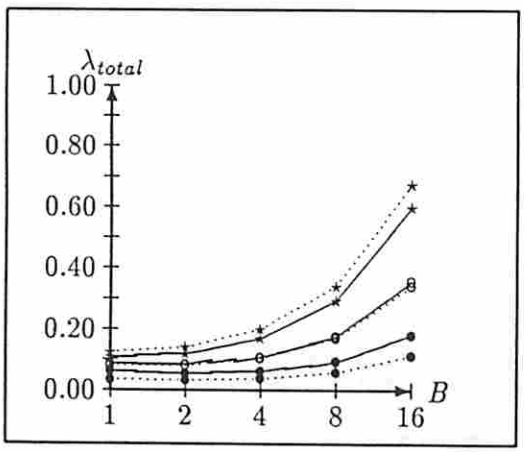



Figure 29: The system total penalty (lock D) for the single source shortest path algorithm (128 nodes) (plain line: simulation results, dotted line: model predictions) (•: $P$=2, o: $P$=4, ⋆: $P$=8)

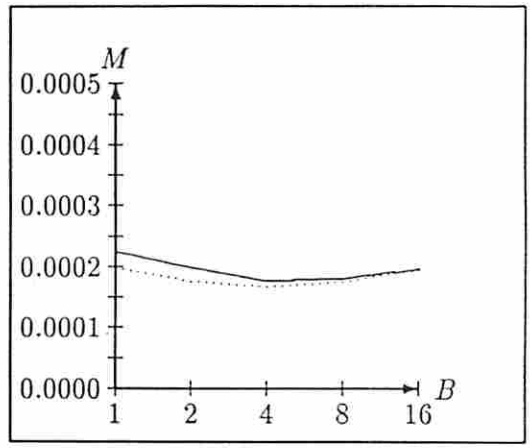



Figure 30: The system miss ratio for the image component labeling algorithm ($P$=4, grid size of 128 × 128) (plain line: simulation results, dotted line: model predictions)

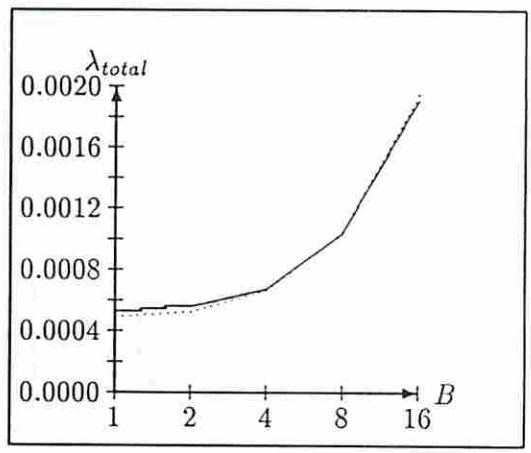



Figure 31: The system total penalty for the image component labeling algorithm ($P$=4, grid size of 128 × 128) (plain line: simulation results, dotted line: model predictions)

# 4 A PERFORMANCE COMPARISON OF CACHE COHERENCE PROTOCOLS BASED ON THE ACCESS BURST MODEL

The choice of a cache coherence protocol is an important design consideration in multiprocessor systems. Several protocols have been proposed and techniques must be developed to compare their relative merits in different computing environments.

In this chapter, we compare five write-invalidate protocols for their effectiveness in handling shared writable blocks. The access burst program model is applied to the different protocols.

Protocols are modeled by Markov chains; an analytical closed-form solution is derived for all components of the cache coherence overhead and for all cache coherence protocols in systems with caches of infinite sizes.

## 4.1 Introduction

In this chapter we apply the access burst model to compare the effectiveness of different coherence protocols in handling shared writable blocks. The access burst model was introduced in Chapter 2, and is based on the observation that shared writable blocks are accessed in critical or semi-critical sections. Caches are assumed to have infinite sizes and models are derived for computations in steady-state. This simplification drastically reduces the number of parameters in the models. The results obtained are an indication of protocol efficiency for very large caches and compute-intensive, iterative algorithms. Many of these algorithms exist for asynchronous multiprocessors [12]. An example of such an algorithm is given at the end of the chapter.

To study write-broadcast protocols in steady state and in a system with infinite caches is trivial since the only coherence events are Writes to shared writable blocks. Therefore, the coherence overhead can be measured by the probability of a Write to any shared writable block multiplied by the average time that a processor is blocked at the occurrence of the event. Hence, in this chapter, we concentrate on write-invalidate protocols only.

The remainder of this chapter is organized as follows. In Section 4.2, the access burst model is applied to the analysis of five write-invalidate protocols. In Section 4.3, we compare the predictions of the model with execution-driven simulations on the Jacobi iterative algorithm. Finally, the efficiencies of the protocols are compared to the model in Section 4.4.

## 4.2 Cache Coherence Protocols

In this section, five different cache coherence protocols are described and analyzed. A closed-form formula for the overhead of each coherence event is derived based on the access burst program model.

### 4.2.1 The Basic Coherence Protocol

This coherence protocol was described in detail in Chapter 2. A block may exist in one of three states in a cache: INVALID (no copy of the block in the cache), RO (Read-Only; an arbitrary number of caches can have this block, and all the copies are identical), and RW (Read-Write; the block has been locally modified since it was brought into the cache and the shared memory copy is stale).

Protocol Description

The Basic coherence protocol works in steady state as follows:

1. *Read hit*: The block may be accessed locally without delay.
2. *Read miss*: If a remote cache has an RW copy of the block, the modified block must first be written back to shared memory, and then shared memory supplies the block to the requesting cache. Otherwise, the block comes directly from shared memory. Each cache with a copy of the block sets the state of its copy to RO.
3. *Write hit*: If the copy of the block is in state RO an invalidation signal must be sent to all other caches. The state of the local copy is changed to RW.
4. *Write miss*: A Write miss is treated like a *Read miss* with the following difference: if copies exist in other caches, they are invalidated and the state of the local copy is set to RW.

Coherence Analysis

The detailed coherence analysis for the Basic cache coherence protocol has been described in Chapter 2. In this protocol, four coherence events exist, which are miss ($M$), transition from RO to RW ($IN\_RO$), transition fro RW to RO ($CS\_RW$), and transition fro RW to RW ($IN\_RW$). An average penalty, $\lambda$, is associated with each of these events. The penalty associated with an event is defined as the average time a processor is blocked at each occurrence of the event. Let us define $t_{mc}$ and $t_{inv}$ as the times taken by the transfer of a cache block between shared memory and a cache and by the invalidation of a block in a different cache, respectively. Thus, $\lambda_M$ is equal to $t_{mc}$, $\lambda_{IN\_RO}$ is equal to $t_{inv}$, $\lambda_{CS\_RW}$ is equal to $t_{mc}$, and $\lambda_{IN\_RW}$ is equal to $t_{mc}$.

Closed-Form Derivation

The protocol is modeled as a discrete Markov diagram shown in Figure ??. An analytical closed-form solution is derived for the probabilities of the four coherence events and are listed as follows:

$$P(M) = \frac{1}{l} \cdot \frac{(J-1) \cdot W}{1 + (J-1) \cdot W},$$

$$P(IN\_RO) = \frac{1}{l} \cdot \frac{(J-1) \cdot W \cdot (1 - W \cdot f)}{J - 1 + W},$$

$$P(CS\_RW) = \frac{1}{l} \cdot \frac{(J-1) \cdot W \cdot (1 - W \cdot f)}{J - 1 + W},$$

and:

$$P(IN\_RW) = \frac{1}{l} \cdot \frac{(J-1) \cdot W^2 \cdot f}{J - 1 + W}.$$

The total penalty is

$$\begin{aligned}
\lambda_{total} &= P(M) \cdot \lambda_M + P(IN_RO) \cdot \lambda_{IN_RO} \\
&\quad + P(CS_RW) \cdot \lambda_{CS_RW} + P(IN_RW) \cdot \lambda_{IN_RW} \\
&= \frac{1}{l} \cdot \left\{ \frac{J \cdot (J-1) \cdot W \cdot (1+W)}{(1 + (J-1) \cdot W) \cdot (J - 1 + W)} \cdot t_{mc} \right. \\
&\quad \left. + \frac{(J-1) \cdot W \cdot (1 - W \cdot f)}{J - 1 + W} \cdot t_{inv} \right\}.
\end{aligned}$$

### 4.2.2 The Write-Once Coherence Protocol

In the Write-Once protocol [44, 87], a block in a cache can be in one of four states: INVALID, VALID (as RO in the Basic protocol), RESERVED (a datum in the block has been locally modified exactly once since it was brought into the cache and shared memory is updated), and DIRTY (data in the block has been locally modified more than once since it was brought into the cache and the shared memory copy is stale).

Protocol Description

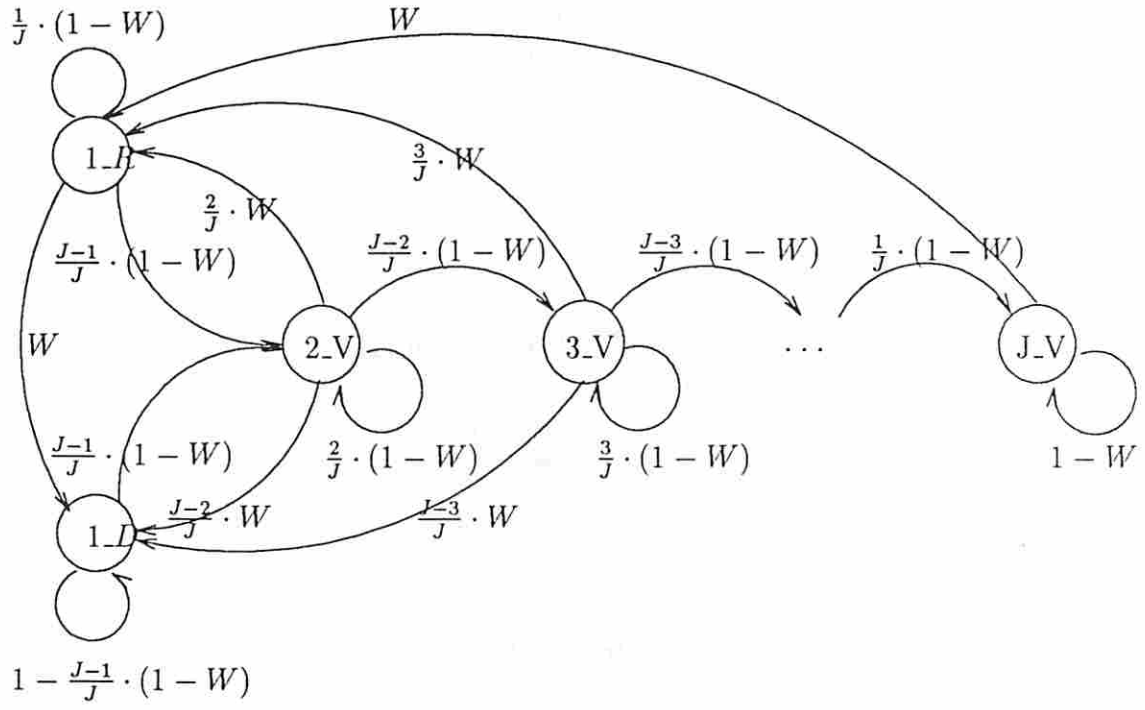
The Write-Once coherence protocol works in steady state as follows:

1. *Read hit*: The block may be accessed locally without delay.

2. *Read miss*: If a remote cache has the copy of the block in state DIRTY, the remote cache supplies the block to the requester and updates shared memory at the same time. Otherwise, the block is loaded from shared memory. All caches having a copy of the block set its state to VALID.

3. *Write hit*: If the block is already in state DIRTY or RESERVED, the Write can be processed locally without delay and the state of the block is always set to DIRTY. If the block is in state VALID, the word being modified is written through to shared memory, block copies in other caches are invalidated and the state of the block is set to RESERVED.

4. *Write miss*: If one remote cache owns a copy of the block in state DIRTY, the block is loaded from the remote cache and the remote cache invalidates its own copy; otherwise, the block is loaded from shared memory. Upon detecting the write miss signal on shared bus, all caches with the copy of the block invalidate their copies at the same time. Once the block is loaded, the Write takes place and the state of the block is always set to DIRTY.




1_$D$ : Dirty copy
1_$R$ : Reserved copy
$k$_V : $k$ processors own a valid copy

Figure 32: Markov chain for the Write-Once coherence protocol

We denote the state of a block in the system by 1_$R$, 1_$D$, 2_$V$,..., $J$_$V$, where 1_$R$ or 1_$D$ mean that the block is owned by one cache and is a RESERVED or DIRTY copy, respectively; $k$_$V$ means that there are VALID copies of the block in $k$ caches. The discrete Markov chain for the Write-Once coherence protocol is shown in Figure 32. This discrete Markov diagram is the same as the one shown in Figure ??(b) with the following differences. The state 1_$RW$ in Figure ??(b) is split into two states, 1_$R$ and 1_$D$; at the end of each Write burst, the next state is 1_$D$ if a miss occurs; otherwise, the next state is 1_$R$.

Coherence Analysis

Three possible cache coherence events can occur:

1. **Miss:** This event is like the $M$ event in the Basic cache coherence protocol except that the block is supplied by a remote cache rather than shared memory if the remote cache has a DIRTY copy of the block. Hence, some misses cause cache-to-cache transfers (these miss events are denoted $M$_$cc$), and some misses cause memory-to-cache transfers (these misses are denoted $M$_$mc$.) $P(M\_cc)$ is equal to $[P_{1\_D} \cdot (J-1)/J]/l$; $P(M\_mc)$ is equal to $[(P_{1\_R} \cdot (J-1)/J) + \sum_{j=2}^{J-1} P_{j\_V} \cdot (J-j)/J]/l$;

2. **Transition from VALID to RESERVED:** this event, denoted $CS\_V\_R$ (Change State from Valid to Reserved), occurs either at the end of a Write burst when no miss event happens in the burst, or occurs in a transition from 1_$R$ to 1_$R$, provided the

second access burst is executed by a different processor and starts with a Read. The modified *word* is written through to shared memory. Thus, $P(CS\_V\_R)$ is equal to $[W \cdot (1-f) \cdot P_{1\_R} \cdot (J-1)/J + \sum_{j=2}^{J} W \cdot P_{j\_V} \cdot j/J]/l$.

3. **Transition from DIRTY to VALID**: this event, denoted $CS\_D$ (Change State of a DIRTY copy), is very much like the $CS\_RW$ event in the Basic cache coherence protocol except that the cache having the DIRTY copy of the block supplies the block to the requesting cache and at the same time updates shared memory. $P(CS\_D)$ is equal to $[P_{1\_D} \cdot (1-W) \cdot (J-1)/J + P_{1\_D} \cdot W \cdot (1-f) \cdot (J-1)/J]/l$. When the time to update shared memory is longer than the time of a cache-to-cache transfer, an extra penalty must be added to the miss penalty for the $CS\_D$ event. On the other hand, in systems where the latency of updating shared memory is less than that of the cache-to-cache transfer, no extra penalty is needed to account for memory update. At the end of the $M$ event, shared memory has already been updated.

In addition to the $t_{mc}$ and $t_{inv}$ defined previously, we define two new terms, $t_{word}$ and $t_{cc}$, which are the times to write a word to shared memory and to transfer a block between two caches, respectively. Hence, $\lambda_{M\_mc}$ is equal to $t_{mc}$. $\lambda_{M\_cc}$ is equal to $t_{cc}$. $\lambda_{CS\_V\_R}$ which is equal to $t_{word}$. $\lambda_{CS\_D}$ is equal to $t_{diff}$ where $t_{diff}=(t_{mc} - t_{cc})$, if $t_{mc} > t_{cc}$, or $t_{diff}=0$, otherwise.

Closed-Form Derivation

From Figure 32, we can equate the state probability equations:

$$\frac{J-1}{J} \cdot (1-W) \cdot P_{1\_D} = W \cdot P_{1\_R} + \sum_{j=2}^{J} \frac{J-j}{J} \cdot W \cdot P_{j\_V},$$

$$(W + \frac{J-1}{J} \cdot (1-W)) \cdot P_{1\_R} = \sum_{j=2}^{J} \frac{j}{J} \cdot W \cdot P_{j\_V},$$

$$(W + \frac{J-2}{J} \cdot (1-W)) \cdot P_{2\_V} = \frac{J-1}{J} \cdot (1-W) \cdot (P_{1\_D} + P_{1\_R}),$$

and:

$$(W + \frac{J-k}{J} \cdot (1-W)) \cdot P_{k\_V} = \frac{J-(k-1)}{J} \cdot (1-W), \cdot P_{(k-1)\_V}, \quad for\ 3 \leq k \leq J.$$

After transformations similar to those in Appendix A, we have:

$$P_{1\_D} = \frac{J \cdot W^2 \cdot (J^2 + 2JW - 2J - 2W + 2)}{(J-1+W)^2 \cdot (1+(J-1)W)},$$

and:

$$P_{1\_R} = \frac{J \cdot W \cdot (J - J \cdot W^2 + W^2 - 1)}{(J-1+W)^2 \cdot (1+(J-1)W)}.$$

Therefore, the probability of each event can be written as follows:

$$P(M\_cc) = \frac{1}{l} \cdot \frac{(J-1) \cdot W^2 \cdot (J^2 + 2JW - 2J - 2W + 2)}{(J-1+W)^2 \cdot (1+(J-1)W)},$$

$$P(M\_mm) = \frac{1}{l} \cdot \frac{(J-1) \cdot W \cdot (1-W) \cdot (J^2 + 2JW - 2J - 3W + 1)}{(J-1+W)^2 \cdot (1+(J-1)W)},$$

$$P(CS\_V\_R) = \frac{1}{l} \cdot \left\{ \frac{(J-1) \cdot W \cdot (1-W^2)}{(J-1+W) \cdot (1+(J-1)W)} + \frac{(J-1) \cdot W^2 \cdot (1-f)}{J-1+W} \right\},$$

and

$$P(CS\_D) = \frac{1}{l} \cdot \frac{(J-1) \cdot W^2 \cdot (1-fW) \cdot (J^2 + 2JW - 2J - 2W + 2)}{(J-1+W)^2 \cdot (1+(J-1)W)}.$$

The total penalty is:

$$\begin{aligned}
\lambda_{total} &= P(M_cc) \cdot \lambda_{M_cc} + P(M_mc) \cdot \lambda_{M_mc} \\
&\quad + P(CS_V_R) \cdot \lambda_{CS_V_R} + P(CS_D) \cdot \lambda_{CS_D} \\
&= \frac{1}{l} \cdot \left\{ \frac{(J-1) \cdot W^2 \cdot (J^2 + 2JW - 2J - 2W + 2)}{(J-1+W)^2 \cdot (1+(J-1)W)} \cdot t_{cc} \right. \\
&\quad + \frac{(J-1) \cdot W \cdot (1-W) \cdot (J^2 + 2JW - 2J - 3W + 1)}{(J-1+W)^2 \cdot (1+(J-1)W)} \cdot t_{mc} \\
&\quad + \left[ \frac{(J-1) \cdot W \cdot (1-W^2)}{(J-1+W) \cdot (1+(J-1)W)} + \frac{(J-1) \cdot W^2 \cdot (1-f)}{J-1+W} \right] \cdot t_{word} \\
&\quad \left. + \frac{(J-1) \cdot W^2 \cdot (1-fW) \cdot (J^2 + 2JW - 2J - 2W + 2)}{(J-1+W)^2 \cdot (1+(J-1)W)} \cdot t_{diff} \right\}.
\end{aligned}$$

### 4.2.3 The Synapse Coherence Protocol

In the Synapse protocol [42], there is a single-bit tag with each cache block in shared memory, indicating whether shared memory is to respond to a miss on that block. If a remote cache has a modified copy of the block, the bit inhibit shared memory from supplying the block. This bit can prevent a possible race condition when the remote cache does not respond quickly enough to inhibit shared memory.

A cache block may be in one of the three states: INVALID, VALID (as RO in the Basic protocol), and DIRTY (as RW in the Basic protocol).

Protocol Description

The Synapse coherence protocol works in steady state as follows:

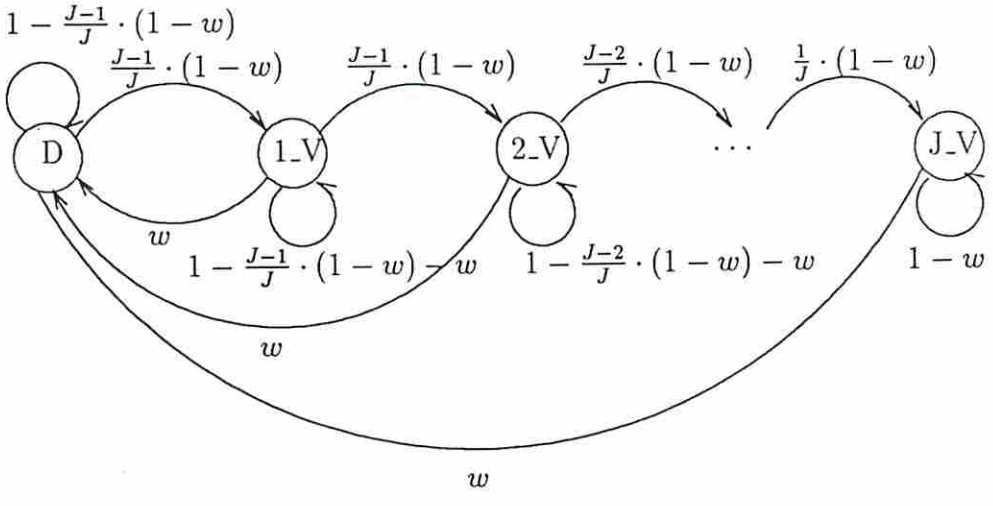
1. *Read hit*: The access may be processed locally without delay.

2. *Read miss*: If a remote cache has a DIRTY copy of the block, the modified block must first be written back to shared memory; the tag bit of the block in shared memory is set; the remote cache invalidates its local copy and sends a busy acknowledge signal to the requesting cache. When the requesting cache receives this busy signal, it must send an additional read miss request in order to get the copy of the block from shared memory. In all other cases, the block is directly supplied by shared memory. The state of the loaded block is always set to VALID.

3. *Write hit*: If the block is in state DIRTY in local cache, the Write can be processed locally without delay. If the local copy of the block is in state VALID, the procedure is as follows: shared memory has to transfer the ownership along with the copy to the requesting cache and each cache with a copy of the block observes this bus transaction and at the same time invalidates its copy of the block.

4. *Write miss*: If a remote cache has a DIRTY copy of the block, the remote cache transfers the ownership along with the block copy to the requester. If all copies of the block in the system are VALID, shared memory supplies the copy to the requesting cache and each cache which has a VALID block copy invalidates its copy at the same time. The tag bit in shared memory is reset.

We can denote the states of a block in the system by 1_$D$, 1_$V$, 2_$V$,..., $J$_$V$, where 1_$D$ is the state in which the block is owned by one cache and is a DIRTY copy; $k$_$V$ means that there are VALID copies of the block in $k$ caches. If we observe state transitions at the end of each access burst, the discrete Markov chain of the Synapse coherence protocol can be drawn as shown in Figure 33. This discrete Markov diagram is the same as the one shown in Figure ??(b), with one exception: one additional state, 1_$V$, is introduced.




D : Dirty copy

i_V : i processors own the valid copy

Figure 33: Markov chain for the Synapse coherence protocol

Coherence Analysis

Three possible cache coherence events can occur:

1. **Miss:** There are two types of miss events (as in the Write-Once protocol). These events are denoted $M\_cc$ and $M\_mc$ for the cases of cache-to-cache and memory-to-cache transfers respectively. A miss causes a cache-to-cache transfer when a remote cache has a DIRTY copy of the block and the access burst is a Write burst. Therefore, $P(M\_cc)$ is equal to $[W \cdot P_{1\_D} \cdot (J-1)/J]/l$; $P(M\_mc)$ is equal to $[((1-W) \cdot P_{1\_D} \cdot (J-1)/J) + \sum_{j=1}^{J-1} P_j \cdot (J-j)/J]/l$;

2. **Transition from VALID to DIRTY on hit:** this event, denoted $IN\_V\_h$ (INvalidation of Valid Copy(ies) on hit), occurs either at the end of a Write burst when no miss event happened in the burst, or in a transition from $1\_D$ to $1\_D$, provided the second access burst is executed by a different processor and starts with a Read. This event includes a block transfer from shared memory to the requesting cache. Thus, $P(IN\_V\_h)$ is equal to $[W \cdot (1-f) \cdot P_{1\_D} \cdot (J-1)/J + \sum_{j=1}^{J} W \cdot P_{j\_V} \cdot j/J]/l$;

3. **Transition from DIRTY to VALID:** this event, denoted $CS\_D$, is the same as the $CS\_RW$ event in the Basic cache coherence protocol. $P(CS\_D)$ is equal to $[P_{1\_D} \cdot (1-W) \cdot (J-1)/J + P_{1\_D} \cdot W \cdot (1-f) \cdot (J-1)/J]/l$.

The penalty of each event is: $\lambda_{M\_cc}$ is equal to $t_{cc}$; $\lambda_{M\_mc}$ is equal to $t_{mc}$; $\lambda_{IN\_V\_h}$ is equal to $t_{mc}$; and $\lambda_{CS\_D}$ is equal to $t_{mc}$.

Closed-Form Derivation

From Figure 33, we can derive the state probabilities from the following equations:

$$\frac{J-1}{J} \cdot (1-W) \cdot P_{1\_D} = \sum_{j=1}^{J} W \cdot P_{j\_V},$$

$$(W + \frac{J-1}{J} \cdot (1-W)) \cdot P_{1\_V} = \frac{J-1}{J} \cdot (1-W) \cdot P_{1\_D},$$

and:

$$(W + \frac{J-k}{J} \cdot (1-W)) \cdot P_{k\_V} = \frac{J-(k-1)}{J} \cdot (1-W) \cdot P_{(k-1)\_V}, \quad for\ 2 \leq k \leq J.$$

After transformations similar to those in Appendix A, we have:

$$P_{1\_D} = \frac{J \cdot W}{J-1+W},$$

$$P(M\_cc) = \frac{1}{l} \cdot \frac{(J-1) \cdot W^2}{J-1+W},$$

and:

$$P(M\_mc) = \frac{1}{l} \cdot \frac{(J-1) \cdot W \cdot (1-W) \cdot (J + JW - W)}{(J-1+W) \cdot (1+(J-1)W)}.$$

The probability of other events can be written as follows:

$$P(IN\_V\_h) = \frac{1}{l} \cdot \frac{(J-1) \cdot W \cdot (1 + JW^2 - W^2 - fW \cdot (1+(J-1)W))}{(J-1+W) \cdot (1+(J-1)W)},$$

and:

$$P(CS\_D) = \frac{1}{l} \cdot \frac{(J-1) \cdot W \cdot (1-fW)}{J-1+W}.$$

The total penalty for the Synapse protocol is:

$$\begin{aligned}
\lambda_{total} &= P(M_cc) \cdot \lambda_{M_cc} + P(M_mc) \cdot \lambda_{M_mc} \\
&\quad + P(IN_V_h) \cdot \lambda_{IN_V_h} + P(CS_D) \cdot \lambda_{CS_D} \\
&= \frac{1}{l} \left\{ \frac{(J-1) \cdot W^2}{J-1+W} \cdot t_{cc} + \frac{(J-1) \cdot W \cdot (JW - 2W + J + 2)}{(J-1+W) \cdot (1+(J-1)W)} \cdot t_{mc} \right. \\
&\quad \left. - \frac{2 \cdot (J-1) \cdot W^2 \cdot f}{J-1+W} \cdot t_{mc} \right\}.
\end{aligned}$$

### 4.2.4 The Illinois Coherence Protocol

In the Illinois protocol [64], a block in a cache can be in one of four states: INVALID, EXCL-UNMOD (Exclusive-Unmodified; no other cache has this block; data in block is consistent with shared memory), SHARED-UNMOD (Shared-Unmodified; as RO in the Basic protocol) and EXCL-MOD (Exclusive-Modified; as RW in the Basic protocol).

Protocol Description

The scheme works in steady state as follows:

1. *Read hit*: The access may be processed locally without delay.

2. *Read miss*: If a remote cache has an EXCL-MOD copy of the block, the remote cache sends the copy to the requesting cache and at the same time updates shared memory. Otherwise, any one cache supplies the copy to the requester. Both caches set their copy to SHARED-UNMOD.

3. *Write hit*: If the local copy of the block is in state EXCL-MOD, it can be updated without delay. Otherwise, the Write cannot be processed until an invalidation signal is sent. The copy in the local cache is set to EXCL-MOD.

4. *Write miss*: A write miss request is broadcast to all caches. Each cache with the copy of the block invalidates its copy. The block is always loaded from a remote cache and its state is set to EXCL-MOD.

We can denote the state of a block in the system by $1\_E$, $2\_S$,..., $J\_S$, where $1\_E$ means that the block is owned by one cache and is an EXCL-MOD copy; $k\_S$ means that there are SHARED-UNMOD copies of the block in $k$ caches. The discrete Markov chain of the Illinois coherence protocol is the same as the one shown in Figure ??(b) provided that the state names are changed.

Coherence Analysis

Three possible cache coherence events can occur:

1. **Miss:** This event is like the $M$ event in the Basic cache coherence protocol except that the block is always supplied by a cache. $P(M)$ is equal to $[(P_{1\_E} \cdot (J-1)/J) + \sum_{j=2}^{J-1} P_{j\_S} \cdot (J-j)/J]/l$.

2. **Transition from SHARED-UNMOD to EXCL-MOD on hit:** This event, denoted $IN\_S\_h$ (INvalidation of SHARED-UNMOD Copy(ies) on hit), and the $IN\_V\_h$ event in the Synapse cache coherence protocol are alike except that the coherence overhead of this event is to broadcast an invalidation signal. $P(IN\_S\_h)$ is equal to $[W \cdot (1-f) \cdot P_{1\_E} \cdot (J-1)/J + \sum_{j=2}^{J} W \cdot P_{j\_S} \cdot j/J]/l$.

3. **Transition from EXCL-MOD to SHARED-UNMOD:** This event, denoted $CS\_E$ (Change State of an EXCL-MOD copy), is the same as the $CS\_D$ event in the Write-Once cache coherence protocol. $P(CS\_E)$ is equal to $[P_{1\_E} \cdot (1-W) \cdot (J-1)/J + P_{1\_E} \cdot W \cdot (1-f) \cdot (J-1)/J]/l$.

The penalty of each event is: $\lambda_M$ is equal to $t_{cc}$, $\lambda_{IN\_S\_h}$ is equal to $t_{inv}$, and $\lambda_{CS\_E}$ is equal to $t_{diff}$ where $t_{diff}$=$(t_{mc} - t_{cc})$, if $t_{mc} > t_{cc}$, or $t_{diff}$=0, otherwise.

Closed-Form Derivation

The state probability equations of the Illinois cache coherence protocol are the same as those of the Basic cache coherence protocol. The probability of each event can be written as follows:

$$P(M) = \frac{1}{l} \cdot \frac{(J-1) \cdot W}{1 + (J-1) \cdot W},$$

$$P(IN\_S\_h) = \frac{1}{l} \cdot \left\{ \frac{(J-1) \cdot W \cdot (1-W^2)}{(J-1+W) \cdot (1+(J-1)W)} + \frac{(J-1) \cdot W^2 \cdot (1-f)}{J-1+W} \right\},$$

and:

$$P(CS\_E) = \frac{1}{l} \cdot \frac{(J-1) \cdot W \cdot (1-W \cdot f)}{J-1+W}.$$

And the total penalty for the Illinois protocol is:

$$\begin{aligned} \lambda_{total} &= P(M) \cdot \lambda_M + P(IN\_S\_h) \cdot \lambda_{IN\_S\_h} + P(CS\_E) \cdot \lambda_{CS\_E} \\ &= \frac{1}{l} \cdot \left\{ \frac{(J-1) \cdot W}{1 + (J-1) \cdot W} \cdot t_{cc} + \frac{(J-1) \cdot W \cdot (1-W \cdot f)}{J-1+W} \cdot t_{diff} \right. \\ &\quad \left. + \left[ \frac{(J-1) \cdot W \cdot (1-W^2)}{(J-1+W) \cdot (1+(J-1)W)} + \frac{(J-1) \cdot W^2 \cdot (1-f)}{J-1+W} \right] \cdot t_{inv} \right\}. \end{aligned}$$

### 4.2.5 The Berkeley Coherence Protocol

In the Berkeley protocol [54], a block in a cache can be in one of the following four states: INV (INValid; as INVALID in the Basic protocol), UNO (UNOwned; as RO in the Basic protocol), EXC (owned EXClusively; the block copy is unique, and therefore it can be updated locally without delay; the cache must respond to any request on the bus for a copy of the block; this state is equivalent to the RW in the Basic protocol), or NON (Owned NON-exclusively; the block copy is owned, but it cannot be modified without informing the other caches). At any time up to one NON copy and several UNO copies of a block can exist. In steady state, there is one and only one NON copy of a block in the system if there exist some UNO copies of the block. On the other hand, there is never a NON copy of the block in the system if there is an EXC copy of the block. The cache, which has a copy of the block in state NON or EXC, is called the owner of the block. If a block is not owned by any cache, shared memory is the owner. In a system with infinite caches in which replacements never occur, the memory cannot be an owner in steady state.

Protocol Description

The Berkeley protocol works as follows in steady state, for the case of infinite caches.

1. *Read hit*: The access is processed locally without delay.

2. *Read miss*: The block is always loaded from another cache and its local state is set to UNO.

3. *Write hit*: If the local copy of the block is in state EXC, the Write is processed without delay. Otherwise, all copies must be invalidated before the Write can be processed; the cache sets its copy to state NON.

4. *Write miss*: The block always comes from another cache and each cache with the copy of the block invalidates its copy. The requesting cache sets its copy to state EXC.

We can denote the state of a block in the system by $1\_E$, $2\_N$,..., $J\_N$, where $1\_E$ means that the block is owned by one cache and is an EXC copy; $k\_N$ means that there are one NON and $(k-1)$ UNO copies of the block in $k$ caches. Provided the state names are changed, the Markov chain is the same as the one shown in Figure ??(b).

Coherence Analysis

In this scheme, two possible cache coherence events can occur:

1. **Miss:** The fraction of misses in this protocol is given by the same expression as in the Illinois protocol, that is, $P(M) = [(P_{1\_E} \cdot (J-1)/J) + \sum_{j=2}^{J-1} P_{j\_N} \cdot (J-j)/J]/l$.

2. **Transition from UNO to NON on hit:** The fraction of references causing this event $IN\_U\_h$ (INvalidation of UNO Copy(ies) on hit), is given by the same expression as for the event $IN\_S\_h$ in the Illinois protocol, that is, $P(IN\_U\_h) = [W \cdot (1-f) \cdot P_{1\_E} \cdot (J-1)/J + \sum_{j=2}^{J} W \cdot P_{j\_N} \cdot j/J]/l$

The penalty of each event is: $\lambda_M = t_{cc}$, and $\lambda_{IN\_U\_h} = t_{inv}$.

Closed-Form Derivation

The state probability equations of the Berkeley cache coherence protocol is the same as those of the Basic cache coherence protocol. The probability of each event can be written as follows:

$$P(M) = \frac{1}{l} \cdot \frac{(J-1) \cdot W}{1 + (J-1)W},$$

and:

$$P(IN\_U\_h) = \frac{1}{l} \cdot \left\{ \frac{(J-1) \cdot W \cdot (1 - W^2)}{(J-1+W) \cdot (1 + (J-1)W)} + \frac{(J-1) \cdot W^2 \cdot (1-f)}{J-1+W} \right\}.$$

And the total penalty is:

$$\begin{aligned} \lambda_{total} &= P(M) \cdot \lambda_M + P(IN\_U\_h) \cdot \lambda_{IN\_U\_h} \\ &= \frac{1}{l} \cdot \left\{ \frac{(J-1) \cdot W}{1 + (J-1)W} \cdot t_{cc} + \frac{(J-1) \cdot W \cdot (1 - W^2)}{(J-1+W) \cdot (1 + (J-1)W)} \cdot t_{inv} \right. \\ &\quad \left. + \frac{(J-1) \cdot W^2 \cdot (1-f)}{J-1+W} \cdot t_{inv} \right\}. \end{aligned}$$

## 4.3 Multitasked Jacobi Iterative Algorithm

In this section, we apply the model to one particular multitasked algorithm, the Jacobi iterative algorithm, to solve Laplace's equation $\nabla^2 x = 0$ on a rectangular domain of $R^2$. The Jacobi iterative algorithm is an iterative, compute-intensive algorithm. The infinite cache condition is met when the data cache of each processor is large enough to contain all the grid elements accessed by the processor. In this case, steady-state is reached after the first iteration. This important algorithm is therefore a good benchmark to apply the model. The details of the algorithm can be found in Section ??. In this algorithm, we have identified eight sets of shared writable blocks. The values of the parameters for each set is given in Table ?? for a grid size of 128 $\times$ 128.

In the computation of the total penalty, we examine two different systems. In system 1, the cache-to-cache transfer time is taken as eight time units: one time unit for bus arbitration, one time unit for address transfer, four time units for a block access and transfer, and two time units for acknowledgement. The memory-to-cache transfer time is taken as ten time units because an access to the memory takes six time units. The time to write a word to shared memory is seven time units: one time unit for bus arbitration, one time unit for address transfer, three time units for a word transfer and memory access, and two time units for acknowledgement. An invalidation signal only takes two time units: one for bus arbitration and one for signal broadcasting. The difference between system 1 and system 2 is the cache-to-cache transfer time. In system 2, the time to retrieve a block in a remote cache is eight time units so that the total cache-to-cache transfer time is twelve time units.

Therefore, in system 1, a cache-to-cache transfer takes less time than a memory-to-cache transfer. It is the opposite in system 2.

In the following, we will express all penalties in units of the penalty of transferring a single word between a cache and the shared memory, that is, $\lambda_{word} = 1$. If the penalty to read a word from memory is the same as the penalty to write a word to memory, then we can estimate the performance improvement due to the caching of shared writable data as $1 - \lambda_{total}$. In particular if $\lambda_{total} > 1$, then caching shared writable data is not productive. The penalties of different coherence events in system 1 are $t_{mc} = 10/7$, $t_{cc} = 8/7$, $t_{word} = 1$ and $t_{inv} = 2/7$; in system 2, they are $t_{mc} = 10/7$, $t_{cc} = 12/7$, $t_{word} = 1$ and $t_{inv} = 2/7$.

From Table ??, we can calculate the miss ratio, $M$, and the total penalty, $\lambda_{total}$, for the Jacobi iterative algorithm for the five different protocols. The results are compared to the results of execution-driven simulations for the five protocols in Table 7, Table 8 and Table 9. The difference between model predictions and simulations in most cases is never more than 12.5%.

Table 7: Comparison between the miss ratios of the model and of the simulation

| Protocol | Model Prediction | Simulation Result | Difference (%) |
|---|---|---|---|
| Basic | 0.004920 | 0.005518 | 10.83% |
| Write-Once | 0.004920 | 0.005518 | 10.83% |
| Synapse | 0.008665 | 0.008622 | 0.50% |
| Illinois | 0.004920 | 0.005518 | 10.83% |
| Berkeley | 0.004920 | 0.005518 | 10.83% |

Table 8: Comparison between the total penalties of the model and of the simulation (system 1: $t_{mc} = \frac{10}{7}$, $t_{cc} = \frac{8}{7}$, $t_{word} = \frac{7}{7}$, $t_{inv} = \frac{2}{7}$)

| Protocol | Model Prediction | Simulation Result | Difference (%) |
|---|---|---|---|
| Basic | 0.015141 | 0.016540 | 8.46% |
| Write-Once | 0.011190 | 0.011615 | 3.66% |
| Synapse | 0.023576 | 0.019955 | 18.15% |
| Illinois | 0.008030 | 0.008074 | 0.54% |
| Berkeley | 0.006825 | 0.007185 | 5.01% |

## 4.4 Discussion

The performance metrics derived from the model are the *miss ratio* and the *total coherence penalty*. This discussion applies for steady-state computations in systems with infinite caches, and for block sharing patterns for which the access burst model is acceptable.

Table 9: Comparison between the total penalties of the model and of the simulation (system 2: $t_{mc} = \frac{10}{7}$, $t_{cc} = \frac{12}{7}$, $t_{word} = \frac{7}{7}$, $t_{inv} = \frac{2}{7}$)

| Protocol | Model Prediction | Simulation Result | Difference (%) |
|---|---|---|---|
| Basic | 0.015141 | 0.016540 | 8.46% |
| Write-Once | 0.011545 | 0.011191 | 3.16% |
| Synapse | 0.023929 | 0.021288 | 12.41% |
| Illinois | 0.009636 | 0.010338 | 6.79% |
| Berkeley | 0.009636 | 0.010338 | 6.79% |

### 4.4.1 Miss Ratio

The miss ratio is given by the sum of $P(M\_cc)$ and $P(M\_mc)$. The Basic, the Write-once, the Illinois and the Berkeley coherence protocols have very similar Markov chains, and the same miss ratio, which is

$$P(M) = \frac{1}{l} \cdot \frac{(J-1) \cdot W}{1 + (J-1) \cdot W}. \tag{1}$$

The transition from state 1_$RW$ to state 2_$RO$ is replaced by a transition from D to 1_$V$ in the Synapse coherence protocol; in this case the miss ratio is

$$P(M) = \frac{1}{l} \cdot \frac{J \cdot (J-1) \cdot W}{(J-1+W) \cdot (1 + (J-1) \cdot W)}. \tag{2}$$

This value is always higher than the value of equation (1) since $J$ is always larger than $J-1+W$ ($W \leq 1$). The miss ratio is displayed in Figure 34 for different value of $J$ and for $W$=0.25 and in Figure 35 for different value of $W$ and for $J$=16. The access burst model predicts that the miss ratio on S-blocks shared by 16 processors is more than 0.6 $/l$, even for cases where $W$ is less than 15%. Also, when either $W$=0.25 and $J \geq 16$ or $J$=16 and $W \geq 0.25$, the miss ratio is insensitive to either $J$ or $W$; however, if we can double the value of $l$, the miss ratio will be halved. It appears therefore that the average burst length is critical to maintaining a good hit ratio on shared writable blocks.

### 4.4.2 Total Penalty

Figures 36 and 37 display the product (penalty $\times$ $l$) as a function of $W$ and $J$ when $f$=1, and Figures 38 and 39 show the product (penalty $\times$ $l$) as a function of $W$ and $J$ when $f$=0 (two extreme cases) for system 1. From these four figures, the Berkeley coherence protocol always shows the best performance. The Illinois coherence protocol always has less total penalty than the Write-once coherence protocol. The Basic or the Synapse coherence protocols always exhibit the worst performance. Our examples show that, under the access

burst model with $f$=1, with $J$ less than 10 and $W$=0.25, the Synapse coherence protocol shows the worst performance; however, when $J$ is larger than 10 and $W$=0.25, the Basic coherence protocol has the worst penalty for shared block accesses; when $f$=0, the total penalty of the Synapse protocol is always higher than that of the Basic protocol. These conclusions are similar to those of Archibald and Baer's, in [10].

The above conclusion may vary for different values of the penalties, which in turn depend on the architecture of the system. For system 2, Figures 40 and 41 display the product (penalty $\times$ $l$) as a function of $W$ and $J$ when $f$=1, and Figures 42 and 43 show the product (penalty $\times$ $l$) as a function of $W$ and $J$ when $f$=0. From these four figures, the Berkeley coherence protocol always has the same penalty as the Illinois coherence protocol since there is no extra time needed to update shared memory when the $CS\_E$ event occurs. The Write-once coherence protocol has the least penalty in most cases except that in the case of $W$=0.25, $f$=0 and $J \leq 4$ and in the case of $J$=16, $f$=0 and $W \geq 0.5$, the Illinois and the Berkeley coherence protocols show the best performance. The average penalty of the Illinois and the Berkeley coherence protocols is less than the average penalty of the Basic and the Synapse coherence protocols except in the case of $J$=16 and $W$ is less than 0.15. When work load model $f$ is equal to zero, the Synapse coherence protocol always shows the worst performance. However, when $f$ is equal to one, the Basic coherence protocol pays more penalty than the Synapse coherence protocol in the case where $W$=0.25 and $J$ is greater than 16 and in the case where $J$=16 and $W$ is larger than 0.1.

Overall, we have seen that the miss ratio is roughly the same for all coherence protocols. The coherence protocols can be ranked in terms of increasing penalty (or decreasing efficiency). For system 1, the order is: the Berkeley, the Illinois, the Write-once, the Synapse and the Basic coherence protocols. For system 2, the order is: the Write-once, the Berkeley, the Illinois, the Synapse and the Basic coherence protocols. Therefore, while the Write-once coherence protocol is an *average* coherence protocol for system 1, it becomes the best coherence protocol for system 2 because the cache-to-cache transfer time has only a minor effect on the Write-once coherence protocol. Hence, the choice of a coherence protocol is greatly affected by the system architecture and parameters. The model proposed in Chapter 2 can be used for rapid evaluations of various protocols for a given system.

## 4.5 Conclusion

In this chapter we have applied the access burst program model to five coherence protocols for the caching of shared writable blocks in multiprocessor systems with infinite caches. The trend towards large caches seems inevitable in general-purpose computing because of the large hit ratio required by more powerful processors and because of the expected availability in a few years of VLSI chips with several megabytes of memory. Iterative, compute-intensive algorithms occur very often in multiprocessor algorithms [12]. The hypothesis of infinite caches is obtained when the data cache is large enough to contain all the data accessed by each processor, and steady state is reached after the first iteration.

It is remarkable that all coherence components are very simple to analyze by using the access burst program model for accessing shared writable blocks in the infinite cache environment. The maximum number of parameters needed for the most complicated case is no

more than four. The infinite cache results for a given set of S-blocks are independent of all cache parameters (organization, replacement policy) and of the reference pattern to other blocks. This is a distinct advantage, considering the complexity of the problem investigated in this dissertation. Because the model can be easily derived analytically, efficient evaluation of the whole design space can be done.

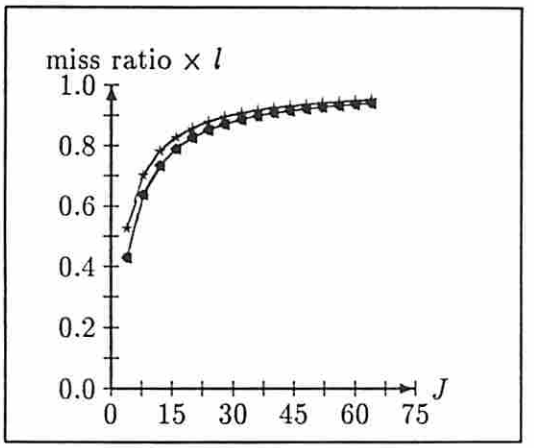



Figure 34: miss ratio × $l$ for the access burst model ($W$=0.25) (◁: Basic, ○: Write-Once, ⋆: Synapse, •: Illinois, ◇: Berkeley)

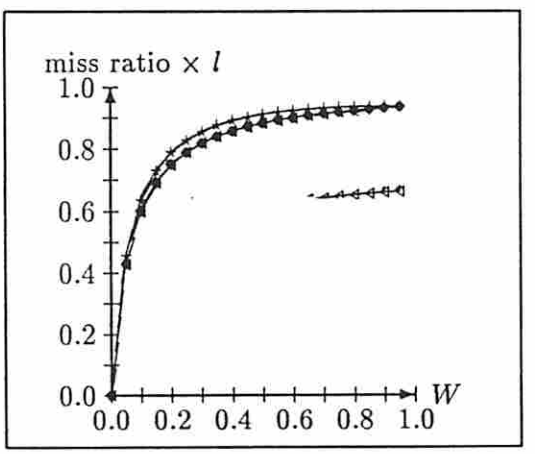



Figure 35: miss ratio × $l$ for the access burst model ($J$=16) (◁: Basic, ○: Write-Once, ⋆: Synapse, •: Illinois, ◇: Berkeley)

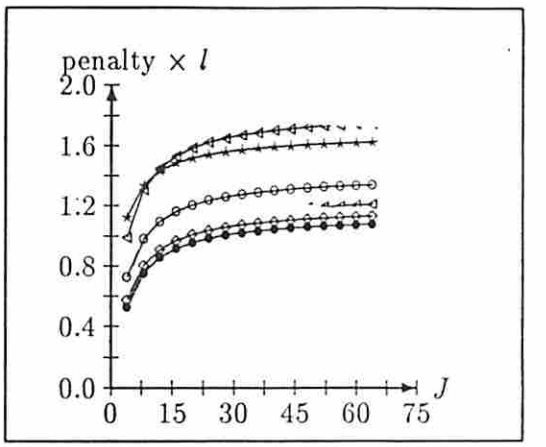



Figure 36: penalty × $l$ for system 1 ($W$=0.25, $f$=1) (◁: Basic, ○: Write-Once, ⋆: Synapse, •: Illinois, ◇: Berkeley)

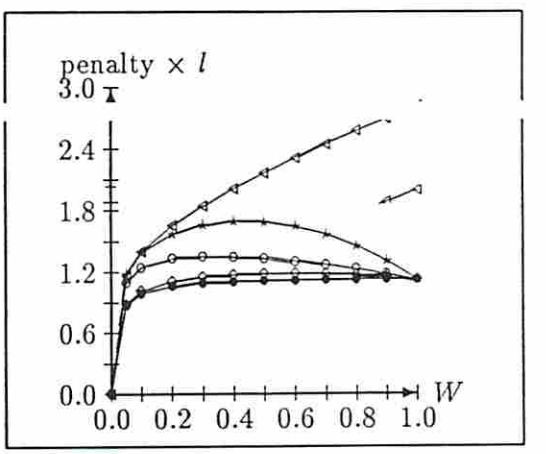



Figure 37: penalty × $l$ for system 1 ($J$=16, $f$=1) (◁: Basic, ○: Write-Once, ⋆: Synapse, •: Illinois, ◇: Berkeley)

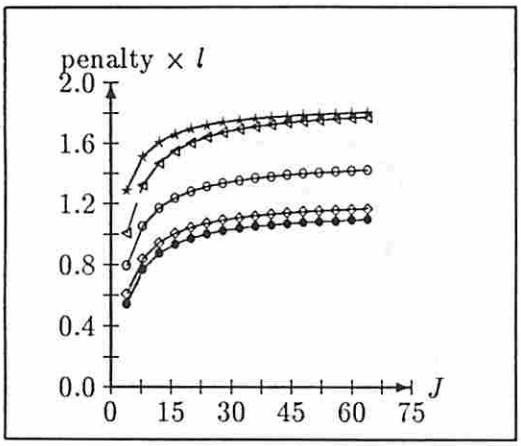



Figure 38: penalty × $l$ for system 1 ($W$=0.25 and $f$=0) (◁: Basic, ○: Write-Once, ⋆: Synapse, •: Illinois, ◇: Berkeley)

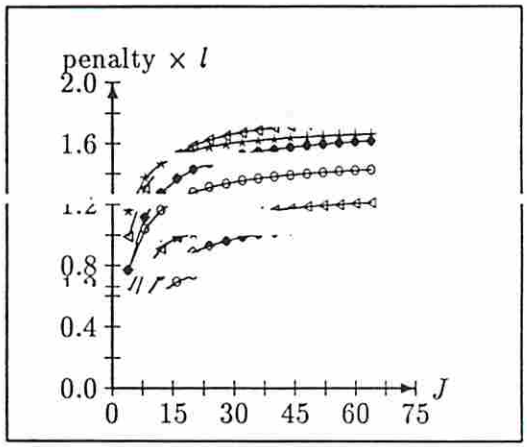



Figure 40: penalty × $l$ for system 2 ($W$=0.25, $f$=1) (◁: Basic, ○: Write-Once, ⋆: Synapse, •: Illinois, ◇: Berkeley)

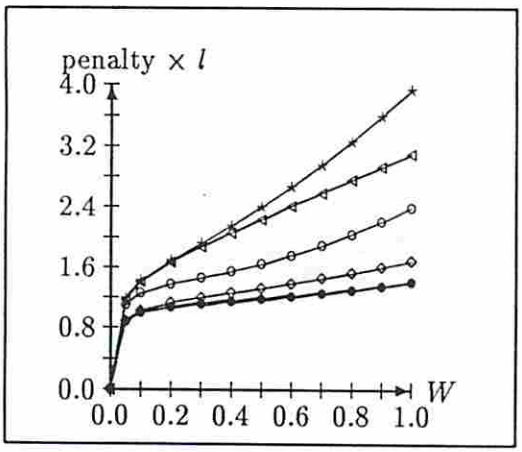



Figure 39: penalty × $l$ for system 1 ($J$=16 and $f$=0) (◁: Basic, ○: Write-Once, ⋆: Synapse, •: Illinois, ◇: Berkeley)

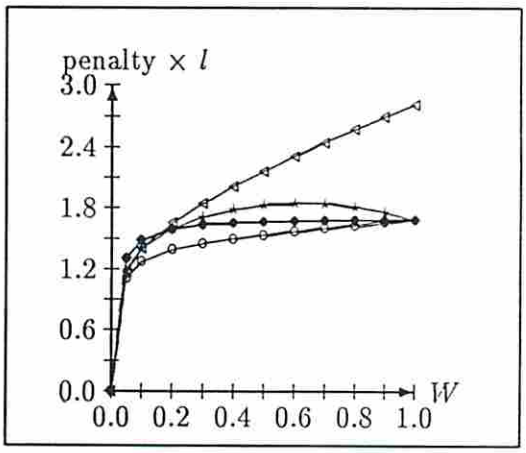



Figure 41: penalty × $l$ for system 2 ($J$=16, $f$=1) (◁: Basic, ○: Write-Once, ⋆: Synapse, •: Illinois, ◇: Berkeley)

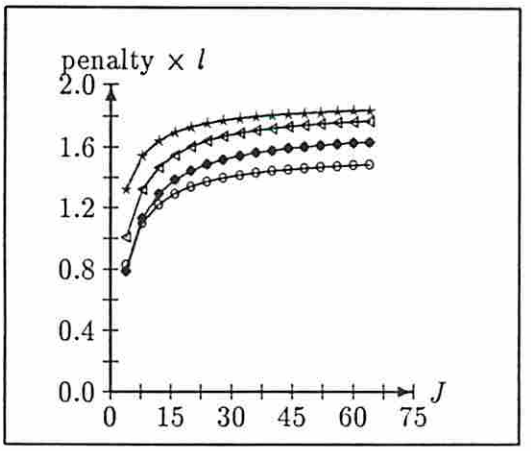



Figure 42: penalty × $l$ for system 2 ($W$=0.25 and $f$=0) (◁: Basic, ○: Write-Once, ⋆: Synapse, •: Illinois, ◇: Berkeley)

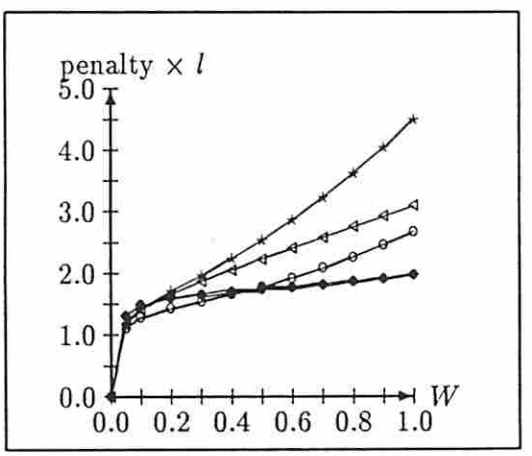



Figure 43: penalty × $l$ for system 2 ($J$=16 and $f$=0) (◁: Basic, ○: Write-Once, ⋆: Synapse, •: Illinois, ◇: Berkeley)

# 5 Finite Cache Effects

In this chapter, the finite cache effects are studied. First we prove some inclusion properties among similar finite cache systems and we compare their coherence overhead. Then we extend the access burst program model to the case of finite cache systems. Replacements are assumed to be uniformly distributed throughout the whole execution. We apply the model to the Basic cache coherence protocol and the finite cache system is modeled by a discrete Markov chain. An approximate solution is found for each component of the coherence overhead. Again, the accuracy of the model is verified by comparing the model predictions with execution-driven simulation results for several parallel algorithms.

# 6 Introduction

Although the infinite cache model provides a convenient environment for analyzing the execution of a wide class of problems on multiprocessors, in the real-world, cache size is always finite in the sense that one can find important problems whose working set size cannot fit into multiprocessor caches. In such a case, the infinite cache model fails to represent the real-world model and clearly finite cache effects should be explored.

The infinite cache model is much simpler to study than the real-world system with finite caches, mainly because it is described by fewer parameters. On the other hand, modeling finite-cache effects is very complex [?] because the number of parameters that potentially affect the outcome of the model is very large. In a finite-cache system, the hit ratio depends on many factors such as the block size, cache size, cache organization, cache replacement policy and cache coherence protocol [?].

## 6.1 Correlation Between Cache Size and Cache Coherence Overhead

In Chapters 2 and 4, we studied in detail five write-invalidate cache coherence protocols. In these write-invalidate cache coherence protocols, the cache events on a shared writable block in a multiprocessor system[1] are $M$ (Miss), $IN\_RO$, $CS\_RW$ and $IN\_RW$. These events and their associated penalty were identified for different protocols in Section 2.2 and in Chapter 4.

In this section, we will also denote the number of misses due to invalidations by $M_I$, that due to replacements by $M_R$, and that due to initial loading by $M_{init}$. The total number of misses is $M$ such that $M = M_I + M_R + M_{init}$. Furthermore, we will denote the subset of $IN\_RO$ events such that the local cache miss by $IN\_RO\_m$ and the local cache hit by $IN\_RO\_h$. Note that $IN\_RO = IN\_RO\_m + IN\_RO\_h$. Table 10 lists only those events in different protocols which have nonzero penalty associated with them (see Chapter 4.2).

We first identify the configurations of systems which can be compared with each other, based on the type of events listed in the previous section. The parameters which can affect the coherence overhead are:

[1] Different cache coherence protocols may have different names for the same cache coherence event.

Table 10: Cache coherence events with non-zero penalties

| Protocols | $M$ | $IN\_RO\_m$ | $IN\_RO\_h$ | $CS\_RW$ | $IN\_RW$ |
|---|---|---|---|---|---|
| Basic | $\otimes$ | $\otimes$ | $\otimes$ | $\otimes$ | $\otimes$ |
| Write-Once | $\otimes$ | | $\otimes$ | $\otimes$ | |
| Synapse | $\otimes$ | | $\otimes$ | $\otimes$ | |
| Illinois | $\otimes$ | | $\otimes$ | $\otimes$ | |
| Berkeley | $\otimes$ | | $\otimes$ | | |

- the cache size
- the cache block size
- the set size
- the replacement policy
- the cache coherence protocol (including write policy)
- the interleaved trace (this therefore includes the number of processors)

For two systems with different cache sizes to be compared, the following restrictions on these parameters must apply.

1. The cache block size has to be identical in both systems [?, ?].
2. The cache coherence protocols and the global interleaved trace have to be the same for all the systems under comparison so that the relative order of references from different processors is the same across different systems. Similar restrictions can be found in [?].

### 6.1.1 Inclusion Property

No Invalidation

A property called the *inclusion property* [?] must be satisfied in order to enable a comparison of two systems. In the absence of invalidations, the inclusion property can be stated as follows:

**Definition 1** Inclusion Property (no invalidation): *Cache $C_1$ includes cache $C_2$ if after any series of references, any block present in $C_2$ is also present in $C_1$.*

In the absence of invalidations, the inclusion property holds for systems that have the same block size, do not prefetch, and use the same set-mapping function (the same number of sets) with a stack replacement algorithm in each set [?]. Hill and Smith [?] have also proved the inclusion property for systems which have the same block size, do not prefetch

and use arbitrary set-mapping functions; the restrictions are that the replacement algorithm in each set must be LRU that the set-mapping function of the larger cache *refines*[2] that of the smaller cache and that the associativity of larger cache is no less than that of the smaller cache.

Invalidations

We now define an inclusion property in the presence of invalidations. The address trace is a vector of $N$ addresses $r(k)$, $k$=1,...,$N$. These records contain the block addresses of consecutive references made by a processor and are ordered in program order. There are a total of $I$ invalidations received by the processor. Each invalidation $i$, $i$=1,...,$I$ is characterized by a couple ($A_i$,$k_i$), where $A_i$ is the block address and $k_i$ is the position in the trace where the invalidation $i$ is inserted (that is between $r(k_i)$ and $r(k_{i+1})$). In the following, we will refer to index $k$ as the *time*.

In the presence of invalidations, a replacement policy may be *oblivious* to invalidations. In this case, the stack management algorithm for each set does not distinguish between valid or invalid blockframes. A replacement algorithm is *non-oblivious* to invalidations if it distinguishes between valid and invalid blockframes. The strategy we consider here for non-oblivious replacement is as follows. If there is any invalid block in a set at the time of replacement in that set, then the stack management chooses any one of the invalidated block; valid blocks are only victimized if there are no invalid blocks in the set.

In caches with invalidations, the inclusion property is defined as follows.

**Definition 2** Inclusion Property (with invalidations): *Cache $C_1$ includes cache $C_2$ iff, for all traces and for any set of invalidations, we have*

$$V(C_2) \subseteq V(C_1)$$

at all times $k$, where $V(C)$ is the set of valid blocks in $C$.

**Assumption 1** When we compare two systems with finite caches, $C_1$ and $C_2$ ($C_1 \geq C_2$), we restrict the caches to have the following features:

1. same cache block size
2. LRU replacement in each set
3. set-mapping function of $C_1$ refines that of $C_2$
4. associativity of $C_1$ is at least equal to that of $C_2$

Under these assumption, the inclusion property holds in the absence of invalidations, that is, $C_2 \subseteq C_1$.

[2]Set-refinement: Set-mapping function $f_2$ refines set-mapping function $f_1$ if $f_2(x) = f_2(y)$ implies $f_1(x) = f_1(y)$, for all blocks $x$ and $y$ [?].

**Proposition 1** *If the replacement algorithm is oblivious to invalidations, and under Assumption 1, the inclusion property holds in the presence of invalidations.*

**Proof:** First of all we extend the addresses in the original trace by $log_2 I$ bits in the most significant bit positions. Initially, these extensions are all set to zeroes in all trace records. We also assume that the tags in cache have been extended by $log_2 I$ bits; tag matching is done on the address and its extension.

We preprocess the extended trace as follows. For each $i$, $i$=1,..,$I$, we scan the trace records following record $k_{i+1}$. Each of these records with address $A_i$ have their address extension set to $i$, the identity of the invalidation. The timestamp of a memory block with address $A$ at time $k$ is either zero or the identity of the latest invalidation to address $A$.

This procedure simply renames the memory blocks after invalidations. Since the inclusion property is independent of the trace and since for LRU the replacement is not affected by the value of addresses, the inclusion property also holds for the extended trace and the cache with the tag extensions in the most significant bit positions, that is, ${C_2}^{ext} \subseteq {C_1}^{ext}$, in the sense of definition 1.

In the cache we can also see the extension field as a way to tag invalidations: a block with address $A$ is valid in the cache iff its tag extension is equal to its current timestamp. From this perspective, there is a one to one correspondence between each block contained in the cache with tag extension and each block contained in the cache without extension at any time $k$, and the inclusion property also holds for the cache with no tag extension. Namely, we have that if block $i$ is present and invalid in $C_2$, then it is also present and invalid in $C_1$; also if block $i$ is present and valid in $C_2$ it is also present and valid in $C_1$. Therefore $C_2 \subseteq C_1$ in the sense of definition 2. ■

In an infinite cache system, a special case of a larger cache system, the restrictions imposed on replacement algorithms and associativities become irrelevant as there are no replacements. The inclusion property between infinite and finite caches, therefore, holds with milder restrictions and these are

- same cache block size
- same cache coherence protocol
- same interleaved trace
- no prefetching

### 6.1.2 Finite Cache Systems

In this subsection, we compare the number of coherence events in two multiprocessor systems with different cache sizes. It is assumed that the inclusion property holds in the presence of invalidation, that is, $C_2 \subseteq C_1$. From the previous section, we know that $C_2 \subseteq C_1$ when $C_1$ has infinite size and same block size as $C_2$. Also if $C_1$ has finite size, we require that

Assumption 1 holds and that the replacement algorithm is oblivious to invalidations.

**Proposition 2.** *The number of invalidation misses in the system with larger cache size ($C_1$) is greater than or equal to that in the system with smaller cache size ($C_2$). That is,*

$$M_I(C_1,B) \geq M_I(C_2,B)$$

**Proof:** If $P_i$ makes two consecutive references (either Read or Write) to a block at time $t_a$ and time $t_c$ (see Figure 44), for the block stored in the cache of $P_i$, there are two possible cases to consider based on the references between $t_a$ and $t_c$:

1. No write by $P_j$ (j $\neq$ i) occurs between time $t_a$ and $t_c$. As a result, at time $t_c$, no miss due to invalidation occurs in $P_i$ regardless of the cache size.

2. At least one write by $P_j$ (j $\neq$ i) occurs between time $t_a$ and $t_c$ and the first write occurs at time $t_b$ (see Figure 44). At time $t_c$, in the system with larger cache size ($C_1$), an invalidation miss occurs in $P_i$ occurs provided no replacement takes place before time $t_b$; otherwise, the miss occurs at time $t_c$ due to a replacement. Because of the inclusion property, the number of invalidation misses in the system with $C_2$ cache can therefore never exceed that in the system with $C_1$ cache in this case.

For the Synapse coherence protocol, if $P_i$'s access at time $t_a$ is a Read, the argument given earlier for other protocols is valid. However, at time $t_a$, if the access is a Write to the block, and if the first reference by $P_j$ (j $\neq$ i) is a Read or a Write at time $t_b$, then the aforementioned argument can still be applied.
When $P_i$ references the block for the very first time, a miss occurs in both systems under consideration; but this is not an invalidation miss. ■

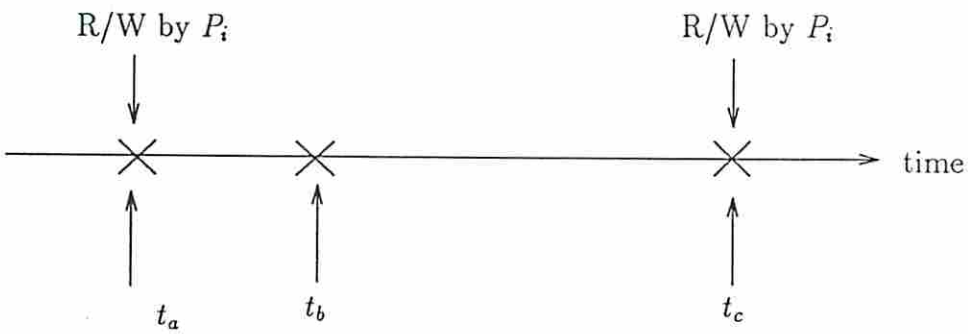


Figure 44: Trace for Proposition 2 and 3

**Proposition 3.** *The total number of misses in the system with larger cache size ($C_1$) is less than or equal to that in the system with smaller cache size ($C_2$). That is,*

$$M(C_2,B) \geq M(C_1,B).$$

**Proof:** This results directly from the inclusion property.

**Proposition 4.** *The number of $IN\_RO\_h$ events in the system with larger cache size ($C_1$) is greater than or equal to that in the system with smaller cache size ($C_2$). That is,*

$$IN\_RO\_h(C_1,B) \geq IN\_RO\_h(C_2,B).$$

**Proof:** Consider two consecutive Writes to a block made at time $t_a$ by $P_i$ and at time $t_b$ by $P_j$ (see Figure 45).

1. No Read by $P_k$ (k $\neq$ i) occurs between time $t_a$ and $t_b$. No $IN\_RO\_h$ event occurs in both systems at time $t_b$ in $P_j$ since either the state of the block in $P_i$ remains $RW$ (the only copy in the system) or the block has been replaced (no copy of the block exists in the systems) at time $t_b$.

2. $i = j$ and at least one Read by $P_k$ (k $\neq$ i) occurs between time $t_a$ and $t_b$. In the system with larger cache ($C_1$), an $IN\_RO\_h$ event occurs in $P_i$ at time $t_b$ provided no replacement takes place before time $t_b$; otherwise, no $IN\_RO\_h$ event occurs at time $t_b$ since no copy of the block exists in the system at time $t_b$. Because of the inclusion property, the number of $IN\_RO\_h$ events in the system with $C_2$ cache can therefore never exceed that in the system with $C_1$ cache in this case.

3. $i \neq j$ and at least one Read by $P_j$ between time $t_a$ and $t_b$. The argument is as in case 2.

4. $i \neq j$ and at least one Read by $P_k$ ($k \neq i$ and $k \neq j$) occurs between $t_a$ and $t_b$. No $IN\_RO\_h$ event occurs in both systems at time $t_b$ in $P_j$ since $P_j$ does not own the copy of the block at time $t_b$. ■

**Proposition 5.** *The number of $IN\_RO$ events in the system with larger cache size ($C_1$) is greater than or equal to that in the system with smaller cache size ($C_2$). That is,*

$$IN\_RO(C_1,B) \geq IN\_RO(C_2,B).$$

**Proof:** If $P_i$ writes to a block at time $t_a$ and the next write in the system to the same block is made by any processor $P_j$ at time $t_b$ (see Figure 45), two possible cases exist:

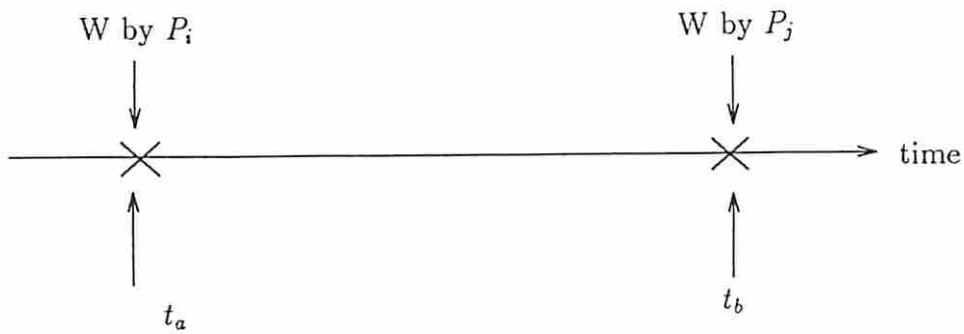


Figure 45: Trace for Proposition 4 and 5

1. No Read is made by $P_k$ (k $\neq$ i) between time $t_a$ and $t_b$. No $IN\_RO$ event occurs at time $t_b$ in both systems in $P_j$ since either the state of the block in $P_i$ remains $RW$ or the block has been replaced (i.e., no copy of the block exists in the systems) at time $t_b$.

2. At least one Read by $P_k$ (k $\neq$ i) occurs between time $t_a$ and $t_b$. In the system with larger cache ($C_1$), an $IN\_RO$ event occurs in $P_i$ at time $t_b$ provided no replacement takes place before time $t_b$; otherwise, no $IN\_RO$ event occurs at time $t_b$ since no copy of the block exists in the system at time $t_b$. Because of the inclusion property, the number of $IN\_RO$ events in the system with $C_2$ cache can therefore never exceed that in the system with $C_1$ cache in this case. ■

**Proposition 6.** *The number of $CS\_RW$ events in the system with larger cache size ($C_1$) is greater than or equal to that in the system with smaller cache size ($C_2$). That is,*

$$CS\_RW(C_1,B) \geq CS\_RW(C_2,B).$$

**Proof:** If $P_i$ writes to a block at time $t_a$ and the next Read by a different processor to the same block at time $t_b$ (see Figure 46), two possible cases exist:

1. At least one Write is made by $P_k$ (k $\neq$ i) between time $t_a$ and $t_b$. No $CS\_RW$ event occurs at time $t_b$ in $P_i$ since either the state of the block in $P_i$ has already been invalidated or the block has been replaced before time $t_b$.

2. No Write occurs is made by $P_k$ (k $\neq$ i) between time $t_a$ and $t_b$. In the system with larger cache ($C_1$), an $CS\_RW$ event occurs in $P_i$ at time $t_b$ provided no replacement takes place before time $t_b$; otherwise, no $CS\_RW$ event occurs at time $t_b$ since the block does not exist in $P_i$ at time $t_b$. Because of the inclusion property, the number of $CS\_RW$ events in the system with $C_2$ cache can therefore never exceed that in the system with $C_1$ cache in this case. ■

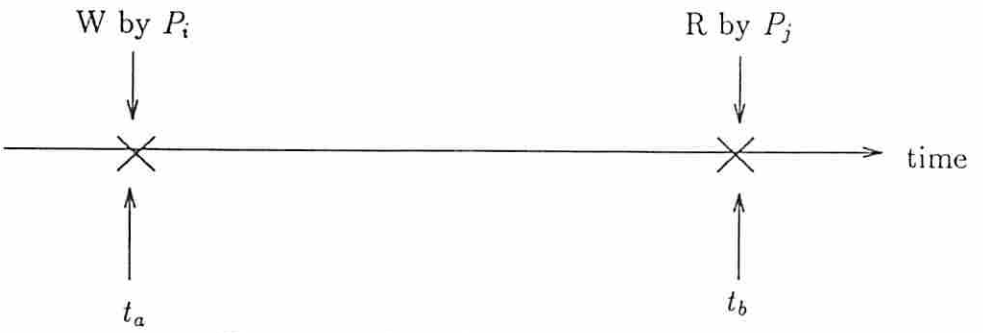


Figure 46: Trace for Proposition 6

**Proposition 7.** *The number of $IN\_RW$ events in the system with larger cache size ($C_1$) is greater than or equal to that in the system with smaller cache size ($C_2$). That is,*

$$IN\_RW(C_1,B) \geq IN\_RW(C_2,B).$$

**Proof:** If $P_i$ writes to a block at time $t_a$ and the next write to the same block by $P_j$ (j $\neq$ i) at time $t_b$ (see Figure 47), two possible cases exist:

1. At least one Read is made by $P_k$ (k $\neq$ i) between time $t_a$ and $t_b$. No $IN\_RW$ event occurs in the systems at time $t_b$ in $P_i$ since either the state of the block in $P_i$ has already been changed to *RO* or the block has been replaced (or has been invalidated for the Synapse protocol) before time $t_b$.

2. No Read by $P_k$ (k $\neq$ i) occurs between time $t_a$ and $t_b$. In the system with larger cache ($C_1$), an $IN\_RW$ event occurs in $P_i$ at time $t_b$ provided no replacement takes place before time $t_b$; otherwise, no $IN\_RW$ event occurs at time $t_b$ since the block does not exist in $P_i$ at time $t_b$. Because of the inclusion property, the number of $IN\_RW$ events in the system with $C_2$ cache can therefore never exceed that in the system with $C_1$ cache in this case. ■

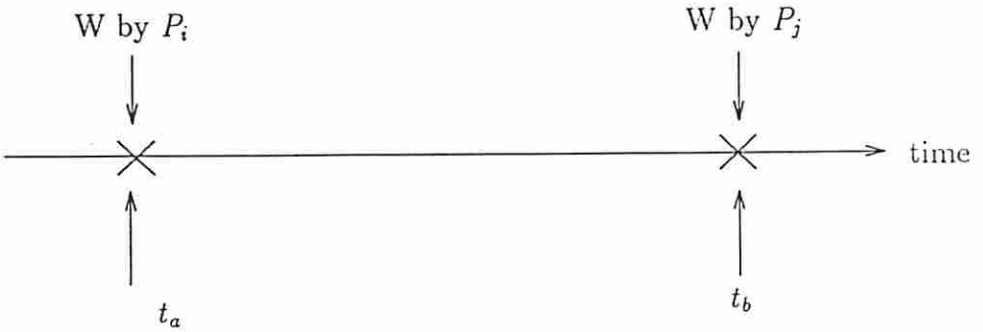


Figure 47: Trace for Proposition 7

When a processor modifies an S-block, it has to check the other caches in the system to invalidate all possible copies. This operation is called $Cross - interrogate$, and is denoted by $XI$. The number of $XI$ events is equal to the sum of the number of $IN\_RO$ events and $IN\_RW$ events.

**Corollary 1.** The number of $XI$ events in the system with larger cache size ($C_1$) is greater than or equal to that in the system with smaller cache size ($C_2$). That is,

$$XI(C_1,B) \geq XI(C_2,B).$$

**Corollary 2.**

$$\begin{gathered} M_I(\infty,B) \geq M_I(C,B) \\ M(C,B) \geq M(\infty,B) \\ IN\_RO\_h(\infty,B) \geq IN\_RO\_h(C,B) \\ IN\_RO(\infty,B) \geq IN\_RO(C,B) \\ CS\_RW(\infty,B) \geq CS\_RW(C,B) \\ IN\_RW(\infty,B) \geq IN\_RW(C,B) \\ XI(\infty,B) \geq XI(C,B) \end{gathered}$$

where $B$ is the cache block size and $C$ is the cache size.

## 6.2 The Access Burst Program Model for Finite Caches

In infinite cache systems, the access burst model can be characterized by parameters $q_s$, $J$, $W$ , $l$ and $f$ which have been discussed in detail in Chapter 2. In finite cache systems, replacements are assumed to be uniformly distributed throughout the whole execution and no replacement occurs in the processor which generates the current burst for S-block $i$. A new parameter $r_i$ for finite cache systems is defined as the fraction of number of replacements to number of accesses for the S-block $i$. Note that for infinite cache systems, the value of parameter $r_i$ is always zero since no replacement can occur.

The *global state* of S-block $i$ is described by the number of caches possessing a copy of the block and by the status RO or RW of the block. The global states are denoted by MEM, 1_RW, 1_RO, 2_RO, ..., $J$_RO, where MEM is the state in which no cache has a copy of S-block $i$. A state transition occurs at the time when a burst is completed by a processor or at the time when a copy of the S-block is replaced. The average number of accesses in between two state transitions, $b_i$, for S-block $i$ can be computed as follows:

$$b_i = \frac{l_i}{r_i \cdot l_i + 1},$$

where $l_i$ is the total number of accesses per access burst, and $r_i \cdot l_i + 1$ is the total number of state transitions in a burst which is the sum of the number of state transitions caused by replacements ($r_i \cdot l_i$) and an additional transition which ends the burst. The probability of starting a state transition is therefore $\frac{q_s \cdot p_i}{b_i}$. The probability that a state transition is caused by a replacement for S-block $i$, $R_i$, is therefore

$$R_i = \frac{r_i \cdot l_i}{r_i \cdot l_i + 1}. \tag{1}$$

The Markov chain for the state transitions of S-block $i$ is shown in Figure 48 (We have dropped the index $i$ in the Figure for clarity. Note that all parameters are for a given S-block $i$.) The transition probabilities from state $k\_RO$, $1 < k < J_i$, are found as follows.

1. From state $k\_RO$ to state $(k-1)\_RO$: The probability of this transition is the probability that a copy of the block is replaced, which is $R_i$.

2. From state $k\_RO$ to state $(k+1)\_RO$: The probability of this transition is the product of the probability that the transition is generated by the end of a burst, $(1 - R_i)$, of the probability that the next burst contains only Read accesses, $(1 - W_i)$, and of the probability that the access burst is made in one of the $J_i - k$ other caches, $(\frac{J_i - k}{J_i})$.

3. From state $k\_RO$ to state $k\_RO$: This is the case when the state transition is generated because of the end of a burst, and the next access burst contains only Read accesses in one of the $k$ caches. The transition probability is $(1 - W_i) \cdot (1 - R_i) \cdot \frac{k}{J_i}$.

4. From state $k\_RO$ to state $1\_RW$: This is the case when the state transition is generated because of the end of a burst, and the next access burst modifies the block. The transition probability is $W_i \cdot (1 - R_i)$.

The transition probabilities from states MEM, $1\_RO$, $1\_RW$ and $J_i\_RO$ are derived from similar arguments.

From the definitions of cache coherence events listed in Section ??, we can compute the probability of occurrence of each coherence event. When no copy of S-block $i$ is present in the system, a miss occurs at the beginning of a state transition, that is, at the beginning of a new access burst. If there are $k$ copies in the system, a miss occurs at the beginning of a new access burst when the next processor starting the access burst is one of the $(J_i - k)$ processors without a copy in their caches. Therefore, the fraction of references to S-block $i$ which miss in the cache is equal to the fraction of state transitions causing a miss divided by the average number of accesses in between two state transitions.

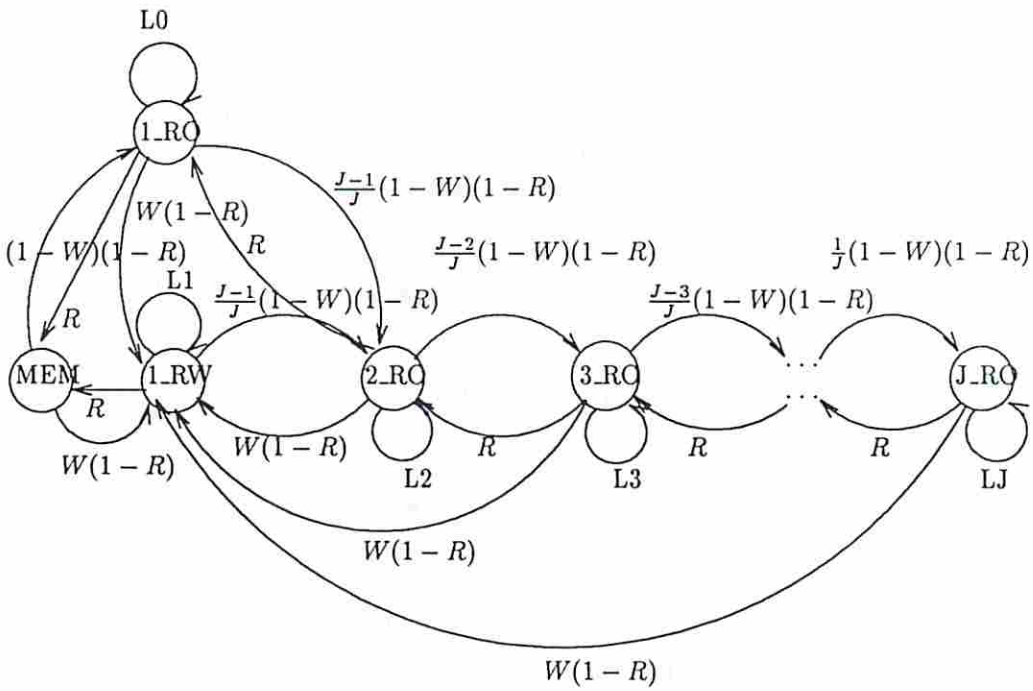
$$\begin{aligned} Pr(M_i) &= \frac{1}{b_i}\left\{ Pr(MEM) + Pr(1\_RW) \cdot \frac{(J_i - 1)}{J_i} \cdot (1 - R_i) \right. \\ &\quad \left. + \sum_{k=1}^{k=J_i-1} \frac{(J_i - k)}{J_i} \cdot (1 - R_i) \cdot Pr(k\_RO) \right\}. \end{aligned} \tag{2}$$

In order to find $\Pr(M_i)$ analytically, we first compute $XS_i$, the average number of copies of S-block $i$ in steady state.

$$XS_i = Pr(1\_RW) + \sum_{k=1}^{k=J_i} k \cdot Pr(k\_RO). \tag{3}$$

From Equations (2) and (3), we have

$$b_i \cdot Pr(M_i) = (1 - R_i) \cdot (1 - \frac{XS_i}{J_i}) + R_i \cdot Pr(MEM). \tag{4}$$




L0 : $(1-R)\cdot(1-W)\cdot\frac{1}{J}$
L1 : $(1-R)\cdot(1-W)\cdot\frac{1}{J}$
L2 : $(1-R)\cdot(1-W)\cdot\frac{2}{J}$
L3 : $(1-R)\cdot(1-W)\cdot\frac{3}{J}$
LJ : $(1-W)\cdot(1-R)$

Figure 48: Markov chain for the state transitions of an S-block shared by $J$ processors (finite cache case)

In steady state, the number of misses is equal to the number of copies which are either replaced or invalidated; this yields

$$b_i \cdot Pr(M_i) = R_i + \frac{J_i - 1}{J_i} \cdot XS_i \cdot W_i \cdot (1 - R_i). \tag{5}$$

From the above two equations, we have

$$XS_i = \frac{J_i \cdot (1 - 2 \cdot R_i + R_i \cdot Pr(MEM))}{(1 - R_i) \cdot (1 + (J_i - 1) \cdot W_i)}. \tag{6}$$

Substituting in Equation (4), we have

$$Pr(M_i) = r_i + \frac{1}{b_i} \cdot \frac{(J_i - 1) \cdot W_i \cdot (1 - 2R_i + R_i \cdot Pr(MEM))}{1 + (J_i - 1) \cdot W_i}.$$

If we lump the states $1\_RW$, and $k\_RO$ $(k = 1, \ldots, J_i)$ into a state called CHE, the Markov diagram in Figure 48 can be reduced to the Markov diagram in Figure 49. From the reduced discrete Markov diagram, the global flow balance equation can be written as

$$Pr(MEM) \cdot (1 - R_i) = [1 - Pr(MEM)] \cdot R_i \cdot \delta,$$

where $\delta$ is equal to $R \cdot \frac{Pr(1\_RO) + Pr(1\_RW)}{1 - Pr(MEM)}$ and $0 < \delta < 1$. Therefore,

$$R_i \geq Pr(MEM). \tag{7}$$

This yields

$$Pr(M_i) \leq r_i + \frac{1}{b_i} \cdot \frac{(J_i - 1) \cdot W_i \cdot (1 - R_i)^2}{1 + (J_i - 1) \cdot W_i}.$$

That is,

$$Pr(M_i) \leq r_i + \frac{1}{l_i} \cdot \frac{(J_i - 1) \cdot W_i \cdot (1 - R_i)}{1 + (J_i - 1) \cdot W_i}. \tag{8}$$

Similarly, the rest of the coherence events can be equated and solved analytically, which are listed as follows:
The fraction of accesses to S-block $i$ invalidating RO copies in other caches is given by

$$\begin{aligned} Pr(IN\_RO_i) &= \frac{1}{b_i} \cdot \left[ Pr(1\_RW) \cdot W_i \cdot \frac{J_i - 1}{J_i} \cdot (1 - f_i) \cdot (1 - R_i) \right. \\ &\quad \left. + \sum_{i=1}^{i=J_i} Pr(i\_RO) \cdot W_i \cdot (1 - R_i) \right]. \qquad (9) \\ &\geq \frac{1}{b_i} \frac{W_i (1 - R_i)^2 [J_i (1 - W_i f_i) - (1 - W_i)(1 - R_i) - W_i (1 - f_i)]}{J_i - (1 - W_i) \cdot (1 - R_i)} \\ &\geq \frac{1}{l_i} \frac{W_i (1 - R_i) [J_i (1 - W_i f_i) - (1 - W_i)(1 - R_i) - W_i (1 - f_i)]}{J_i - (1 - W_i) \cdot (1 - R_i)}. \end{aligned}$$

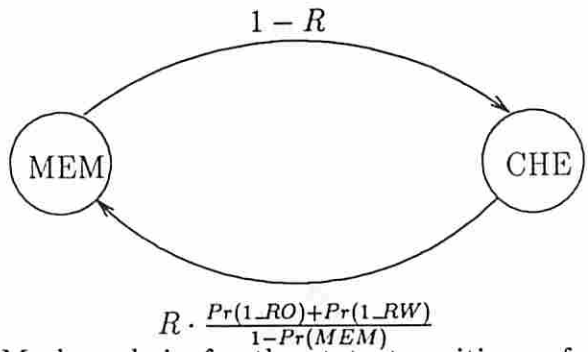


Figure 49: A reduced Markov chain for the state transitions of an S-block shared by $J$ processors (finite cache case)

The fraction of references to S-block $i$ changing the state from RW to RO is

$$\begin{aligned} Pr(CS\_RW_i) &= \frac{1}{b_i} \cdot \Big[ Pr(1\_RW) \cdot (1 - W_i) \cdot \frac{J_i - 1}{J_i} \cdot (1 - R_i) \\ &\quad + Pr(1\_RW) \cdot W_i \cdot \frac{J_i - 1}{J_i} \cdot (1 - f_i) \cdot (1 - R_i) \Big]. \\ &= \frac{1}{b_i} \cdot \frac{(J_i - 1) \cdot W_i \cdot (1 - W_i \cdot f_i) \cdot (1 - R_i)^2}{J_i - (1 - W_i) \cdot (1 - R_i)} \\ &= \frac{1}{l_i} \cdot \frac{(J_i - 1) \cdot W_i \cdot (1 - W_i \cdot f_i) \cdot (1 - R_i)}{J_i - (1 - W_i) \cdot (1 - R_i)}. \end{aligned} \tag{10}$$

The fraction of references to S-block $i$ causing such an event is therefore

$$\begin{aligned} Pr(IN\_RW_i) &= \frac{1}{b_i} \cdot \Big[ Pr(1\_RW) \cdot W_i \cdot \frac{J_i - 1}{J_i} \cdot f_i \cdot (1 - R_i) \Big]. \\ &= \frac{1}{b_i} \cdot \frac{(J_i - 1) \cdot W_i^2 \cdot f_i \cdot (1 - R_i)^2}{J_i - (1 - W_i) \cdot (1 - R_i)} \\ &= \frac{1}{l_i} \cdot \frac{(J_i - 1) \cdot W_i^2 \cdot f_i \cdot (1 - R_i)}{J_i - (1 - W_i) \cdot (1 - R_i)}. \end{aligned} \tag{11}$$

The second term on the right hand side of inequality (8) is an upper bound for the invalidation miss rate, $Pr(M_{I_i})$. The difference between this term and the miss rate in the infinite cache model is $(1 - R_i)$ which appears in the numerator of inequality (8). Since $R_i$ is a proper fraction, that is $0 < R_i < 1$, the above inequality supports Proposition 2 discussed in Section 6.1.2. Similarly, inequality (10) and Equalities (10) and (11) support Propositions 4,5, and 6, respectively.

Unlike the infinite cache systems, in the finite cache systems, only approximate system effects can be obtained since we assume that S-blocks are analyzed in isolation. This is done by summing the individual contributions of all S-blocks. In the finite cache model, when a system works in steady state, the invalidation miss rate generated by S-blocks is:

$$Pr(M_I) = q_s \cdot \sum_{i=1}^{N_s} p_i \cdot Pr(M_{I_i}), \tag{12}$$

where $N_s$ is the total number of shared writable blocks. The average system coherence penalty generated by S-blocks is:

$$\lambda_{coherence} = q_s \cdot \sum_{i=1}^{N_s} p_i \cdot \lambda_i, \tag{13}$$

where $\lambda_i = Pr(M_{I_i}) \cdot \lambda_M + Pr(IN\_RO_i) \cdot \lambda_{IN\_RO} + Pr(CS\_RW_i) \cdot \lambda_{CS\_RW} + Pr(IN\_RW_i) \cdot \lambda_{IN\_RW}$.

## 6.3 Multitasked Algorithms

In this section, the accuracy of the analytical model for finite cache systems is verified by comparing the model predictions with the execution-driven simulation results of seven parallel algorithms. The seven algorithms are the Jacobi iterative, the S.O.R. iterative, the quicksort, the bitonic merge sort, the non-shuffling FFT, the shuffling FFT and the image component labeling algorithms. Simulation methodology was described in Chapter 3.

For simplicity, we study only the direct mapping cache since there is no replacement policy associated with the cache organization. A probe is inserted in the execution-driven simulator to derive the values of parameter $r$ for the seven parallel algorithms, which are illustrated in Appendix D. With the values of parameters $J$, $W$, $l$, and $f$ discussed in Chapter 3, the system invalidation miss ratio and the system coherence penalty for the seven parallel algorithms are computed and shown in Figures 50 to Figure 63.

From these figures, the bigger the cache size is, the higher the invalidation miss ratio and coherence penalty. This is because the average life time of an S-block in the large cache is longer. As a result, the coherence penalty for the S-block can be expected to be higher. The model predictions and simulation results are close for the Jacobi iterative, the S.O.R. iterative, the quicksort and the image component labeling algorithms. The discrepancy observed between model predictions and simulation results for the non-shuffling FFT, the shuffling FFT and the bitonic merge sort algorithms can be explained as follows: In the shuffling FFT and bitonic merge sort algorithms, the number of bursts is equal to the number of synchronization points, which is usually less than ten. However there are thousands of replacements in the simulations. This violates the assumption that no replacement occurs in the processor which generates the current burst. For the non-shuffling FFT algorithm, S-blocks will definitely be replaced before the next burst begins. That is, the coherence penalty for S-blocks is equal to zero. From the equation 1, we can see that $R$ never takes the value of one. This implies the coherence penalty is never zero for the non-shuffling FFT algorithm; therefore, the model predictions are far from accurate for the three algorithms. The model has to be further refined to take care of such inaccuracies.

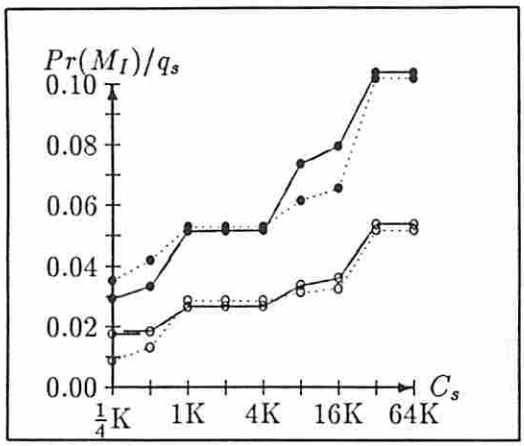



Figure 50: The system invalidation miss ratio for the Jacobi iterative algorithm ($P$=4, grid size of 128 × 128) (plain line: simulation results, dotted line: model predictions) (•: $B$=4, o: $B$=8)

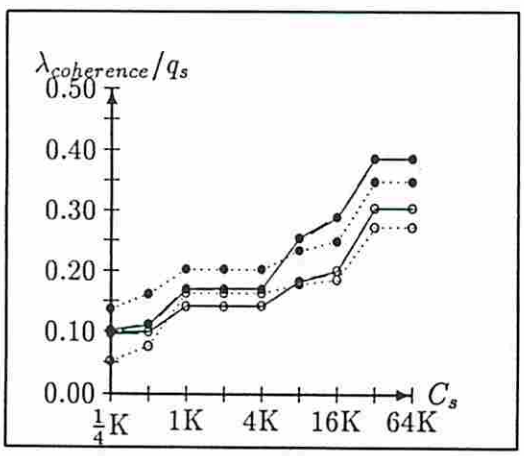



Figure 51: The system coherence penalty for the Jacobi iterative algorithm ($P$=4, grid size of 128 × 128) (plain line: simulation results, dotted line: model predictions) (•: $B$=4, o: $B$=8)

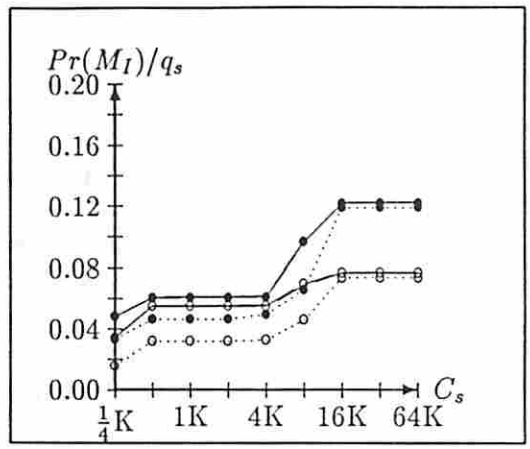



Figure 52: The system invalidation miss ratio for the S.O.R. iterative algorithm ($P$=4, grid size of 128 × 128) (plain line: simulation results, dotted line: model predictions) (•: $B$=4, o: $B$=8)

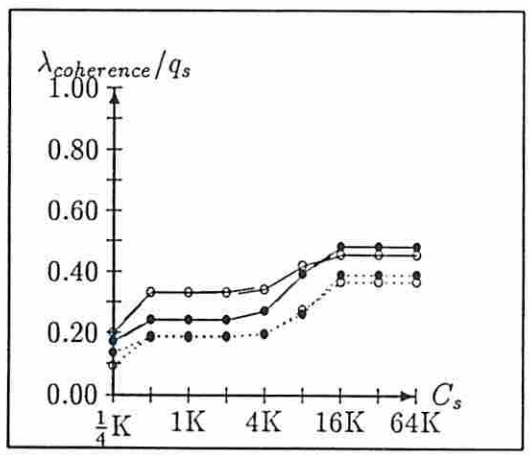



Figure 53: The system coherence penalty for the S.O.R. iterative algorithm ($P$=4, grid size of 128 × 128) (plain line: simulation results, dotted line: model predictions) (•: $B$=4, o: $B$=8)

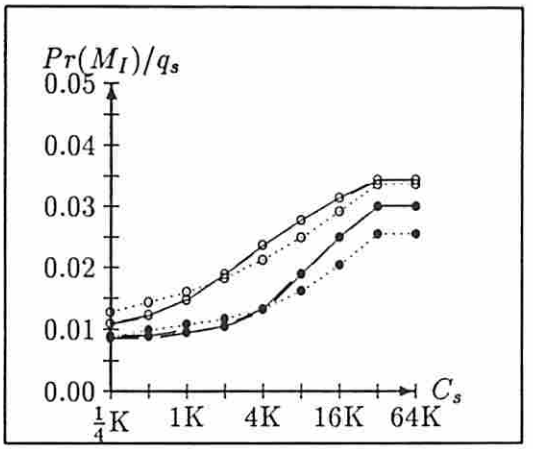



Figure 54: The system invalidation miss ratio for the dynamic quicksort ($B$=8, $N$=32768) (plain line: simulation results, dotted line: model predictions) (•: $P$=4, o: $P$=8)

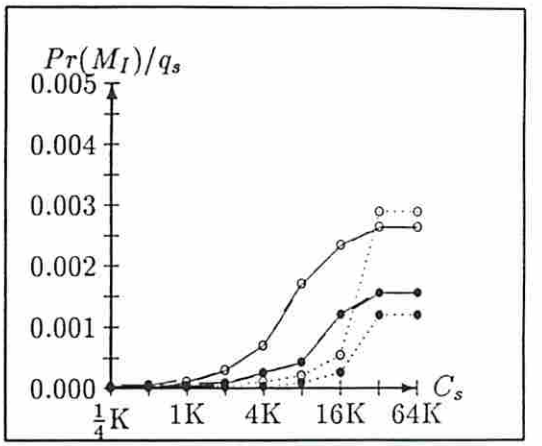



Figure 56: The system invalidation miss ratio for the bitonic merge sort ($B$=8, $N$=32768) (plain line: simulation results, dotted line: model predictions) (•: $P$=4, o: $P$=8)

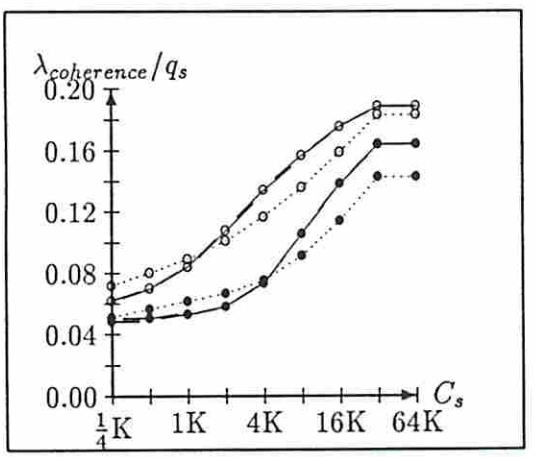



Figure 55: The system coherence penalty for the dynamic quicksort ($B$=8, $N$=32768) (plain line: simulation results, dotted line: model predictions) (•: $P$=4, o: $P$=8)

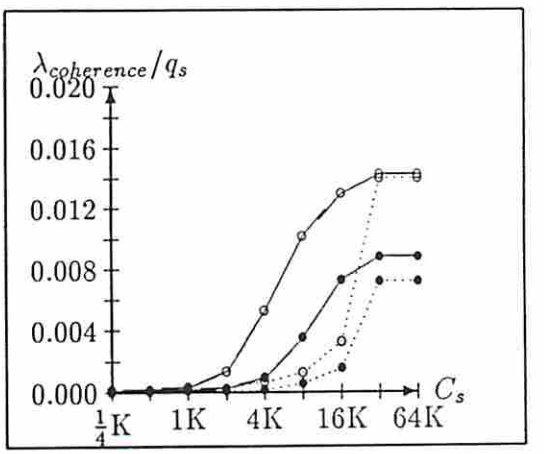



Figure 57: The system coherence penalty for the bitonic merge sort ($B$=8, $N$=32768) (plain line: simulation results, dotted line: model predictions) (•: $P$=4, o: $P$=8)

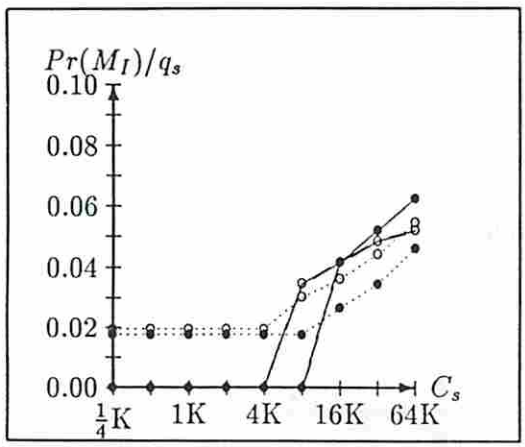



Figure 58: The system invalidation miss ratio for the non-shuffling FFT algorithm ($B$=8, $N$=65536) (plain line: simulation results, dotted line: model predictions) (•: $P$=4, o: $P$=8)

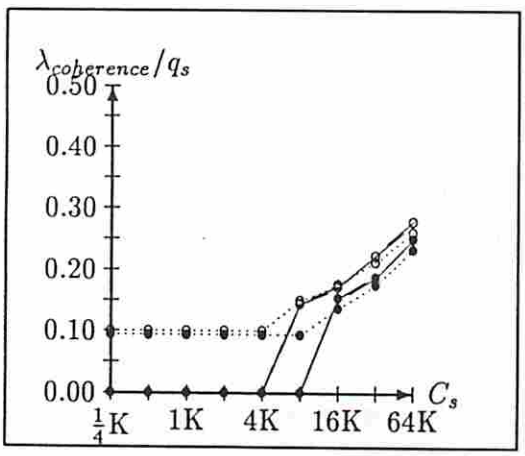



Figure 59: The system coherence penalty for the non-shuffling FFT algorithm ($B$=8, $N$=65536) (plain line: simulation results, dotted line: model predictions) (•: $P$=4, o: $P$=8)

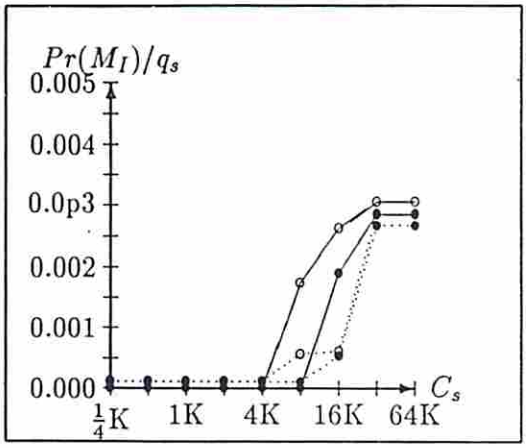



Figure 60: The system invalidation miss ratio for the shuffling FFT algorithm ($B$=8, $N$=65536) (plain line: simulation results, dotted line: model predictions) (•: $P$=4, o: $P$=8)

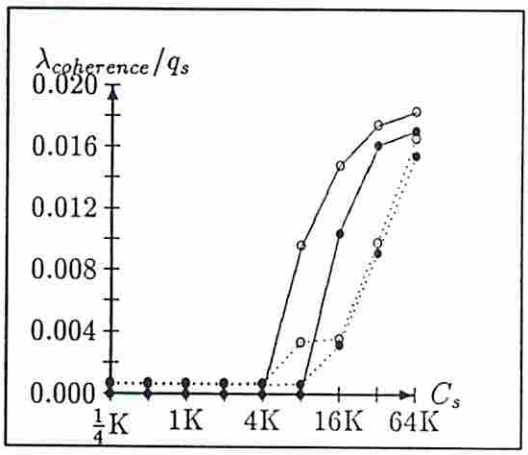



Figure 61: The system coherence penalty for the shuffling FFT algorithm ($B$=8, $N$=65536) (plain line: simulation results, dotted line: model predictions) (•: $P$=4, o: $P$=8)

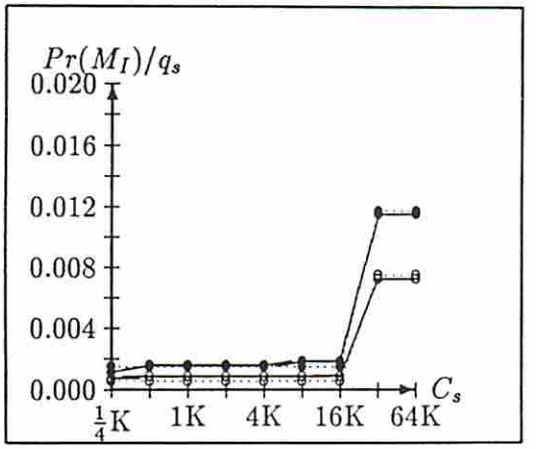



Figure 62: The system invalidation miss ratio for the image component labeling algorithm ($P$=4, grid size of 128 × 128) (plain line: simulation results, dotted line: model predictions) (•: $B$=4, ○: $B$=8)

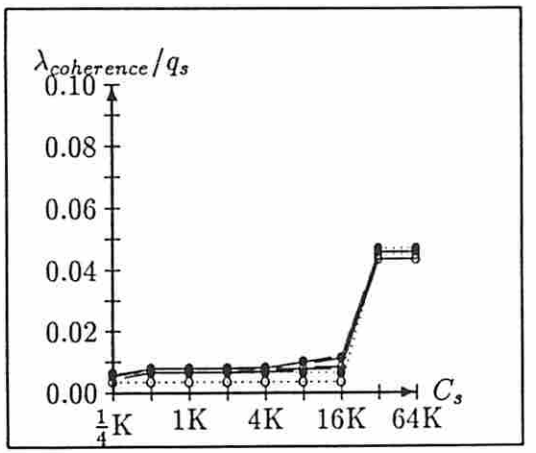



Figure 63: The system coherence penalty for the image component labeling algorithm ($P$=4, grid size of 128 × 128) (plain line: simulation results, dotted line: model predictions) (•: $B$=4, ○: $B$=8)

# 5 Summary and Conclusions

We have proposed a model of block sharing and solved the model analytically; we have compared the predictions from the model with the simulation results for eight important parallel algorithms, and we have applied the model to compare the effectiveness of five write-invalidate cache coherence protocols for accessing shared writable blocks. The model, based upon stochastic processes, appears to be a good approximation to shared block contention effects in multiprocessor systems. This chapter reiterates the most significant results of the dissertation and discusses directions for future research.

## 5.1 Concluding Remarks

In this dissertation, a simple program model, namely *the access burst program model*, used to capture the program behavior in terms of shared block accesses in cache-based multiprocessor systems was introduced. We started by analyzing a baseline system, the infinite cache multiprocessor system. In the infinite cache multiprocessor system, the traffic incurred in maintaining coherence is isolated and the traffic due to replacement is eliminated. Most parameters of the cache system, such as cache size, cache organization and cache replacement policy, do not affect the outcome of model predictions. The infinite cache system is therefore much simpler to study than the real system with finite caches.

Another reason to study the infinite cache system is because the trend towards large caches seems inevitable in general-purpose computing and also because of the large hit ratio required by more powerful processors and the expected availability of VLSI chips in a few years. In these caches, most of the misses are due to the initial loading of the data and to coherence invalidations. It is expected that the infinite cache model will become more and more relevant as the level of integration of memory chips increases.

We first applied the access burst program model on the Basic cache coherence protocol. When a processor completes an access burst for a shared writable block, all processors are assumed to have the same probability to start the next access burst to the block. The shared writable block accesses were modeled by a discrete Markov chain and an analytical closed-form solution was found for all components of the cache coherence overhead in the protocol. It is remarkable that all coherence components are very simple to analyze based on the access burst program model for accessing shared writable blocks in the infinite cache environment. The maximum number of parameters needed for the most complicated case is no more than four.

We then validated the accuracy of the access burst program model by comparing the model predictions with the execution-driven simulations of eight parallel algorithms. It appears that the model can demonstrate very accurate predictions on several parallel algorithms.

One of the applications for the access burst program model is to study and compare the effectiveness in handling shared writable blocks by different cache coherence protocols. The model was therefore applied to the five write-invalidate cache coherence protocols. Protocols are modeled by Markov chains; an analytical closed-form solution is derived for all components of the cache coherence overhead and for the protocols in systems with caches of infinite

sizes. It turns out that protocols perform differently for different block sizes and for different cache-to-cache and memory-to-cache transfer times.

Another reason that the infinite cache model is so attractive is because its outcome represents an upper bound on the coherence overhead in the finite cache system. In this dissertation, we clarified this point by proving a series of propositions. These propositions are true for all multiprocessor systems provided the hardware organizations in the two systems that we compare are identical except for cache size. The propositions are also true for any possible traces of accesses to the caches provided that the same trace of events drive all systems.

Finally, we studied the finite cache effects. In the finite cache systems, replacements are assumed to be uniformly distributed throughout the whole execution. Again we applied the access burst program model to the Basic cache coherence protocol and modeled the finite cache system by a discrete Markov chain. An approximate solution was found for each component of the coherence overhead. The accuracy of the model predictions was verified by comparing the model predictions with execution-driven simulation results for several parallel algorithms.

## 5.2 Suggestions for Future Work

The research presented in this dissertation provides a basis for futher investigations. Additional algorithms should be studied in order to understand the program behavior in detail. Different types of shared blocks have to be identified and classified based on the model parameter characteristic. Since different types of shared blocks may have very distinct reference patterns, different optimization strategies should be adopted. This information certainly helps parallel compilers to improve system performance.

Since the access burst program model is a good approximation to cache-based multiprocessor systems, it describes the parallel program behavior very well. This suggests that the model can serve as the basis for generating *artificial traces* which may be used to study future computer systems.

In the model all processors are assumed to have the same probability of starting the next access burst on the same block, after a burst is completed. It is clear that this may not be a good approximation for some parallel algorithms such as the non-shuffling FFT algorithm. Rather, it seems that a processor which has just finished an access burst has a higher probability of starting the next one. The model could therefore be refined by introducing a new parameter which captures this *affinity*.

Computing systems presently being built or simulated are in general limited to configurations of a small number of processors. Most parallel computers available commercially have 16 or less processors. Simulations are very time-consuming when the configuration of simulated systems are large; multiprocessor traces are not available for such systems. The access burst program model does not have these limitations to study the performance of systems with a large number of processors. With the estimation of the values of parameters for two, four, eight and sixteen processor systems, the values of parameters for the system with a large number of processors can be obtained through extrapolation. Of course, extensive simulations have to be performed in order to verify the model predictions.

Other areas for research include the effect of migration, preemption and dynamic scheduling. Solutions to all these problems are essential for the success of cache-based multiprocessor systems.

# References


[1] A. Agarwal. Analysis of cache performance for operating systems and multiprogramming. Technical Report CSL-TR-87-332, Computer Systems Laboratory, Stanford University, May 1987.

[2] A. Agarwal. A locality-based multiprocessor cache interference model. Technical report, Laboratory for Computer Science MIT, 1990.

[3] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. Technical Report CSL-TR-86-304, Computer Systems Laboratory, Stanford University, September 1986.

[4] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of 15th Annual International Symposium on Computer Architecture*, pages 280–289, June 1988.

[5] A. Agarwal, R.L. Sites, and M. Horowitz. ATUM: A new technique for capturing address trace using microcode. In *Proceedings of 11th Annual International Symposium on Computer Architecture*, pages 119–130, June 1984.

[6] A.O. Allen. *Probability, Statistics, and Queueing Theory.* Academic Press, 1978.

[7] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1), March 1983.

[8] J. Archibald and J.L. Baer. An economical solution to the cache coherence problem. In *Proceedings of 12th Annual International Symposium on Computer Architecture*, June 1985.

[9] J. Archibald and J.L. Baer. An evaluation of cache coherence solution in shared-bus multiprocessors. Technical Report TR85-10-05, University of Washington, 1985.

[10] J. Archibald and J.L. Baer. Cache-coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[11] J.L. Baer and W.H. Wang. Architectural choices for multi-level cache hierarchies. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 258–261, August 1987.

[12] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation.* Prentice-Hall, 1989.

[13] P. Bitar. A critique of trace-driven simulation for shared-memory multiprocessors. In M. Dubois and S.S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*, pages 37–52. Kluwer Academic Publishers, 1990.

[14] P. Bitar and A. Despain. Multiprocessor cache synchronization: Issues, innovations, evolution. In *Proceedings of 13th Annual International Symposium on Computer Architecture*, June 1986.

[15] B.R. Borgerson, M.D. Godfrey, P.E. Hagerty, and T.R. Rykken. The architecture of the sperry Univac 1100 series systems. In *Proceedings of 6th Annual International Symposium on Computer Architecture*, pages 137–146, April 1979.

[16] F.A. Briggs and M. Dubois. Effectiveness of private caches in multiprocessor systems with parallel-pipelined memories. *IEEE Transactions on Computers*, C-32(1), January 1983.

[17] L.M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[18] H. Cheong and A.V. Veidenbaum. Compiler-directed cache management in multiprocessors. *Computer*, 23(6):39–47, June 1990.

[19] Digital Signal Processing Committee, editor. *Programs for Digital Signal Processing.* IEEE Press, 1979.

[20] IBM Corporation. Special issue on IBM 3081. *IBM Journal of RES. and Devel.*, 26(1):2–19, January 1982.

[21] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, October 1971.

[22] Z. Cvetanovic. The effect of problem partitioning, allocation, and granularity on the performance of multiple-processor systems. *IEEE Transactions on Computers*, C-36(4), April 1987.

[23] G. Dahlquist and A. Bjorck. *Numerical Methods.* Prentice-Hall, 20 edition, 1974.

[24] F. Darema-Rogers, G.F. Pfister, and K. So. Memory access patterns of parallel scientific programs. Technical Report RC 12086 (No. 54146), IBM T.J. Watson Research Center, September 1986.

[25] P.J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[26] N. Deo, C.Y. Pang, and R.E. Lord. Two parallel algorithms for shortest path problems. In *Proceedings of the 1980 International Conference on Parallel Processing*, pages 244–253, August 1980.

[27] M. Dubois. A cache-based multiprocessor with high-efficiency. *IEEE Transactions on Computers*, C-34(10):968–972, October 1985.

[28] M. Dubois. Effect of invalidations on the hit ratio of cache-based multiprocessors. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 255–257, August 1987.

[29] M. Dubois. Throughput analysis of cache-based multiprocessors with multiple buses. *IEEE Transactions on Computers*, C-37(1):58–70, January 1988.

[30] M. Dubois and F.A. Briggs. Effects of cache coherency in multiprocessors. *IEEE Transactions on Computers*, C-31(11):1083–1099, November 1982.

[31] M. Dubois, F.A. Briggs, I. Patil, and M. Balakrishnan. Trace-driven simulations of parallel and distributed algorithms in multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 909–916, August 1986.

[32] M. Dubois and S.S. Thakkar, editors. *Memory-Access Penalties in Write-Invalidate Cache Coherence Protocols*, pages 109–130. Kluwer Academic Publishers, 1990.

[33] M. Dubois and J.C. Wang. Shared data contention in a cache coherence protocol. *IEEE Transactions on Computers.*

[34] M. Dubois and J.C. Wang. Program models for data sharing and their application to cache-based multiprocessors. Technical Report CRI 86-27, CRI Department of Electrical Engineering, University of Southern California, August 1986.

[35] M. Dubois and J.C. Wang. Program models for data sharing and their application to cache-based multiprocessors. In *Proceedings of International Computer Symposium 1988*, pages 1371–1376, December 1988.

[36] M. Dubois and J.C. Wang. Shared data contention in a cache coherence protocol. Technical Report CRI 88-26, CRI Department of Electrical Engineering, University of southern California, 1988.

[37] M. Dubois and J.C. Wang. Shared data contention in a cache coherence protocol. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 146–155, August 1988.

[38] M. Dubois and J.C. Wang. Analytical modeling of data sharing in cache based multiprocessors. Technical Report CENG 89-18, Computer Engineering Division, Electrical Engineering - Systems Department, University of Southern California, August 1989.

[39] S.J. Eggers. *Simulation Analysis of Data Sharing in Shared Memory Multiprocessors.* PhD thesis, University of California, Berkeley, April 1989.

[40] S.J. Eggers and R.H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of 15th Annual International Symposium on Computer Architecture*, pages 373–382, June 1988.

[41] S.J. Eggers and R.H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of 16th Annual International Symposium on Computer Architecture*, pages 2–15, May 1989.

[42] S.J. Frank. Tightly coupled multiprocessor system speeds memory-access times. *Electronics*, 57(1):164–169, January 1984.

[43] B. Furht and V. Milutinovic. A survey of multiprocessor architectures for memory management. *Computer*, 20(3):48–67, March 1987.

[44] J.R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of 10th Annual International Symposium on Computer Architecture*, pages 124–131, June 1983.

[45] J.R. Goodman and P.J. Woest. The wisconsin multicube : A new large-scale cache-coherent multiprocessor. In *Proceedings of 15th Annual International Symposium on Computer Architecture*, pages 422–431, May 1988.

[46] H.A. Goosen and D.R. Cheriton. Predicting the performance of shared multiprocessor caches. In M. Dubois and S.S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*, pages 153–164. Kluwer Academic Publishers, 1990.

[47] A. Gupta and W.-D. Weber. Analysis of cache invalidation patterns in shared-memory multiprocessors. In M. Dubois and S.S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*, pages 83–107. Kluwer Academic Publishers, 1990.

[48] M.D. Hill and A.J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, C-38(12):1612–1630, December 1989.

[49] K. Hwang and F.A.Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[50] S.K. Johnson. *The Effects of Cache Coherence on the Performance of Parallel PDE Algorithms in Multiprocessor Systems.* PhD thesis, Rice University, 1988.

[51] A.K. Jones and P. Schwartz. Experience using multiprocessor systems - a status report. *Computing Surveys*, 12(2), June 1980.

[52] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator. Competitive snoopy caching. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 244–254, October 1986.

[53] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, July 1988.

[54] R.H. Katz, S.J. Eggers, D.A. Wood., C.L. Perkins and R.G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of 12th Annual International Symposium on Computer Architecture*, pages 276–283, June 1985.

[55] H.T. Kung. *Algorithms and Complexity: New Directions and Recent Results.* J.F. Traub Ed., New York: Academic Press, 1976.

[56] R.L. Lee, P-C Yew, and D.H. Lawrie. Multiprocessor cache design considerations. In *Proceedings of 14th Annual International Symposium on Computer Architecture*, June 1987.

[57] S.T. Leutenegger and M.K. Vernon. A mean-value performance analysis of a new multiprocessor architecture. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 167–176, May 1988.

[58] T. Lovett and S. Thakkar. The symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[59] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[60] E. McCreight. The Dragon computer system: An early overview. In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, July 1984.

[61] G.D. McNiven and E.S. Davidson. Analysis of memory referencing behavior for design of local memories. In *Proceedings of 15th Annual International Symposium on Computer Architecture*, pages 56–63, June 1988.

[62] S.L. Min and J.L. Baer. A timestamp-based cache coherence scheme. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 23–32, August 1989.

[63] R.L. Norton and J.L. Abraham. Using write-back cache to improve performance of multiuser multiprocessors. In *Proceedings of the 1982 International Conference on Parallel Processing*, pages 326–331, August 1982.

[64] M.S. Papamarcos and J.H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of 11th Annual International Symposium on Computer Architecture*, pages 348–354, June 1984.

[65] J.H. Patel. Analysis of multiprocessors with private cache memories. *IEEE Transactions on Computers*, C-31(4):296–304, April 1982.

[66] M.J. Quinn. *Design Efficient Algorithm for Parallel Computers.* McGraw-Hill, 1987.

[67] A. Rosenfeld and A. Kak. *Digital Picture Processing*, volume 1-2. Academic Press, 1982.

[68] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of 12th Annual International Symposium on Computer Architecture*, pages 340–347, June 1985.

[69] R. Sedgewick. *Quicksort.* New York: Garland Publishing, Inc., 1980.

[70] A.J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[71] A.J. Smith. Cache evaluation and the impact of workload choice. In *Proceedings of 12th Annual International Symposium on Computer Architecture*, pages 64–73, June 1985.

[72] A.J. Smith. Cache memory design : an evolving art. *IEEE Spectrum*, 24(12):40–44, December 1987.

[73] A.J. Smith. Line (block) size selection in cpu cache memories. *IEEE Transactions on Computers*, C-36(9):1063–1075, September 1987.

[74] J.E. Smith and J.R. Goodman. A study of instruction cache organizations and replacement policies. In *Proceedings of 10th Annual International Symposium on Computer Architecture*, pages 132–137, June 1983.

[75] J.R. Spirn. *Program Behavior : Models and Measurements.* Elsevier Computer Science Library, 1977.

[76] H.S. Stone. *High-Performance Computer Architecture.* Addison Wesley, 1987.

[77] C.P. Thacker and L.C. Stewart. Firefly: A multiprocessor workstation. In *Proceedings of the Second International Architectural Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.

[78] C.P. Thacker, L.C. Stewart, and E.H. Satterthwaite, Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, C-37(8):909–920, August 1988.

[79] J.G. Thompson. *Efficient Analysis of Caching Systems.* PhD thesis, University of California, Berkeley, September 1987.

[80] M.K. Vernon and M.A. Holliday. Performance analysis of multiprocessors cache consistency protocols using generalized timed petri nets. In *Proceedings of the Performance 1986 and ACM Sigmetrics 1986 Joint Computer Performance Modeling, Measurement and Evaluation*, pages 9–17, May 1986.

[81] M.K. Vernon, E.D. Lazowska, and J. Zahorjan. An accurate and efficient performance analysis technique for multiprocessor snooping cache-consistency protocols. In *Proceedings of 15th Annual International Symposium on Computer Architecture*, pages 308–315, June 1988.

[82] J.C. Wang and M. Dubois. A performance comparison of cache coherence protocols based on the access burst model. In *Proceedings of the Second Annual Parallel Processing Symposium*, pages 73–87, April 1988.

[83] J.C. Wang and M. Dubois. Correlation between cache size and coherence protocol overhead. In *Proceedings of International Computer Symposium 1990*, 1990.

[84] J.C. Wang and M. Dubois. A performance comparison of cache coherence protocols based on the access burst model. *Computer Systems Science and Engineering*, 5(3):147–158, July 1990.

[85] W.H. Wang. *Multilevel Cache Hierarchies.* PhD thesis, University of Washington, Seattle, September 1989.

[86] W.-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the Third International Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[87] A.W. Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of 14th Annual International Symposium on Computer Architecture*, pages 244–252, June 1987.

[88] Q. Yang and L.N. Bhuyan. A queueing network model for a cache coherence protocol on multiple-bus multiprocessors. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 130–137, August 1988.

[89] Q. Yang, L.N. Bhuyan, and B.C. Liu. Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor. *IEEE Transactions on Computers*, 38(8):1143–1153, Aughst 1989.

[90] D. Young. *Iterative Solution of Large Linear Systems.* Academic Press: New York, 1971.

# A Miss Ratio Derivation

Let's denote

$$X_1 = P_{1\_RW} + \sum_{k=2}^{J} k \cdot P_{k\_RO},$$

and

$$X_2 = P_{1\_RW} + \sum_{k=2}^{J} k^2 \cdot P_{k\_RO}.$$

After multiply both sides of equation (3.1) of Section 4.1 by $k$, we obtain for $k$=2,...,$J$

$$k \cdot [J - k \cdot (1 - W)] \cdot P_{k\_RO} = k \cdot [J - (k-1)] \cdot (1 - W) \cdot P_{(k-1)},$$

or

$$\begin{aligned} k \cdot [J - k \cdot (1 - W)] \cdot P_{k\_RO} = \quad & \{(k-1) \cdot [J - (k-1)] \cdot (1 - W) \\ + \quad & [J - (k-1)] \cdot (1 - W)\} \cdot P_{(k-1)}, \end{aligned}$$

or

$$\begin{aligned} & [k \cdot J - k^2 \cdot (1 - W)] \cdot P_{k\_RO} \\ = \quad & (1 - W) \cdot [J + (J-1) \cdot (k-1) - (k-1)^2] \cdot P_{(k-1)}, \end{aligned}$$

where $P_{(k-1)} = P_{1\_RW}$ for $k$=2 and $P_{(k-1)} = P_{(k-1)\_RO}$ otherwise.

Let's sum all the left hand sides of these equations and let's equate the result to the sum of the right hand sides; we have

$$\begin{aligned} & J \cdot X_1 - J \cdot P_{1\_RW} - (1 - W) \cdot X_2 + (1 - W) \cdot P_{1\_RW} \\ = \quad & J \cdot (1 - W) \cdot (1 - P_{J\_RO} + (J-1) \cdot (1 - W) \cdot (X_1 - J \cdot P_{J\_RO}) \\ & -(1 - W) \cdot (X_2 - J^2 \cdot P_{J\_RO}). \end{aligned}$$

After some simplification, we have

$$\begin{aligned} [J - (J-1) \cdot (1 - W)] \cdot X_1 \quad &= \quad J \cdot (1 - W) + (J - 1 + W) \cdot P_{1\_RW} \\ &= \quad J \end{aligned}$$

or

$$X_1 = \frac{J}{1 + (J-1) \cdot W}.$$

But

$$1 - \frac{X_1}{J} = \frac{J-1}{J} \cdot P_{1\_RW} + \sum_{k=2}^{J} \frac{J-k}{J} \cdot P_{k\_RO} = ls \cdot P(M).$$

Therefore,

$$P(M) = \frac{1}{ls} \cdot \frac{(J-1) \cdot W}{1 + (J-1) \cdot W}.$$

# B Source Codes

## B.1 Source Codes of the Jacobi Iterative Algorithm

```
;
; Process MASTER
;
   BEGIN
         wait_a:=0;     wait_b:=0;     sync:=0;
         modify:="no";     done:="no";
         barrier_1:="close";   barrier_2:="close";
         barrier_3:="close";
         prevmax:=0.0;
         P=4;
         FOR i:=2 STEP 1 UNTIL P DO
            CREATE process WORKER(i);
L1:      LOCK sync;   sync:=sync+1;   UNLOCK sync;
L2:      LOCK sync;
         IF sync != P THEN
         BEGIN
            UNLOCK sync;     GOTO L2;
         END
         ELSE BEGIN
            UNLOCK sync;
            wait_a:=0;        barrier_2:="close";
            LOCK barrier_1;   barrier_1:="open";   UNLOCK barrier_1;
         FOR m:=1 STEP 1 UNTIL N/sqrt(P) DO
            FOR n:=1 STEP 1 UNTIL N/sqrt(P) DO
               A[m,n]:=(B[m-1,n]+B[m+1,n]+B[m,n-1]+B[m,n+1])/4;
         maxi:=-999999;                    ; -999999 is the minus infty
         FOR m:=1 STEP 1 UNTIL N/sqrt(P) DO
         BEGIN
            sum:=0.0;
            FOR n:=1 STEP 1 UNTIL N/sqrt(P) DO
               sum:=sum+A[m,n];
            IF sum > maxi THEN maxi:=sum;
         END
         IF FABS(maxi-prevmax) > 0.000001 THEN   modify:="yes";
         prevmax:=maxi;
         LOCK wait_a;  wait_a:=wait_a+1;  UNLOCK wait_a;
L3:      LOCK wait_a;
         IF wait_a != P THEN
         BEGIN
            UNLOCK wait_a;     GOTO L3;
         END
         ELSE BEGIN
            UNLOCK wait_a;
            IF modify = "yes" THEN
            BEGIN
               modify:="no";
               wait_b:=0;       barrier_3:="close";
               LOCK barrier_2;  barrier_2:="open";  UNLOCK barrier_2;
            END
            ELSE BEGIN
```

```
            LOCK done;   done:="yes";   UNLOCK done;
            LOCK barrier_2;     barrier_2:="open";
            UNLOCK barrier_2;
            GOTO L5;
         END
      FOR m:=1 STEP 1 UNTIL N/sqrt(P) DO
         FOR n:=1 STEP 1 UNTIL N/sqrt(P) DO
            B[m,n]:=(A[m-1,n]+A[m+1,n]+A[m,n-1]+A[m,n+1])/4;
      maxi:=-999999;                    ; -999999 is the minus infty
      FOR m:=1 STEP 1 UNTIL N/sqrt(P) DO
      BEGIN
         sum:=0.0;
         FOR n:=1 STEP 1 UNTIL N/sqrt(P) DO
            sum:=sum+B[m,n];
         IF sum > maxi THEN maxi:=sum;
      END
      IF FABS(maxi-prevmax) > 0.000001 THEN
            modify:="yes";
      prevmax:=maxi;
      LOCK wait_b;   wait_b:=wait_b+1;   UNLOCK wait_b;
L4:   LOCK wait_b;
      IF wait_b != P THEN
      BEGIN
         UNLOCK wait_b;     GOTO L4;
      END
      ELSE BEGIN
         UNLOCK wait_b;
         IF modify = "yes" THEN
         BEGIN
            modify:="no";
            sync:=0;          barrier_1:="close";
            LOCK barrier_3;  barrier_3:="open";  UNLOCK barrier_3;
            GOTO L1;
         END
         ELSE BEGIN
            LOCK done;     done:="yes";     UNLOCK done;
            LOCK barrier_3;  barrier_3:="open";  UNLOCK barrier_3;
         END
L5:END

;
; Process WORKER(i)}
;
   BEGIN
      prevmax:=0.0;
L1:   LOCK sync;     sync:=sync+1;     UNLOCK sync;
L2:   LOCK barrier_1;
      IF barrier_1 != "open" THEN
      BEGIN
         UNLOCK barrier_1;     GOTO L2;
      END
      ELSE UNLOCK barrier_1;
      FOR m:={[(i-1) mod sqrt(P)]*N/sqrt(P)}+1 STEP 1
         UNTIL {[(i-1) mod sqrt(P)]+1}*N/sqrt(P) DO
```

```
            FOR n:={[(i-1)/sqrt(P)]*N/sqrt(P)}+1 STEP 1
               UNTIL {[(i-1)/sqrt(P)]+1}*N/sqrt(P) DO
               A[m,n]:=(B[m-1,n]+B[m+1,n]+B[m,n-1]+B[m,n+1])/4;
         maxi:=-999999;                    ; -999999 is the minus infty
         FOR m:={[(i-1) mod sqrt(P)]*N/sqrt(P)}+1 STEP 1
            UNTIL {[(i-1) mod sqrt(P)]+1}*N/sqrt(P) DO
         BEGIN
            sum:=0.0;
            FOR n:={[(i-1)/sqrt(P)]*N/sqrt(P)}+1 STEP 1
               UNTIL {[(i-1)/sqrt(P)]+1}*N/sqrt(P) DO
                  sum:=sum+A[m,n];
            IF sum > maxi THEN maxi:=sum;
         END
         IF FABS(maxi-prevmax) > 0.000001 THEN
            modify:="yes";
         prevmax:=maxi;
         LOCK wait_a;  wait_a:=wait_a+1;  UNLOCK wait_a;
L3:      LOCK barrier_2;
         IF barrier_2 != "open" THEN
         BEGIN
            UNLOCK barrier_2;     GOTO L3;
         END
         ELSE BEGIN
            UNLOCK barrier_2;
            LOCK done;
            IF done != "yes" THEN
               UNLOCK done;
            ELSE BEGIN
               UNLOCK done;
               GOTO L5;
            END
         END
         FOR m:={[(i-1) mod sqrt(P)]*N/sqrt(P)}+1 STEP 1
            UNTIL {[(i-1) mod sqrt(P)]+1}*N/sqrt(P) DO
            FOR n:={[(i-1)/sqrt(P)]*N/sqrt(P)}+1 STEP 1
               UNTIL {[(i-1)/sqrt(P)]+1}*N/sqrt(P) DO
               B[m,n]:=(A[m-1,n]+A[m+1,n]+A[m,n-1]+A[m,n+1])/4;
         maxi:=-999999;                    ; -999999 is the minus infty
         FOR m:={[(i-1) mod sqrt(P)]*N/sqrt(P)}+1 STEP 1
            UNTIL {[(i-1) mod sqrt(P)]+1}*N/sqrt(P) DO
         BEGIN
            sum:=0.0;
            FOR n:={[(i-1)/sqrt(P)]*N/sqrt(P)}+1 STEP 1
               UNTIL {[(i-1)/sqrt(P)]+1}*N/sqrt(P) DO
                  sum:=sum+B[m,n];
            IF sum > maxi THEN maxi:=sum;
         END
         IF FABS(maxi-prevmax) > 0.000001 THEN
            modify:="yes";
         prevmax:=maxi;
         LOCK wait_b;  wait_b:=wait_b+1;  UNLOCK wait_b;
L4:      LOCK barrier_3;
         IF barrier_3 != "open" THEN
         BEGIN
```

```
            UNLOCK barrier_3;      GOTO L4;
         END
         ELSE BEGIN
            UNLOCK barrier_3;
            LOCK done;
            IF done != "yes" THEN
            BEGIN
               UNLOCK done;      GOTO L1;
            END
            ELSE UNLOCK done;
         END
L5:END
```

## B.2 Source Codes of the S.O.R. Iterative Algorithm

```
;
; Process MASTER
;
   BEGIN
         wait_a:=0;      wait_b:=0;      sync:=0;
         modify:="no";      done:="no";
         barrier_1:="close";      barrier_2:="close";
         barrier_3:="close";
         prevmax:=0.0;
         P=4;
         FOR i:=2 STEP 1 UNTIL P DO
            CREATE process WORKER(i);
L1:      LOCK sync;      sync:=sync+1;      UNLOCK sync;
L2:      LOCK sync;
         IF sync != P THEN
         BEGIN
            UNLOCK sync;      GOTO L2;
         END
         else BEGIN
            UNLOCK sync;
            wait_a:=0;
            barrier_2:="close";
            LOCK barrier_1;      barrier_1:="open";
            UNLOCK barrier_1;
         FOR m:=1 STEP 1 UNTIL N/sqrt(P) DO
            FOR n:=1 STEP 1 UNTIL N/sqrt(P) DO
               IF (m+n) mod 2 = 0 THEN
               BEGIN
                  A[m,n]:=(1-w) A[m,n]
                     +w(A[m-1,n]+A[m+1,n]+A[m,n-1]+A[m,n+1])/4;
         maxi:=-999999;                    ; -999999 is the minus infty
         FOR m:=1 STEP 1 UNTIL N/sqrt(P) DO
         BEGIN
            sum:=0.0;
            FOR n:=1 STEP 1 UNTIL N/sqrt(P) DO
               sum:=sum+A[m,n];
```

```
            IF sum > maxi THEN maxi:=sum;
         END
         IF FABS(maxi-prevmax) > 0.000001 THEN  modify:="yes";
         prevmax:=maxi;
         LOCK wait_a;     wait_a:=wait_a+1;     UNLOCK wait_a;
L3:      LOCK wait_a;
         IF wait_a != P THEN
         BEGIN
            UNLOCK wait_a;     GOTO L3;
         END
         else BEGIN
            UNLOCK wait_a;
            IF modify = "yes" THEN
            BEGIN
               modify:="no";
               wait_b:=0;
               barrier_3:="close";
               LOCK barrier_2;     barrier_2:="open";
               UNLOCK barrier_2;
            END
            else BEGIN
               LOCK done;     done:="yes";     UNLOCK done;
               LOCK barrier_2; barrier_2:="open";  UNLOCK barrier_2;
               GOTO L5;
            END
         FOR m:=1 STEP 1 UNTIL N/sqrt(P) DO
            FOR n:=1 STEP 1 UNTIL N/sqrt(P) DO
               IF (m+n) mod 2 = 1 THEN
                  A[m,n]:=(1-w) A[m,n]
                     + w(A[m-1,n]+A[m+1,n]+A[m,n-1]+A[m,n+1])/4;
         maxi:=-999999;                    ; -999999 is the minus infty
         FOR m:=1 STEP 1 UNTIL N/sqrt(P) DO
         BEGIN
            sum:=0.0;
            FOR n:=1 STEP 1 UNTIL N/sqrt(P) DO
               sum:=sum+A[m,n];
            IF sum > maxi THEN maxi:=sum;
         END
         IF FABS(maxi-prevmax) > 0.000001 THEN  modify:="yes";
         prevmax:=maxi;
         LOCK wait_b;     wait_b:=wait_b+1;     UNLOCK wait_b;
L4:      LOCK wait_b;
         IF wait_b != P THEN
         BEGIN
            UNLOCK wait_b;     GOTO L4;
         END
         else BEGIN
            UNLOCK wait_b;
            IF modify = "yes" THEN
            BEGIN
               modify:="no";
               sync:=0;
               barrier_1:="close";
               LOCK barrier_3;     barrier_3:="open";
```

```
            UNLOCK barrier_3;
            GOTO L1;
         END
         else BEGIN
            LOCK done;     done:="yes";     UNLOCK done;
            LOCK barrier_3;     barrier_3:="open";
            UNLOCK barrier_3;
         END
L5:END

;
; Process WORKER(i)
;
   BEGIN
      prevmax:=0.0;
L1:   LOCK sync;     sync:=sync+1;     UNLOCK sync;
L2:   LOCK barrier_1;
      IF barrier_1 != "open" THEN
      BEGIN
         UNLOCK barrier_1;     GOTO L2;
      END
      ELSE UNLOCK barrier_1;
      FOR m:={[(i-1) mod sqrt(P)]*N/sqrt(P)}+1 STEP 1
        UNTIL ([(i-1) mod sqrt(P)]+1)*N/sqrt(P) DO
         FOR n:={[(i-1)/sqrt(P)]*N/sqrt(P)}+1 STEP 1
            UNTIL {[(i-1)/sqrt(P)]+1}*N/sqrt(P) DO
            IF (m+n) mod 2 = 0 THEN
               A[m,n]:=(1-w) A[m,n]
                  + w(A[m-1,n]+A[m+1,n]+A[m,n-1]+A[m,n+1])/4;
      maxi:=-999999;                    ; -999999 is the minus infty
      FOR m:={[(i-1) mod sqrt(P)]*N/sqrt(P)}+1 STEP 1
         UNTIL {[(i-1) mod sqrt(P)]+1}*N/sqrt(P) DO
      BEGIN
         sum:=0.0;
         FOR n:={[(i-1)/sqrt(P)]*N/sqrt(P)}+1 STEP 1
            UNTIL {[(i-1)/sqrt(P)]+1}*N/sqrt(P) DO
            sum:=sum+A[m,n];
         IF sum > maxi THEN maxi:=sum;
      END
      IF FABS(maxi-prevmax) > 0.000001 THEN  modify:="yes";
      prevmax:=maxi;
      LOCK wait_a;     wait_a:=wait_a+1;     UNLOCK wait_a;
L3:   LOCK barrier_2;
      IF barrier_2 != "open" THEN
      BEGIN
         UNLOCK barrier_2;     GOTO L3;
      END
      else BEGIN
         UNLOCK barrier_2;
         LOCK done;
         IF done != "yes" THEN
            UNLOCK done;
         else BEGIN
            UNLOCK done;
```

```
            GOTO L5;
         END
      END
      FOR m:={[(i-1) mod sqrt(P)]*N/sqrt(P)}+1 STEP 1
         UNTIL {[(i-1) mod sqrt(P)]+1}*N/sqrt(P) DO
         FOR n:={[(i-1)/sqrt(P)]*N/sqrt(P)}+1 STEP 1
            UNTIL {[(i-1)/sqrt(P)]+1}*N/sqrt(P) DO
            IF (m+n) mod 2 = 1 THEN
               A[m,n]:=(1-w) A[m,n]
                  + w(A[m-1,n]+A[m+1,n]+A[m,n-1]+A[m,n+1])/4;
      maxi:=-999999;                    ; -999999 is the minus infty
      FOR m:={[(i-1) mod sqrt(P)]*N/sqrt(P)}+1 STEP 1
         UNTIL {[(i-1) mod sqrt(P)]+1}*N/sqrt(P) DO
      BEGIN
         sum:=0.0;
         FOR n:={[(i-1)/sqrt(P)]}*N/sqrt(P) +1 STEP 1
            UNTIL {[(i-1)/sqrt(P)]+1}*N/sqrt(P) DO
            sum:=sum+A[m,n];
         IF sum > maxi THEN maxi:=sum;
      END
      IF FABS(maxi-prevmax) > 0.000001 THEN  modify:="yes";
      prevmax:=maxi;
      LOCK wait_b;     wait_b:=wait_b+1;     UNLOCK wait_b;
L4:   LOCK barrier_3;
      IF barrier_3 != "open" THEN
      BEGIN
         UNLOCK barrier_3;     GOTO L4;
      END
      else BEGIN
         UNLOCK barrier_3;
         LOCK done;
         IF done != "yes" THEN
         BEGIN
            UNLOCK done;     GOTO L1;
         END
         ELSE UNLOCK done;
      END
L5:END
```

## B.3 Source Codes of the Bitonic Merge Sort

```
;
; Process MASTER
;
   BEGIN
      a[N]:=999999; ; a[N] is the largest number in the array a[]
      done:=P;
      FOR i:=0 STEP 1 UNTIL P DO
         ps[i]:="idle";
      initialize Q to contain input file descriptor
      FOR i:=1 STEP 1 UNTIL P DO
```

```
            CREATE process WORKER(i);
L1:         LOCK Q;
            IF Q is empty THEN
            BEGIN
               IF ps[0]="busy" THEN
                  BEGIN
                     ps[0]:="idle";
                     LOCK done;   done:=done+1;   UNLOCK done;
                     IF done=P THEN
                     BEGIN
                        UNLOCK Q;      GOTO L3;
                     END
                     ELSE BEGIN
                        UNLOCK Q;      GOTO L1;
                     END
                  END
               ELSE
                  IF done=P THEN
                  BEGIN
                     UNLOCK Q;      GOTO L3;
                  END
                  ELSE BEGIN
                     UNLOCK Q;      GOTO L1;
                  END
            END
            ELSE BEGIN
                  IF ps[0]="idle" THEN
                  BEGIN
                     LOCK done;  done:=done-1;  UNLOCK done;
                  END
                  ps[0]:="busy";
                  delete Q's head file descriptor and store it in r and m
                     where r and m are the locations of the first and the
                     last elements of the file, respectively
                  UNLOCK Q
L2:               size:=m-r+1;
                  pivot:=a[r];      j:=m+1;      i:=r;
                  DO
                  BEGIN
                     DO i:=i+1;       while (a[i] < pivot);
                     DO j:=j-1;        while (a[j] > pivot);
                     IF (i < j) THEN
                     BEGIN
                        temp:=a[i];      a[i]:=a[j];      a[j]:=temp;
                     END
                  end WHILE (j >= i);
                  temp:=a[j];      a[j]:=a[r];      a[r]:=temp;
                  IF (j=r and m=i-1) then GOTO L1;
                  IF j=r THEN
                  BEGIN
                     r:=i;      GOTO L2;
                  END
                  IF i>=m THEN
                  BEGIN
```

```
                m:=j;     GOTO L2;
             END
             IF size/2 > (j-r+1) THEN
             BEGIN     ; return the shorter file descriptor to Q
                LOCK Q;
                insert r and j at the tail of Q;
                r:=i;
             END
             ELSE BEGIN
                LOCK Q
                insert i and m at the tail of Q;
                m:=j;
             END
             GOTO L2;
       END
L3:END

;
; Process WORKER(i)
;
   BEGIN
   L1:       LOCK Q;
         IF Q is empty THEN
         BEGIN
            IF ps[i]="busy" THEN
               BEGIN
                  ps[i]:="idle";
                  LOCK done;  done:=done+1;  UNLOCK done;
                  IF done=P THEN
                  BEGIN
                     UNLOCK Q;     GOTO L3;
                  END
                  ELSE BEGIN
                     UNLOCK Q;     GOTO L1;
                  END
               END
            ELSE
               IF done=P THEN
               BEGIN
                  UNLOCK Q;     GOTO L3;
               END
               ELSE BEGIN
                  UNLOCK Q;     GOTO L1;
               END
         END
         ELSE BEGIN
               IF ps[i]="idle" THEN
               BEGIN
                  LOCK done;  done:=done-1;  UNLOCK done;
               END
               ps[i]:="busy";
               delete Q's head file descriptor and store it in r and m
                  where r and m are the locations of the first and the
                  last elements of the file, respectively
```

```
                UNLOCK Q
L2:             size:=m-r+1;
                pivot:=a[r];      j:=m+1;      i:=r;
                DO
                BEGIN
                   DO i:=i+1;       while (a[i] < pivot);
                   DO j:=j-1;        while (a[j] > pivot);
                   IF (i < j) THEN
                   BEGIN
                      temp:=a[i];      a[i]:=a[j];      a[j]:=temp;
                   END
                end WHILE (j >= i);
                temp:=a[j];      a[j]:=a[r];      a[r]:=temp;
                IF (j=r and m=i-1) then GOTO L1;
                IF j=r THEN
                BEGIN
                   r:=i;      GOTO L2;
                END
                IF i>=m THEN
                BEGIN
                   m:=j;      GOTO L2;
                END
                IF size/2 > (j-r+1) THEN
                BEGIN      ; return the shorter file descriptor to Q
                   LOCK Q;
                   insert r and j at the tail of Q;
                   r:=i;
                END
                ELSE BEGIN
                   LOCK Q
                   insert i and m at the tail of Q;
                   m:=j;
                END
                GOTO L2;
          END
L3:END
```

## B.4 Source Codes of the Bitonic Merge Sort

```
;
; Process MASTER
;
   BEGIN
         wait_a:=0;      wait_b:=0;      switch:="A";
         barrier_a:="close";      barrier_b:="close";
         FOR p:=1 STEP 1 UNTIL P-1 DO
            CREATE process WORKER(p);
         p:=0;
         FOR l:=1 STEP l=2*l UNTIL l<N/P DO
            FOR sl:=l STEP sl=sl/2 UNTIL 1 DO
               FOR i:=0 STEP 1 UNTIL i<N/(2*P) DO
               BEGIN
```

```
                idx:=p*N/P+i/sl*2*sl+i%sl;
                IF (idx/(2*l))%2 = 0 THEN
                BEGIN
                   IF a[idx] < a[idx+sl] THEN
                   BEGIN
                      temp:=a[idx];
                      a[idx]:=a[idx+sl];
                      a[idx+sl]:=temp;
                   END
                END
                ELSE BEGIN
                   IF a[idx] > a[idx+sl] THEN
                   BEGIN
                      temp:=a[idx];
                      a[idx]:=a[idx+sl];
                      a[idx+sl]:=temp;
                   END
                END
             END
          FOR l:=N/P STEP l=2*l UNTIL l<N DO
          BEGIN
             FOR sl:=l STEP sl=sl/2 UNTIL N/(2*P) DO
             BEGIN
                IF switch = "A" THEN
                BEGIN
                   LOCK wait_a;      wait_a:=wait_a+1;
                   UNLOCK wait_a;
L1:                LOCK wait_a;
                   IF wait_a != P THEN
                   BEGIN
                      UNLOCK wait_a;      GOTO L1;
                   END
                   ELSE BEGIN
                      UNLOCK wait_a;
                      wait_b:=0;
                      barrier_b:="close";
                      switch:="B";
                      LOCK barrier_a;      barrier_a:="open";
                      UNLOCK barrier_a;
                   END
                END
                ELSE BEGIN
                   LOCK wait_b;      wait_b:=wait_b+1;
                   UNLOCK wait_b;
L2:                LOCK wait_b;
                   IF wait_b != P THEN
                   BEGIN
                      UNLOCK wait_b;      GOTO L2;
                   END
                   ELSE BEGIN
                      UNLOCK wait_b;
                      wait_a:=0;
                      barrier_a:="close";
                      switch:="A";
```

```
                LOCK barrier_b;     barrier_b:="open";
                UNLOCK barrier_b;
            END
        END
            FOR i:=0 STEP 1 UNTIL i<N/(2*P) DO
            BEGIN
                idx:=(N/(2*sl)<P)?
                    p/(2*P*sl/N)*2*sl+p%(2*P*sl/N)*(N/(2*P))+i:
                    p*N/P+i/sl*2*sl+i%sl;
                IF (idx/(2*l))%2 = 0 THEN
                BEGIN
                    IF a[idx] < a[idx+sl] THEN
                    BEGIN
                        temp:=a[idx];
                        a[idx]:=a[idx+sl];
                        a[idx+sl]:=temp;
                    END
                END
                ELSE BEGIN
                    IF a[idx] > a[idx+sl] THEN
                    BEGIN
                        temp:=a[idx];
                        a[idx]:=a[idx+sl];
                        a[idx+sl]:=temp;
                    END
                END
            END
        END
        FOR sl:=N/(4*P) STEP sl=sl/2 UNTIL 1 DO
            FOR i:=0 STEP 1 UNTIL i<N/(2*P) DO
            BEGIN
                idx:=(N/(2*sl)<P)?
                    p/(2*P*sl/N)*2*sl+p%(2*P*sl/N)*(N/(2*P))+i:
                    p*N/P+i/sl*2*sl+i%sl;
                IF (idx/(2*l))%2 = 0 THEN
                BEGIN
                    IF a[idx] < a[idx+sl] THEN
                    BEGIN
                        temp:=a[idx];
                        a[idx]:=a[idx+sl];
                        a[idx+sl]:=temp;
                    END
                END
                ELSE BEGIN
                    IF a[idx] > a[idx+sl] THEN
                    BEGIN
                        temp:=a[idx];
                        a[idx]:=a[idx+sl];
                        a[idx+sl]:=temp;
                    END
                END
            END
    END
END
```

```
;
; Process WORKER(i)
;
   BEGIN
         FOR l:=1 STEP l=2*l UNTIL l<N/P DO
            FOR sl:=l STEP sl=sl/2 UNTIL 1 DO
               FOR i:=0 STEP 1 UNTIL i<N/(2*P) DO
               BEGIN
                  idx:=p*N/P+i/sl*2*sl+i%sl;
                  IF (idx/(2*l))%2 = 0 THEN
                  BEGIN
                     IF a[idx] < a[idx+sl] THEN
                     BEGIN
                        temp:=a[idx];
                        a[idx]:=a[idx+sl];
                        a[idx+sl]:=temp;
                     END
                  END
                  ELSE BEGIN
                     IF a[idx] > a[idx+sl] THEN
                     BEGIN
                        temp:=a[idx];
                        a[idx]:=a[idx+sl];
                        a[idx+sl]:=temp;
                     END
                  END
               END
         FOR l:=N/P STEP l=2*l UNTIL l<N DO
         BEGIN
            FOR sl:=l STEP sl=sl/2 UNTIL N/(2*P) DO
            BEGIN
               IF switch = "A" THEN
               BEGIN
                  LOCK wait_a;      wait_a:=wait_a+1;
                  UNLOCK wait_a;
L1:               LOCK barrier_a;
                  IF barrier_a != "open" THEN
                  BEGIN
                     UNLOCK barrier_a;      GOTO L1;
                  END
                  ELSE UNLOCK barrier_a;
               END
               ELSE BEGIN
                  LOCK wait_b;      wait_b:=wait_b+1;
                  UNLOCK wait_b;
L2:               LOCK barrier_b;
                  IF barrier_b != "open" THEN
                  BEGIN
                     UNLOCK barrier_b;      GOTO L2;
                  END
               ELSE UNLOCK barrier_b;
               END
               FOR i:=0 STEP 1 UNTIL i<N/(2*P) DO
```

```
            BEGIN
               idx:=(N/(2*sl)<P)?
                  p/(2*P*sl/N)*2*sl+p%(2*P*sl/N)*(N/(2*P))+i:
                  p*N/P+i/sl*2*sl+i%sl;
               IF (idx/(2*l))%2 = 0 THEN
               BEGIN
                  IF a[idx] < a[idx+sl] THEN
                  BEGIN
                     temp:=a[idx];
                     a[idx]:=a[idx+sl];
                     a[idx+sl]:=temp;
                  END
               END
               ELSE BEGIN
                  IF a[idx] > a[idx+sl] THEN
                  BEGIN
                     temp:=a[idx];
                     a[idx]:=a[idx+sl];
                     a[idx+sl]:=temp;
                  END
               END
            END
         END
         FOR sl:=N/(4*P) STEP sl=sl/2 UNTIL 1 DO
            FOR i:=0 STEP 1 UNTIL i<N/(2*P) DO
            BEGIN
               idx:=(N/(2*sl)<P)?
                  p/(2*P*sl/N)*2*sl+p%(2*P*sl/N)*(N/(2*P))+i:
                  p*N/P+i/sl*2*sl+i%sl;
               IF (idx/(2*l))%2 = 0 THEN
               BEGIN
                  IF a[idx] < a[idx+sl] THEN
                  BEGIN
                     temp:=a[idx];
                     a[idx]:=a[idx+sl];
                     a[idx+sl]:=temp;
                  END
               END
               ELSE BEGIN
                  IF a[idx] > a[idx+sl] THEN
                  BEGIN
                     temp:=a[idx];
                     a[idx]:=a[idx+sl];
                     a[idx+sl]:=temp;
                  END
               END
            END
      END
END
```

## B.5 Source Codes of the Non-Shuffling FFT Algorithm

```
;
; Process MASTER
;
   BEGIN
        wait_a:=0;      wait_b:=0;      switch:="A";
        barrier_a:="close";     barrier_b:="close";
        FOR p:=1 STEP 1 UNTIL P-1 DO
           CREATE process WORKER(p);
        p:=0;
        FOR m:=1 STEP m:=m*2 UNTIL m<N/P DO
        BEGIN
           i:=2*m;
           FOR j:=1 STEP 1 UNTIL m DO
           BEGIN
              theta:=-1*PI*(j-1)/m;
              omega:=COMPLEX(COS(theta),SIN(theta));
              FOR k:=j+p*(N/P) STEP i UNTIL (p+1)*N/P DO
              BEGIN
                 jj:=k+m;
                 temp:=omega*DATA[jj];
                 DATA[jj]:=DATA[k]-temp;
                 DATA[k]:=DATA[k]+temp;
              END
           END
        END
        mask:=1;
        FOR m:=N/P STEP m:=m*2 UNTIL m<N DO
        BEGIN
           IF switch = "A" THEN
           BEGIN
              LOCK wait_a;      wait_a:=wait_a+1;
              UNLOCK wait_a;
L1:           LOCK wait_a;
              IF wait_a != P THEN
              BEGIN
                 UNLOCK wait_a;      GOTO L1;
              END
              ELSE BEGIN
                 UNLOCK wait_a;
                 wait_b:=0;
                 barrier_b:="close";
                 switch:="B";
                 LOCK barrier_a;      barrier_a:="open";
                 UNLOCK barrier_a;
              END
           END
           ELSE BEGIN
              LOCK wait_b;      wait_b:=wait_b+1;
              UNLOCK wait_b;
L2:           LOCK wait_b;
              IF wait_b != P THEN
              BEGIN
                 UNLOCK wait_b;      GOTO L2;
```

```
            END
            ELSE BEGIN
               UNLOCK wait_b;
               wait_a:=0;
               barrier_a:="close";
               switch:="A";
               LOCK barrier_b;     barrier_b:="open";
               UNLOCK barrier_b;
            END
         END
         i:=2*m;
         index:=0;
         FOR j:=1 STEP 1 UNTIL m DO
         BEGIN
            theta:=-1*PI*(j-1)/m;
            omega:=COMPLEX(COS(theta),SIN(theta));
            FOR k:=j+p*(N/P) STEP i UNTIL (p+1)*N/P DO
            BEGIN
               sign:=mask&p;
               IF sign = 0 THEN
                  local[index]:=DATA[k]+omega*DATA[k+m];
               ELSE local[index]:=DATA[k]-omega*DATA[k-m];
               index:=index+1;
            END
         END
         IF switch = "A" THEN
         BEGIN
            LOCK wait_a;     wait_a:=wait_a+1;
            UNLOCK wait_a;
L3:         LOCK wait_a;
            IF wait_a != P THEN
            BEGIN
               UNLOCK wait_a;     GOTO L3;
            END
            ELSE BEGIN
               UNLOCK wait_a;
               wait_b:=0;
               barrier_b:="close";
               switch:="B";
               LOCK barrier_a;     barrier_a:="open";
               UNLOCK barrier_a;
            END
         END
         ELSE BEGIN
            LOCK wait_b;     wait_b:=wait_b+1;
            UNLOCK wait_b;
L4:         LOCK wait_b;
            IF wait_b != P THEN
            BEGIN
               UNLOCK wait_b;     GOTO L4;
            END
            ELSE BEGIN
               UNLOCK wait_b;
               wait_a:=0;
```

```
                barrier_a:="close";
                switch:="A";
                LOCK barrier_b;     barrier_b:="open";
                UNLOCK barrier_b;
             END
          END
          index:=0;
          FOR k:=p*(N/P)+1 STEP 1 UNTIL (p+1)*N/P DO
          BEGIN
             DATA[k]:=local[index];
             index:=index+1;
          END
       mask:=mask*2;
       END
   END

;
; Process WORKER(i)
;
   BEGIN
       FOR m:=1 STEP m:=m*2 UNTIL m<N/P DO
       BEGIN
          i:=2*m;
          FOR j:=1 STEP 1 UNTIL m DO
          BEGIN
             theta:=-1*PI*(j-1)/m;
             omega:=COMPLEX(COS(theta),SIN(theta));
             FOR k:=j+p*(N/P) STEP i UNTIL (p+1)*N/P DO
             BEGIN
                jj:=k+m;
                temp:=omega*DATA[jj];
                DATA[jj]:=DATA[k]-temp;
                DATA[k]:=DATA[k]+temp;
             END
          END
       END
       mask:=1;
       FOR m:=N/P STEP m:=m*2 UNTIL m<N DO
       BEGIN
          IF switch = "A" THEN
          BEGIN
             LOCK wait_a;     wait_a:=wait_a+1;
             UNLOCK wait_a;
L1:          LOCK barrier_a;
             IF barrier_a != "open" THEN
             BEGIN
                UNLOCK barrier_a;     GOTO L1;
             END
             ELSE UNLOCK barrier_a;
          END
          ELSE BEGIN
             LOCK wait_b;     wait_b:=wait_b+1;
             UNLOCK wait_b;
L2:          LOCK barrier_b;
```

```
                IF barrier_b != "open" THEN
                BEGIN
                   UNLOCK barrier_b;     GOTO L2;
                END
             ELSE UNLOCK barrier_b;
             END
             i:=2*m;
             index:=0;
             FOR j:=1 STEP 1 UNTIL m DO
             BEGIN
                theta:=-1*PI*(j-1)/m;
                omega:=COMPLEX(COS(theta),SIN(theta));
                FOR k:=j+p*(N/P) STEP i UNTIL (p+1)*N/P DO
                BEGIN
                   sign:=mask&p;
                   IF sign = 0 THEN
                      local[index]:=DATA[k]+omega*DATA[k+m];
                   ELSE local[index]:=DATA[k]-omega*DATA[k-m];
                   index:=index+1;
                END
             END
                IF switch = "A" THEN
                BEGIN
                   LOCK wait_a;     wait_a:=wait_a+1;
                   UNLOCK wait_a;
L3:                LOCK barrier_a;
                   IF barrier_a != "open" THEN
                   BEGIN
                      UNLOCK barrier_a;     GOTO L3;
                   END
                   ELSE UNLOCK barrier_a;
                END
                ELSE BEGIN
                   LOCK wait_b;     wait_b:=wait_b+1;
                   UNLOCK wait_b;
L4:                LOCK barrier_b;
                   IF barrier_b != "open" THEN
                   BEGIN
                      UNLOCK barrier_b;     GOTO L4;
                   END
                ELSE UNLOCK barrier_b;
                END
             index:=0;
             FOR k:=p*(N/P)+1 STEP 1 UNTIL (p+1)*N/P DO
             BEGIN
                DATA[k]:=local[index];
                index:=index+1;
             END
          mask:=mask*2;
          END
    END
```

## B.6 Source Codes of the Shuffling FFT Algorithm

```
;
; Process MASTER
;
   BEGIN
         wait_a:=0;      wait_b:=0;
         barrier_a:="close";      barrier_b:="close";
         FOR p:=1 STEP 1 UNTIL P-1 DO
            CREATE process WORKER(p);
         p:=0;
         FOR iter:=1 STEP 1 UNTIL CEILING(LOG(N)/LOG(N/P)) DO
         BEGIN
         FOR m:=1 STEP m:=m*2 UNTIL m<N/P DO
         BEGIN
            i:=2*m;
            FOR j:=1 STEP 1 UNTIL m DO
            BEGIN
               theta:=-1*PI*(j-1)/m;
               omega:=COMPLEX(COS(theta),SIN(theta));
               FOR k:=j+p*(N/P) STEP i UNTIL (p+1)*N/P DO
               BEGIN
                  jj:=k+m;
                  temp:=omega*DATA[jj];
                  DATA[jj]:=DATA[k]-temp;
                  DATA[k]:=DATA[k]+temp;
               END
            END
         END
         LOCK wait_a;      wait_a:=wait_a+1;
         UNLOCK wait_a;
L1:      LOCK wait_a;
         if} wait_a != P then}
         BEGIN
            UNLOCK wait_a;      GOTO L1;
         END
         ELSE BEGIN
            UNLOCK wait_a;
            wait_b:=0;
            barrier_b:="close";
            SWITCH:="B";
            LOCK barrier_a;      barrier_a:="open";
            UNLOCK barrier_a;
         END
         index:=0;
         k:=p;
         if} k<P/2 then}
         BEGIN
            FOR i:=k*2*N/P+1 STEP 2 UNTIL i<(k+1)*2*N/P DO
               local[index]:=DATA[i];
            index:=index+1;
         END
         ELSE BEGIN
            k:=k-P/2;
            FOR i:=k*2*N/P+2 STEP 2 UNTIL (k+1)*2*N/P DO
```

```
            local[index]:=DATA[i];
         index:=index+1;
      END
      LOCK wait_b;     wait_b:=wait_b+1;
      UNLOCK wait_b;
L2:   LOCK wait_b;
      if} wait_b != P then}
      BEGIN
         UNLOCK wait_b;     GOTO L2;
      END
      ELSE BEGIN
         UNLOCK wait_b;
         wait_a:=0;
         barrier_a:="close";
         SWITCH:="A";
         LOCK barrier_b;     barrier_b:="open";
         UNLOCK barrier_b;
      END
      index:=0;
      FOR k:=p*(N/P)+1 STEP 1 UNTIL (p+1)*N/P DO
      BEGIN
         DATA[k]:=local[index];
         index:=index+1;
      END
      END
   END

;
; Process WORKER(i)
;
   BEGIN
      FOR iter:=1 STEP 1 UNTIL LCEIL(LOG(N)/LOG(N/P))  DO
      BEGIN
      FOR m:=1 STEP m:=m*2 UNTIL m<N/P DO
      BEGIN
         i:=2*m;
         FOR j:=1 STEP 1 UNTIL m DO
         BEGIN
            theta:=-1*PI*(j-1)/m;
            omega:=COMPLEX(COS(theta),SIN(theta));
            FOR k:=j+p*(N/P) STEP i UNTIL (p+1)*N/P DO
            BEGIN
               jj:=k+m;
               temp:=omega*DATA[jj];
               DATA[jj]:=DATA[k]-temp;
               DATA[k]:=DATA[k]+temp;
            END
         END
      END
      LOCK wait_a;     wait_a:=wait_a+1;
      UNLOCK wait_a;
L1:   LOCK barrier_a;
      if} barrier_a != "open" then}
      BEGIN
```

```
            UNLOCK barrier_a;     GOTO L1;
         END
         ELSE UNLOCK barrier_a;
         index:=0;
         k:=p;
         if} k<P/2 then}
         BEGIN
            FOR i:=k*2*N/P+1 STEP 2 UNTIL i<(k+1)*2*N/P DO
               local[index]:=DATA[i];
            index:=index+1;
         END
         ELSE BEGIN
            k:=k-P/2;
            FOR i:=k*2*N/P+2 STEP 2 UNTIL (k+1)*2*N/P DO
               local[index]:=DATA[i];
            index:=index+1;
         END
         LOCK wait_b;     wait_b:=wait_b+1;
         UNLOCK wait_b;
L2:      LOCK barrier_b;
         if} barrier_b != "open" then}
         BEGIN
            UNLOCK barrier_b;     GOTO L2;
         END
         ELSE UNLOCK barrier_b;
         index:=0;
         FOR k:=p*(N/P)+1 STEP 1 UNTIL (p+1)*N/P DO
         BEGIN
            DATA[k]:=local[index];
            index:=index+1;
         END
         END
   END
```

## B.7 Source Codes of the Single Source Shortest Path

```
;
; Process MASTER
;
   BEGIN
         msyn:= "yes";     wait:=0;     done:=0;
         FOR i:=2 STEP 1 UNTIL K DO
            CREATE process WORKER(i);
         FOR u:=1 STEP K UNTIL NODES DO
            D[u]:=999999;    ; 999999 is the infinite number
L1:      IF wait < K-1 THEN GOTO L1;
         D[SOURCE]:=0;
         initialize Q to contain SOURCE only;
L2:      LOCK Q;
         IF Q is empty THEN GOTO L3;
         delete Q's head node u;
         UNLOCK Q;
         msyn:="no";
```

```
            FOR each arc (u,v) that starts at u DO
               BEGIN       ; reach successor node of u
                  newdv:=D[u]+d(u,v);
                  LOCK D[v];
                  IF D[v] <= newdv THEN
                     UNLOCK D[v];
                  ELSE BEGIN
                     P[v]:=u;
                     D[v]:=newdv;
                     UNLOCK D[v];
                     msyn:="yes";     LOCK Q;     MYSN:="no";
                     IF v was never in Q THEN
                        insert v at the tail of Q;
                     IF v was in Q, but is not currently in Q THEN
                        insert v at the head of Q;
                     UNLOCK Q;
                  END
                  END
            msyn:="yes";
            GOTO L2;
L3:         IF wait = K-1 THEN GOTO L4;
            UNLOCK Q;
            GOTO L2;
L4:         done:=1;
            UNLOCK Q;
L5:         IF done < K THEN GOTO L5;
      END

;
; Process WORKER(i)
;
      BEGIN
            FOR u:=1 STEP K UNTIL NODES DO
               D[u]:=999999;    ; 999999 is the infinite number
L1:         IF msyn = "yes" THEN GOTO L3;
            LOCK Q;
            IF Q is empty THEN GOTO L2;
            delete Q's head node u;
            UNLOCK Q;
            FOR each arc (u,v) that starts at u DO
               BEGIN       ;  reach successor node of u
                  newdv:=D[u]+d(u,v);
                  LOCK D[v];
                  IF D[v] <= newdv THEN
                     UNLOCK D[v];
                  ELSE BEGIN
                     P[v]:=u;
                     D[v]:=newdv;
                     UNLOCK D[v];
                     LOCK Q;
                     IF v was never in Q THEN
                        insert v at the tail of Q;
                     IF v was in Q, but is not currently in Q THEN
                        insert v at the head of Q;
```

```
                UNLOCK Q;
              END
              END
        GOTO L1;
L2      UNLOCK Q;
        GOTO L1;
L3:     LOCK wait;      wait:=wait+1;      UNLOCK wait;
L4:     IF done > 0 THEN GOTO L5;
        IF msyn = "yes" THEN GOTO L4:
        LOCK wait;      wait:=wait-1;      UNLOCK wait;
        GOTO L1;
L5:      LOCK done;      done:=done+1;      UNLOCK done;
    END
```

## B.8 Source Codes of the Image Component Labeling

```
;
; Process MASTER
;
   BEGIN
        WAIT:=0;      SYNC:=0;
        MODIFY:="no";      DONE:="no";
        BARRIER_1:="close";      BARRIER_2:="close";
        FOR i:=1 STEP 1 UNTIL N DO
           FOR j:=1 STEP 1 UNTIL N DO
              B[i,j]:=i*N+j;
        FOR i:=2 STEP 1 UNTIL P DO
           CREATE process WORKER(i);
L1:     LOCK SYNC;      SYNC:=SYNC+1;      UNLOCK SYNC;
L2:     LOCK SYNC;
        IF SYNC != P THEN
        BEGIN
           UNLOCK SYNC;      GOTO L2;
        END
        ELSE BEGIN
           UNLOCK SYNC;
           WAIT:=0;
           BARRIER_2:="close";
           LOCK BARRIER_1;      BARRIER_1:="open";
           UNLOCK BARRIER_1;
        FOR m:=1 STEP 1 UNTIL N/sqrt(P) DO
           FOR n:=1 STEP 1 UNTIL N/sqrt(P) DO
           BEGIN
              IF A[m,n] != 0 THEN
              BEGIN
                 newindex:= MIN(neighbors of the B[m,n],
                    where A[m,n] != 0);
                 IF A[m,n] > newindex THEN
                 BEGIN
                    A[m,n]:=newindex;
                    MODIFY:="yes";
                 END
              END
```

```
            END
         LOCK WAIT;     WAIT:=WAIT+1;     UNLOCK WAIT;
L3:      LOCK WAIT;
         IF WAIT != P THEN
         BEGIN
            UNLOCK WAIT;     GOTO L3;
         END
         ELSE BEGIN
            UNLOCK WAIT;
            IF MODIFY = "yes" THEN
            BEGIN
               MODIFY:="no";
               SYNC:=0;
               BARRIER_1:="close";
               LOCK BARRIER_2;     BARRIER_2:="open";
               UNLOCK BARRIER_2;
               GOTO L1;
            END
            ELSE BEGIN
               LOCK DONE;     DONE:="yes";     UNLOCK DONE;
               LOCK BARRIER_2;     BARRIER_2:="open";
               UNLOCK BARRIER_2;
            END
   END

;
; Process WORKER(i)
;
   BEGIN
L1:      LOCK SYNC;     SYNC:=SYNC+1;     UNLOCK SYNC;
L2:      LOCK BARRIER_1;
         IF BARRIER_1 != "open" THEN
         BEGIN
            UNLOCK BARRIER_1;     GOTO L2;
         END
         ELSE UNLOCK BARRIER_1;
         FOR m:=([(i-1) mod sqrt(P)]*N/sqrt(P))+1 STEP 1
            UNTIL ([(i-1) mod sqrt(P)]+1)*N/sqrt(P) DO
            FOR n:={[(i-1)/sqrt(P)]*N/sqrt(P)}+1 STEP 1
               UNTIL {[(i-1)/sqrt(P)]+1}*N/sqrt(P) DO
            BEGIN
               IF A[m,n] != 0 THEN
               BEGIN
                  newindex:= MIN(neighbors of the B[m,n],
                     where A[m,n] != 0);
                  IF A[m,n] > newindex THEN
                  BEGIN
                     A[m,n]:=newindex;
                     MODIFY:="yes";
                  END
               END
            END
         LOCK WAIT;     WAIT:=WAIT+1;     UNLOCK WAIT;
L3:      LOCK BARRIER_2;
```

```
      IF BARRIER_2 != "open" THEN
      BEGIN
         UNLOCK BARRIER_2;     GOTO L3;
      END
      ELSE BEGIN
         UNLOCK BARRIER_2;
         LOCK DONE;
         IF DONE != "yes" THEN
         BEGIN
            UNLOCK DONE;     GOTO L1;
         END
         ELSE UNLOCK DONE;
      END
END
```

# C Values of Parameters

Table C.1: Values of parameters for the Jacobi iterative algorithm

| $P$ | $B$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| 4 | 1 | 2.007767 | 0.200631 | 1.000000 | 1.000000 |
| 4 | 2 | 2.015504 | 0.156849 | 1.054330 | 1.000000 |
| 4 | 4 | 2.022593 | 0.109776 | 1.113308 | 1.000000 |
| 4 | 8 | 2.027502 | 0.061006 | 1.174254 | 1.000000 |
| 4 | 16 | 2.029576 | 0.031288 | 1.211260 | 1.000000 |

Table C.2: Miss ratio and total penalty for the Jacobi iterative algorithm

| $P$ | $B$ | $Pr(M)_{sim}$ | $Pr(M)_{model}$ | $\lambda_{sim}$ | $\lambda_{model}$ |
|---|---|---|---|---|---|
| 4 | 1 | 0.006324 | 0.005197 | 0.015604 | 0.012434 |
| 4 | 2 | 0.004743 | 0.004035 | 0.013333 | 0.011712 |
| 4 | 4 | 0.005508 | 0.004895 | 0.020741 | 0.019122 |
| 4 | 8 | 0.005936 | 0.005608 | 0.033905 | 0.033046 |
| 4 | 16 | 0.006037 | 0.005993 | 0.058431 | 0.058993 |

Table C.3: Values of parameters for the S.O.R. iterative algorithm

| $P$ | $B$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| 4 | 1 | 2.007874 | 0.200000 | 1.200000 | 0.000000 |
| 4 | 2 | 2.015625 | 0.442285 | 2.650226 | 0.000000 |
| 4 | 4 | 2.023529 | 0.419091 | 3.490986 | 0.000000 |
| 4 | 8 | 2.027559 | 0.466962 | 6.195342 | 0.000000 |
| 4 | 16 | 2.027559 | 0.485815 | 12.062616 | 0.000000 |

Table C.4: Miss ratio and total penalty for the S.O.R. algorithm

| $P$ | $B$ | $Pr(M)_{sim}$ | $Pr(M)_{model}$ | $\lambda_{sim}$ | $\lambda_{model}$ |
|---|---|---|---|---|---|
| 4 | 1 | 0.005208 | 0.004323 | 0.013014 | 0.010774 |
| 4 | 2 | 0.005177 | 0.003650 | 0.015461 | 0.010912 |
| 4 | 4 | 0.006534 | 0.004643 | 0.026015 | 0.018473 |
| 4 | 8 | 0.008512 | 0.005840 | 0.050708 | 0.034856 |
| 4 | 16 | 0.009553 | 0.006432 | 0.094307 | 0.064005 |

Table C.5: Values of parameters for the quicksort

| $P$ | $B$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| 2 | 1 | 2.000000 | 0.848454 | 8.524792 | 0.000000 |
| 2 | 2 | 2.000000 | 0.927254 | 16.102846 | 0.005311 |
| 2 | 4 | 2.000000 | 0.911621 | 27.379687 | 0.035968 |
| 2 | 8 | 2.000000 | 0.749614 | 33.515399 | 0.161327 |
| 4 | 16 | 2.000000 | 0.527073 | 25.878085 | 0.419271 |
| 4 | 1 | 3.318430 | 0.813541 | 6.489117 | 0.000000 |
| 4 | 2 | 3.371045 | 0.912845 | 12.230106 | 0.005142 |
| 4 | 4 | 3.466228 | 0.905091 | 20.779997 | 0.035184 |
| 4 | 8 | 3.612498 | 0.746883 | 25.614908 | 0.158215 |
| 8 | 16 | 3.784725 | 0.528214 | 20.374663 | 0.410923 |
| 8 | 1 | 4.580748 | 0.796530 | 5.717353 | 0.000000 |
| 8 | 2 | 4.722295 | 0.904706 | 10.754544 | 0.005526 |
| 8 | 4 | 4.987490 | 0.902003 | 18.253418 | 0.034770 |
| 8 | 8 | 5.434978 | 0.753985 | 22.793013 | 0.151325 |
| 8 | 16 | 6.080918 | 0.538388 | 18.419391 | 0.393345 |

Table C.6: Miss ratio and total penalty for the quicksort

| $P$ | $B$ | $Pr(M)_{sim}$ | $Pr(M)_{model}$ | $\lambda_{sim}$ | $\lambda_{model}$ |
|---|---|---|---|---|---|
| 2 | 1 | 0.076779 | 0.053844 | 0.189578 | 0.134610 |
| 2 | 2 | 0.043321 | 0.029878 | 0.134391 | 0.089562 |
| 2 | 4 | 0.025626 | 0.017417 | 0.111185 | 0.069384 |
| 2 | 8 | 0.018372 | 0.012784 | 0.124287 | 0.075928 |
| 2 | 16 | 0.018343 | 0.013338 | 0.210876 | 0.131903 |
| 4 | 1 | 0.115593 | 0.100710 | 0.269707 | 0.239917 |
| 4 | 2 | 0.063170 | 0.055926 | 0.191843 | 0.164091 |
| 4 | 4 | 0.036758 | 0.033234 | 0.157567 | 0.129343 |
| 4 | 8 | 0.025725 | 0.025811 | 0.171496 | 0.143337 |
| 4 | 16 | 0.024854 | 0.029217 | 0.278288 | 0.250823 |
| 8 | 1 | 0.138924 | 0.129502 | 0.314974 | 0.300451 |
| 8 | 2 | 0.074812 | 0.071694 | 0.216137 | 0.207880 |
| 8 | 4 | 0.043398 | 0.042866 | 0.169846 | 0.165058 |
| 8 | 8 | 0.030225 | 0.033773 | 0.176905 | 0.183151 |
| 8 | 16 | 0.028873 | 0.039757 | 0.278211 | 0.324799 |

Table C.7: Values of parameters for the bitonic merge sort

| $P$ | $B$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| 2 | 1 | 2.000000 | 0.833081 | 80.011102 | 0.000000 |
| 2 | 2 | 2.000000 | 0.833113 | 160.022205 | 0.000000 |
| 2 | 4 | 2.000000 | 0.833138 | 320.044409 | 0.000000 |
| 2 | 8 | 2.000000 | 0.833171 | 640.088818 | 0.000000 |
| 4 | 16 | 2.000000 | 0.833268 | 1280.177637 | 0.000000 |
| 4 | 1 | 2.333333 | 0.785900 | 51.569922 | 0.000000 |
| 4 | 2 | 2.333333 | 0.785945 | 103.139844 | 0.000000 |
| 4 | 4 | 2.333333 | 0.785990 | 206.279687 | 0.000000 |
| 4 | 8 | 2.333333 | 0.786070 | 412.559375 | 0.000000 |
| 8 | 16 | 2.333333 | 0.786230 | 825.118750 | 0.000000 |
| 8 | 1 | 2.714286 | 0.750638 | 35.102570 | 0.000000 |
| 8 | 2 | 2.714286 | 0.750716 | 70.205139 | 0.000000 |
| 8 | 4 | 2.714286 | 0.750794 | 140.410278 | 0.000000 |
| 8 | 8 | 2.714286 | 0.750940 | 280.820557 | 0.000000 |
| 8 | 16 | 2.714286 | 0.751204 | 561.641113 | 0.000000 |

Table C.8: Miss ratio and total penalty for the bitonic merge sort

| $P$ | $B$ | $Pr(M)_{sim}$ | $Pr(M)_{model}$ | $\lambda_{sim}$ | $\lambda_{model}$ |
|---|---|---|---|---|---|
| 2 | 1 | 0.003124 | 0.002841 | 0.007809 | 0.007101 |
| 2 | 2 | 0.001562 | 0.001420 | 0.004685 | 0.004261 |
| 2 | 4 | 0.000781 | 0.000710 | 0.003124 | 0.002841 |
| 2 | 8 | 0.000391 | 0.000355 | 0.002343 | 0.002131 |
| 2 | 16 | 0.000196 | 0.000178 | 0.001955 | 0.001776 |
| 4 | 1 | 0.009376 | 0.009707 | 0.021880 | 0.023826 |
| 4 | 2 | 0.004688 | 0.004854 | 0.013155 | 0.014304 |
| 4 | 4 | 0.002344 | 0.002427 | 0.008792 | 0.009542 |
| 4 | 8 | 0.001173 | 0.001214 | 0.006612 | 0.007162 |
| 4 | 16 | 0.000586 | 0.000607 | 0.005519 | 0.005972 |
| 8 | 1 | 0.018494 | 0.020333 | 0.041175 | 0.048480 |
| 8 | 2 | 0.009247 | 0.010167 | 0.024792 | 0.029130 |
| 8 | 4 | 0.004623 | 0.005084 | 0.016597 | 0.019455 |
| 8 | 8 | 0.002312 | 0.002542 | 0.012506 | 0.014618 |
| 8 | 16 | 0.001156 | 0.001271 | 0.010457 | 0.012201 |

Table C.9: Values of parameters for the non-shuffling FFT

| $P$ | $B$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| 2 | 1 | 2.000000 | 0.250000 | 12.000000 | 0..500000 |
| 2 | 2 | 2.000000 | 0.166667 | 16.000000 | 0..500000 |
| 2 | 4 | 2.000000 | 0.100000 | 19.200000 | 0..500000 |
| 2 | 8 | 2.000000 | 0.055556 | 21.333333 | 0..500000 |
| 4 | 16 | 3.000000 | 0.029412 | 22.588235 | 0..500000 |
| 4 | 1 | 3.000000 | 0.285714 | 6.857143 | 0.666667 |
| 4 | 2 | 3.000000 | 0.181818 | 8.727273 | 0.666667 |
| 4 | 4 | 3.000000 | 0.105263 | 10.105263 | 0.666667 |
| 4 | 8 | 3.000000 | 0.057143 | 10.971429 | 0.666667 |
| 8 | 16 | 3.000000 | 0.029851 | 11.462687 | 0.666667 |
| 8 | 1 | 4.000000 | 0.300000 | 4.800000 | 0.750000 |
| 8 | 2 | 4.000000 | 0.187500 | 6.000000 | 0.750000 |
| 8 | 4 | 4.000000 | 0.107143 | 6.857143 | 0.750000 |
| 8 | 8 | 4.000000 | 0.057692 | 7.384615 | 0.750000 |
| 8 | 16 | 4.000000 | 0.030000 | 7.680000 | 0.750000 |

Table C.10: Miss ratio and total penalty for the non-shuffling FFT

| $P$ | $B$ | $Pr(M)_{sim}$ | $Pr(M)_{model}$ | $\lambda_{sim}$ | $\lambda_{model}$ |
|---|---|---|---|---|---|
| 2 | 1 | 0.020833 | 0.016667 | 0.052083 | 0.040625 |
| 2 | 2 | 0.010417 | 0.008929 | 0.031251 | 0.026414 |
| 2 | 4 | 0.005208 | 0.004735 | 0.020832 | 0.018821 |
| 2 | 8 | 0.002604 | 0.002467 | 0.015624 | 0.014768 |
| 2 | 16 | 0.001302 | 0.001265 | 0.013020 | 0.012640 |
| 4 | 1 | 0.041667 | 0.053030 | 0.104168 | 0.104246 |
| 4 | 2 | 0.020833 | 0.030556 | 0.062499 | 0.070457 |
| 4 | 4 | 0.010417 | 0.017210 | 0.041668 | 0.052036 |
| 4 | 8 | 0.005208 | 0.009348 | 0.031248 | 0.042068 |
| 4 | 16 | 0.002604 | 0.004915 | 0.026040 | 0.036791 |
| 8 | 1 | 0.062500 | 0.098684 | 0.156250 | 0.177519 |
| 8 | 2 | 0.031250 | 0.060000 | 0.093750 | 0.124403 |
| 8 | 4 | 0.015625 | 0.035473 | 0.062500 | 0.095416 |
| 8 | 8 | 0.007812 | 0.019979 | 0.046872 | 0.079689 |
| 8 | 16 | 0.003906 | 0.010751 | 0.039060 | 0.071329 |

Table C.11: Values of parameters for the shuffling FFT

| $P$ | $B$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| 2 | 1 | 2.000000 | 0.600000 | 18.799999 | 0.666667 |
| 2 | 2 | 2.000000 | 0.600000 | 37.599998 | 0.666667 |
| 2 | 4 | 2.000000 | 0.600000 | 75.199997 | 0.666667 |
| 2 | 8 | 2.000000 | 0.600000 | 150.399994 | 0.666667 |
| 4 | 16 | 2.000000 | 0.600000 | 300.799988 | 0.666667 |
| 4 | 1 | 2.000000 | 0.600000 | 17.600000 | 0.666667 |
| 4 | 2 | 2.000000 | 0.600000 | 35.200000 | 0.666667 |
| 4 | 4 | 2.000000 | 0.600000 | 70.400000 | 0.666667 |
| 4 | 8 | 2.000000 | 0.600000 | 140.800000 | 0.666667 |
| 8 | 16 | 2.000000 | 0.600000 | 281.600000 | 0.666667 |
| 8 | 1 | 2.000000 | 0.600000 | 16.400000 | 0.666667 |
| 8 | 2 | 2.000000 | 0.600000 | 32.799999 | 0.666667 |
| 8 | 4 | 2.000000 | 0.600000 | 65.600000 | 0.666667 |
| 8 | 8 | 2.000000 | 0.600000 | 131.199997 | 0.666667 |
| 8 | 16 | 2.000000 | 0.600000 | 262.399994 | 0.666667 |

Table C.12: Miss ratio and total penalty for the shuffling FFT

| $P$ | $B$ | $Pr(M)_{sim}$ | $Pr(M)_{model}$ | $\lambda_{sim}$ | $\lambda_{model}$ |
|---|---|---|---|---|---|
| 2 | 1 | 0.010639 | 0.009973 | 0.026596 | 0.022939 |
| 2 | 2 | 0.005319 | 0.004987 | 0.015957 | 0.013963 |
| 2 | 4 | 0.002659 | 0.002493 | 0.010638 | 0.009475 |
| 2 | 8 | 0.001330 | 0.001247 | 0.007980 | 0.007231 |
| 2 | 16 | 0.000665 | 0.000623 | 0.006650 | 0.006109 |
| 4 | 1 | 0.017045 | 0.015980 | 0.042613 | 0.036754 |
| 4 | 2 | 0.008523 | 0.007990 | 0.025569 | 0.022372 |
| 4 | 4 | 0.004262 | 0.003995 | 0.017046 | 0.015181 |
| 4 | 8 | 0.002131 | 0.001998 | 0.012785 | 0.011586 |
| 4 | 16 | 0.001065 | 0.000999 | 0.010650 | 0.009788 |
| 8 | 1 | 0.021341 | 0.020008 | 0.053353 | 0.046018 |
| 8 | 2 | 0.010671 | 0.010004 | 0.032012 | 0.028011 |
| 8 | 4 | 0.005336 | 0.005002 | 0.021343 | 0.019007 |
| 8 | 8 | 0.002668 | 0.002501 | 0.016007 | 0.014506 |
| 8 | 16 | 0.001334 | 0.001250 | 0.013335 | 0.012255 |

Table C.13: Values of parameters for the shortest path (array D)

| $P$ | $B$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| 2 | 1 | 2.000000 | 0.016015 | 3.131582 | 0.007598 |
| 2 | 2 | 2.000000 | 0.018783 | 3.673498 | 0.099374 |
| 2 | 4 | 2.000000 | 0.018888 | 3.687437 | 0.160561 |
| 2 | 8 | 2.000000 | 0.016465 | 3.225471 | 0.206814 |
| 4 | 16 | 2.000000 | 0.013757 | 2.656094 | 0.252902 |
| 4 | 1 | 4.000000 | 0.011215 | 2.075626 | 0.012947 |
| 4 | 2 | 4.000000 | 0.012127 | 2.238288 | 0.134514 |
| 4 | 4 | 4.000000 | 0.011509 | 2.110080 | 0.226021 |
| 4 | 8 | 4.000000 | 0.009945 | 1.815390 | 0.298764 |
| 8 | 16 | 4.000000 | 0.008347 | 1.514768 | 0.378907 |
| 8 | 1 | 8.000000 | 0.009636 | 1.764406 | 0.020354 |
| 8 | 2 | 8.000000 | 0.009551 | 1.738429 | 0.186201 |
| 8 | 4 | 8.000000 | 0.008700 | 1.578921 | 0.317634 |
| 8 | 8 | 8.000000 | 0.007580 | 1.371178 | 0.420822 |
| 8 | 16 | 8.000000 | 0.006690 | 1.206036 | 0.524920 |

Table C.14: Miss ratio and total penalty for the shortest path (array D)

| $P$ | $B$ | $Pr(M)_{sim}$ | $Pr(M)_{model}$ | $\lambda_{sim}$ | $\lambda_{model}$ |
|---|---|---|---|---|---|
| 2 | 1 | 0.000425 | 0.000418 | 0.001035 | 0.001046 |
| 2 | 2 | 0.000425 | 0.000417 | 0.001248 | 0.001251 |
| 2 | 4 | 0.000426 | 0.000418 | 0.001675 | 0.001670 |
| 2 | 8 | 0.000424 | 0.000417 | 0.002516 | 0.002503 |
| 2 | 16 | 0.000431 | 0.000425 | 0.004277 | 0.004245 |
| 4 | 1 | 0.001080 | 0.001255 | 0.001702 | 0.001902 |
| 4 | 2 | 0.001021 | 0.001256 | 0.002008 | 0.002325 |
| 4 | 4 | 0.000979 | 0.001266 | 0.002668 | 0.003194 |
| 4 | 8 | 0.000963 | 0.001278 | 0.004044 | 0.004933 |
| 4 | 16 | 0.000979 | 0.001291 | 0.006939 | 0.008442 |
| 8 | 1 | 0.002381 | 0.003193 | 0.003082 | 0.003922 |
| 8 | 2 | 0.002166 | 0.003214 | 0.003533 | 0.004873 |
| 8 | 4 | 0.002015 | 0.003241 | 0.004598 | 0.006775 |
| 8 | 8 | 0.001944 | 0.003276 | 0.006913 | 0.010608 |
| 8 | 16 | 0.001970 | 0.003307 | 0.011919 | 0.018299 |

Table C.15: Values of parameters for the shortest path (Lock)

| $P$ | $B$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| 2 | 1 | 2.000000 | 0.694307 | 6.268677 | 0.026948 |
| 2 | 2 | 2.000000 | 0.631367 | 7.629046 | 0.089268 |
| 2 | 4 | 2.000000 | 0.588347 | 8.254337 | 0.173938 |
| 2 | 8 | 2.000000 | 0.573504 | 7.975233 | 0.243362 |
| 4 | 16 | 2.000000 | 0.569084 | 6.804678 | 0.300680 |
| 4 | 1 | 4.000000 | 0.603533 | 3.679258 | 0.038105 |
| 4 | 2 | 4.000000 | 0.553122 | 4.110187 | 0.133794 |
| 4 | 4 | 4.000000 | 0.536362 | 4.126874 | 0.235190 |
| 4 | 8 | 4.000000 | 0.540971 | 3.828271 | 0.303074 |
| 8 | 16 | 4.000000 | 0.548917 | 3.248385 | 0.354967 |
| 8 | 1 | 8.000000 | 0.542311 | 2.805754 | 0.052579 |
| 8 | 2 | 8.000000 | 0.516249 | 2.910837 | 0.189223 |
| 8 | 4 | 8.000000 | 0.523668 | 2.750539 | 0.300325 |
| 8 | 8 | 8.000000 | 0.538540 | 2.489579 | 0.362667 |
| 8 | 16 | 8.000000 | 0.547287 | 2.135066 | 0.406702 |

Table C.16: Miss ratio and total penalty for shortest path (Lock)

| $P$ | $B$ | $Pr(M)_{sim}$ | $Pr(M)_{model}$ | $\lambda_{sim}$ | $\lambda_{model}$ |
|---|---|---|---|---|---|
| 2 | 1 | 0.024193 | 0.013939 | 0.060483 | 0.034717 |
| 2 | 2 | 0.018077 | 0.010817 | 0.053689 | 0.032146 |
| 2 | 4 | 0.015558 | 0.009569 | 0.061158 | 0.037785 |
| 2 | 8 | 0.015693 | 0.009745 | 0.092611 | 0.057788 |
| 2 | 16 | 0.018248 | 0.011365 | 0.180312 | 0.112677 |
| 4 | 1 | 0.034244 | 0.036141 | 0.084935 | 0.078099 |
| 4 | 2 | 0.028415 | 0.031336 | 0.082840 | 0.079345 |
| 4 | 4 | 0.027700 | 0.030847 | 0.106379 | 0.103752 |
| 4 | 8 | 0.030415 | 0.033362 | 0.174981 | 0.170032 |
| 4 | 16 | 0.037088 | 0.039535 | 0.354557 | 0.339716 |
| 8 | 1 | 0.043884 | 0.063697 | 0.108161 | 0.123877 |
| 8 | 2 | 0.041292 | 0.060758 | 0.117967 | 0.139392 |
| 8 | 4 | 0.045130 | 0.064497 | 0.169104 | 0.199717 |
| 8 | 8 | 0.052456 | 0.071682 | 0.293467 | 0.340097 |
| 8 | 16 | 0.064764 | 0.083865 | 0.597574 | 0.674214 |

Table C.17: Values of parameters for the image component labeling

| $P$ | $B$ | $J$ | $W$ | $l$ | $f$ |
|---|---|---|---|---|---|
| 4 | 1 | 2.003205 | 0.048688 | 2.085569 | 0.005083 |
| 4 | 2 | 2.005219 | 0.047128 | 2.861882 | 0.007692 |
| 4 | 4 | 2.006617 | 0.048858 | 3.986631 | 0.012630 |
| 4 | 8 | 2.007839 | 0.055315 | 7.046703 | 0.019564 |
| 4 | 16 | 2.009751 | 0.063706 | 12.355282 | 0.037664 |

Table C.18: Miss ratio and total penalty for the image component labeling

| $P$ | $B$ | $Pr(M)_{sim}$ | $Pr(M)_{model}$ | $\lambda_{sim}$ | $\lambda_{model}$ |
|---|---|---|---|---|---|
| 4 | 1 | 0.000225 | 0.000198 | 0.000528 | 0.000493 |
| 4 | 2 | 0.000199 | 0.000176 | 0.000564 | 0.000526 |
| 4 | 4 | 0.000177 | 0.000167 | 0.000673 | 0.000666 |
| 4 | 8 | 0.000180 | 0.000175 | 0.001041 | 0.001044 |
| 4 | 16 | 0.000197 | 0.000195 | 0.001907 | 0.001945 |

# D Finite Cache Results

Table D.1: Values of $r$, miss ratio and total penalty for the Jacobi iterative

| $P$ | $B$ | $C_s$ | $r$ | $\frac{Pr(M)_{sim}}{q_s}$ | $\frac{Pr(M)_{model}}{q_s}$ | $\frac{\lambda_{sim}}{q_s}$ | $\frac{\lambda_{model}}{q_s}$ |
|---|---|---|---|---|---|---|---|
| 4 | 4 | 256 | 0.373044 | 0.029263 | 0.035212 | 0.102136 | 0.136582 |
| 4 | 4 | 512 | 0.276659 | 0.033382 | 0.041904 | 0.111110 | 0.161249 |
| 4 | 4 | 1024 | 0.166079 | 0.051470 | 0.052910 | 0.169732 | 0.201882 |
| 4 | 4 | 2048 | 0.166033 | 0.051516 | 0.052915 | 0.169812 | 0.201901 |
| 4 | 4 | 4096 | 0.165943 | 0.051606 | 0.052926 | 0.169969 | 0.201941 |
| 4 | 4 | 8192 | 0.103180 | 0.073448 | 0.061519 | 0.254595 | 0.234158 |
| 4 | 4 | 16384 | 0.079119 | 0.079203 | 0.065393 | 0.288753 | 0.248963 |
| 4 | 4 | 32768 | 0.000000 | 0.103658 | 0.101641 | 0.385115 | 0.347415 |
| 4 | 4 | 65536 | 0.000000 | 0.103658 | 0.101641 | 0.385115 | 0.347415 |
| 4 | 8 | 256 | 0.534464 | 0.017528 | 0.008764 | 0.096435 | 0.051991 |
| 4 | 8 | 512 | 0.323938 | 0.018499 | 0.013165 | 0.099506 | 0.076829 |
| 4 | 8 | 1024 | 0.081982 | 0.026599 | 0.028567 | 0.141483 | 0.162722 |
| 4 | 8 | 2048 | 0.081960 | 0.026621 | 0.028569 | 0.141543 | 0.162736 |
| 4 | 8 | 4096 | 0.081916 | 0.026665 | 0.028575 | 0.141664 | 0.162768 |
| 4 | 8 | 8192 | 0.061682 | 0.033715 | 0.031285 | 0.182557 | 0.178327 |
| 4 | 8 | 16384 | 0.053518 | 0.036043 | 0.032493 | 0.199818 | 0.185367 |
| 4 | 8 | 32768 | 0.000000 | 0.053672 | 0.051612 | 0.303952 | 0.272178 |
| 4 | 8 | 65536 | 0.000000 | 0.053672 | 0.051612 | 0.303952 | 0.272178 |

Table D.2: Values of $r$, miss ratio and total penalty for the S.O.R. iterative

| $P$ | $B$ | $C_s$ | $r$ | $\frac{Pr(M)_{sim}}{q_s}$ | $\frac{Pr(M)_{model}}{q_s}$ | $\frac{\lambda_{sim}}{q_s}$ | $\frac{\lambda_{model}}{q_s}$ |
|---|---|---|---|---|---|---|---|
| 4 | 4 | 256 | 0.285201 | 0.048155 | 0.033609 | 0.171179 | 0.135314 |
| 4 | 4 | 512 | 0.156088 | 0.060682 | 0.046573 | 0.241339 | 0.185658 |
| 4 | 4 | 1024 | 0.156032 | 0.060738 | 0.046581 | 0.241437 | 0.185688 |
| 4 | 4 | 2048 | 0.155936 | 0.060834 | 0.046594 | 0.241605 | 0.185738 |
| 4 | 4 | 4096 | 0.133963 | 0.061122 | 0.049762 | 0.270970 | 0.198043 |
| 4 | 4 | 8192 | 0.051851 | 0.096875 | 0.065838 | 0.392846 | 0.261147 |
| 4 | 4 | 16384 | 0.000000 | 0.122148 | 0.118930 | 0.482965 | 0.389375 |
| 4 | 4 | 32768 | 0.000000 | 0.122148 | 0.118930 | 0.482965 | 0.389375 |
| 4 | 4 | 65536 | 0.000000 | 0.122148 | 0.118930 | 0.482965 | 0.389375 |
| 4 | 8 | 256 | 0.274955 | 0.034559 | 0.015707 | 0.200141 | 0.094280 |
| 4 | 8 | 512 | 0.078183 | 0.055057 | 0.032122 | 0.330902 | 0.189748 |
| 4 | 8 | 1024 | 0.078159 | 0.055081 | 0.032126 | 0.330968 | 0.189772 |
| 4 | 8 | 2048 | 0.078118 | 0.055122 | 0.032132 | 0.331081 | 0.189807 |
| 4 | 8 | 4096 | 0.072758 | 0.055233 | 0.033019 | 0.341863 | 0.195010 |
| 4 | 8 | 8192 | 0.014730 | 0.069466 | 0.046136 | 0.418870 | 0.274110 |
| 4 | 8 | 16384 | 0.000000 | 0.076767 | 0.073405 | 0.455303 | 0.366125 |
| 4 | 8 | 32768 | 0.000000 | 0.076767 | 0.073405 | 0.455303 | 0.366125 |
| 4 | 8 | 65536 | 0.000000 | 0.076767 | 0.073405 | 0.455303 | 0.366125 |

Table D.3: Values of $r$, miss ratio and total penalty for the quicksort

| $P$ | $B$ | $C_s$ | $r$ | $\frac{Pr(M)_{sim}}{q_s}$ | $\frac{Pr(M)_{model}}{q_s}$ | $\frac{\lambda_{sim}}{q_s}$ | $\frac{\lambda_{model}}{q_s}$ |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 256 | 0.064126 | 0.008623 | 0.008929 | 0.048488 | 0.051197 |
| 4 | 8 | 512 | 0.054876 | 0.008985 | 0.009901 | 0.050366 | 0.056569 |
| 4 | 8 | 1024 | 0.047778 | 0.009526 | 0.010797 | 0.053127 | 0.061499 |
| 4 | 8 | 2048 | 0.041435 | 0.010530 | 0.011742 | 0.058291 | 0.066676 |
| 4 | 8 | 4096 | 0.032969 | 0.013296 | 0.013277 | 0.073478 | 0.075064 |
| 4 | 8 | 8192 | 0.020986 | 0.019013 | 0.016227 | 0.105659 | 0.091122 |
| 4 | 8 | 16384 | 0.009400 | 0.025056 | 0.020470 | 0.138626 | 0.114257 |
| 4 | 8 | 32768 | 0.000000 | 0.030137 | 0.025595 | 0.163962 | 0.142677 |
| 4 | 8 | 65536 | 0.000000 | 0.030137 | 0.025595 | 0.163962 | 0.142677 |
| 8 | 8 | 256 | 0.062791 | 0.010924 | 0.012782 | 0.062160 | 0.071823 |
| 8 | 8 | 512 | 0.052433 | 0.012302 | 0.014325 | 0.069987 | 0.080088 |
| 8 | 8 | 1024 | 0.043255 | 0.014770 | 0.016022 | 0.084113 | 0.089126 |
| 8 | 8 | 2048 | 0.033563 | 0.018882 | 0.018279 | 0.107709 | 0.101084 |
| 8 | 8 | 4096 | 0.023735 | 0.023616 | 0.021259 | 0.134400 | 0.116803 |
| 8 | 8 | 8192 | 0.014626 | 0.027795 | 0.024921 | 0.156859 | 0.136096 |
| 8 | 8 | 16384 | 0.006465 | 0.031523 | 0.029266 | 0.175600 | 0.159109 |
| 8 | 8 | 32768 | 0.000000 | 0.034424 | 0.033700 | 0.188449 | 0.182930 |
| 8 | 8 | 65536 | 0.000000 | 0.034424 | 0.033700 | 0.188449 | 0.182930 |

Table D.4: Values of $r$, miss ratio and total penalty for the bitonic merge sort

| $P$ | $B$ | $C_s$ | $r$ | $\frac{Pr(M)_{sim}}{q_s}$ | $\frac{Pr(M)_{model}}{q_s}$ | $\frac{\lambda_{sim}}{q_s}$ | $\frac{\lambda_{model}}{q_s}$ |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 256 | 0.283192 | 0.000011 | 0.000009 | 0.000030 | 0.000055 |
| 4 | 8 | 512 | 0.228664 | 0.000021 | 0.000011 | 0.000058 | 0.000067 |
| 4 | 8 | 1024 | 0.181870 | 0.000043 | 0.000014 | 0.000118 | 0.000085 |
| 4 | 8 | 2048 | 0.142855 | 0.000087 | 0.000018 | 0.000239 | 0.000110 |
| 4 | 8 | 4096 | 0.111565 | 0.000260 | 0.000023 | 0.000938 | 0.000140 |
| 4 | 8 | 8192 | 0.027766 | 0.000433 | 0.000087 | 0.003550 | 0.000531 |
| 4 | 8 | 16384 | 0.008216 | 0.001213 | 0.000254 | 0.007278 | 0.001538 |
| 4 | 8 | 32768 | 0.000000 | 0.001559 | 0.001205 | 0.008792 | 0.007194 |
| 4 | 8 | 65536 | 0.000000 | 0.001559 | 0.001205 | 0.008792 | 0.007194 |
| 8 | 8 | 256 | 0.283323 | 0.000028 | 0.000021 | 0.000077 | 0.000127 |
| 8 | 8 | 512 | 0.228787 | 0.000056 | 0.000025 | 0.000154 | 0.000155 |
| 8 | 8 | 1024 | 0.181967 | 0.000110 | 0.000032 | 0.000303 | 0.000194 |
| 8 | 8 | 2048 | 0.142834 | 0.000297 | 0.000041 | 0.001333 | 0.000249 |
| 8 | 8 | 4096 | 0.052921 | 0.000705 | 0.000106 | 0.005251 | 0.000647 |
| 8 | 8 | 8192 | 0.026008 | 0.001709 | 0.000207 | 0.010149 | 0.001253 |
| 8 | 8 | 16384 | 0.008191 | 0.002340 | 0.000546 | 0.012976 | 0.003248 |
| 8 | 8 | 32768 | 0.000000 | 0.002636 | 0.002897 | 0.014253 | 0.013994 |
| 8 | 8 | 65536 | 0.000000 | 0.002636 | 0.002897 | 0.014253 | 0.013994 |

Table D.5: Values of $r$, miss ratio and total penalty for the non-shuffling FFT

| $P$ | $B$ | $C_s$ | $r$ | $\frac{Pr(M)_{sim}}{q_s}$ | $\frac{Pr(M)_{model}}{q_s}$ | $\frac{\lambda_{sim}}{q_s}$ | $\frac{\lambda_{model}}{q_s}$ |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 256 | 0.125000 | 0.000000 | 0.017702 | 0.000000 | 0.093432 |
| 4 | 8 | 512 | 0.125000 | 0.000000 | 0.017702 | 0.000000 | 0.093432 |
| 4 | 8 | 1024 | 0.125000 | 0.000000 | 0.017702 | 0.000000 | 0.093432 |
| 4 | 8 | 2048 | 0.125000 | 0.000000 | 0.017702 | 0.000000 | 0.093432 |
| 4 | 8 | 4096 | 0.125000 | 0.000000 | 0.017702 | 0.000000 | 0.093432 |
| 4 | 8 | 8192 | 0.125000 | 0.000000 | 0.017702 | 0.000000 | 0.093432 |
| 4 | 8 | 16384 | 0.062500 | 0.041667 | 0.026671 | 0.153648 | 0.136678 |
| 4 | 8 | 32768 | 0.031250 | 0.052083 | 0.034565 | 0.187500 | 0.174521 |
| 4 | 8 | 65536 | 0.000000 | 0.062500 | 0.046296 | 0.249999 | 0.233216 |
| 8 | 8 | 256 | 0.125000 | 0.000000 | 0.019585 | 0.000000 | 0.100196 |
| 8 | 8 | 512 | 0.125000 | 0.000000 | 0.019585 | 0.000000 | 0.100196 |
| 8 | 8 | 1024 | 0.125000 | 0.000000 | 0.019585 | 0.000000 | 0.100196 |
| 8 | 8 | 2048 | 0.125000 | 0.000000 | 0.019585 | 0.000000 | 0.100196 |
| 8 | 8 | 4096 | 0.125000 | 0.000000 | 0.019585 | 0.000000 | 0.100196 |
| 8 | 8 | 8192 | 0.062500 | 0.034722 | 0.030270 | 0.144097 | 0.149054 |
| 8 | 8 | 16384 | 0.041667 | 0.041666 | 0.036261 | 0.171439 | 0.175994 |
| 8 | 8 | 32768 | 0.020833 | 0.048611 | 0.044250 | 0.222657 | 0.212120 |
| 8 | 8 | 65536 | 0.000000 | 0.052083 | 0.054687 | 0.278646 | 0.261065 |

Table D.6: Values of $r$, miss ratio and total penalty for the shuffling FFT

| $P$ | $B$ | $C_s$ | $r$ | $\frac{Pr(M)_{sim}}{q_s}$ | $\frac{Pr(M)_{model}}{q_s}$ | $\frac{\lambda_{sim}}{q_s}$ | $\frac{\lambda_{model}}{q_s}$ |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 256 | 0.144864 | 0.000000 | 0.000101 | 0.000000 | 0.000603 |
| 4 | 8 | 512 | 0.144842 | 0.000000 | 0.000101 | 0.000000 | 0.000603 |
| 4 | 8 | 1024 | 0.144798 | 0.000000 | 0.000101 | 0.000000 | 0.000603 |
| 4 | 8 | 2048 | 0.144709 | 0.000000 | 0.000102 | 0.000000 | 0.000606 |
| 4 | 8 | 4096 | 0.144531 | 0.000000 | 0.000102 | 0.000000 | 0.000606 |
| 4 | 8 | 8192 | 0.141335 | 0.000000 | 0.000104 | 0.000000 | 0.000621 |
| 4 | 8 | 16384 | 0.023674 | 0.001894 | 0.000535 | 0.010417 | 0.003141 |
| 4 | 8 | 32768 | 0.021780 | 0.002841 | 0.002663 | 0.016099 | 0.009108 |
| 4 | 8 | 65536 | 0.000000 | 0.002841 | 0.002663 | 0.017046 | 0.015445 |
| 8 | 8 | 256 | 0.146294 | 0.000000 | 0.000115 | 0.000000 | 0.000690 |
| 8 | 8 | 512 | 0.146246 | 0.000000 | 0.000115 | 0.000000 | 0.000690 |
| 8 | 8 | 1024 | 0.146151 | 0.000000 | 0.000116 | 0.000000 | 0.000692 |
| 8 | 8 | 2048 | 0.145960 | 0.000000 | 0.000116 | 0.000000 | 0.000693 |
| 8 | 8 | 4096 | 0.142530 | 0.000000 | 0.000118 | 0.000000 | 0.000705 |
| 8 | 8 | 8192 | 0.025697 | 0.001742 | 0.000569 | 0.009581 | 0.003341 |
| 8 | 8 | 16384 | 0.023955 | 0.002613 | 0.000603 | 0.014807 | 0.003537 |
| 8 | 8 | 32768 | 0.023084 | 0.003049 | 0.002858 | 0.017423 | 0.009793 |
| 8 | 8 | 65536 | 0.000000 | 0.003049 | 0.002858 | 0.018294 | 0.016577 |

Table D.7: Values of $r$, miss ratio and total penalty for the image labeling

| $P$ | $B$ | $C_s$ | $r$ | $\frac{Pr(M)_{sim}}{q_s}$ | $\frac{Pr(M)_{model}}{q_s}$ | $\frac{\lambda_{sim}}{q_s}$ | $\frac{\lambda_{model}}{q_s}$ |
|---|---|---|---|---|---|---|---|
| 4 | 4 | 256 | 0.979180 | 0.001161 | 0.001518 | 0.005836 | 0.006418 |
| 4 | 4 | 512 | 0.978254 | 0.001598 | 0.001519 | 0.007779 | 0.006422 |
| 4 | 4 | 1024 | 0.978112 | 0.001598 | 0.001519 | 0.007779 | 0.006423 |
| 4 | 4 | 2048 | 0.977881 | 0.001598 | 0.001520 | 0.007779 | 0.006427 |
| 4 | 4 | 4096 | 0.976826 | 0.001598 | 0.001521 | 0.008029 | 0.006431 |
| 4 | 4 | 8192 | 0.973625 | 0.001859 | 0.001526 | 0.009997 | 0.006451 |
| 4 | 4 | 16384 | 0.971150 | 0.001878 | 0.001530 | 0.011190 | 0.006466 |
| 4 | 4 | 32768 | 0.000000 | 0.011504 | 0.011755 | 0.045703 | 0.046865 |
| 4 | 4 | 65536 | 0.000000 | 0.011504 | 0.011755 | 0.045703 | 0.046865 |
| 4 | 8 | 256 | 0.981803 | 0.000738 | 0.000560 | 0.005458 | 0.003505 |
| 4 | 8 | 512 | 0.981170 | 0.000880 | 0.000560 | 0.006521 | 0.003508 |
| 4 | 8 | 1024 | 0.981079 | 0.000880 | 0.000560 | 0.006521 | 0.003508 |
| 4 | 8 | 2048 | 0.980934 | 0.000880 | 0.000560 | 0.006521 | 0.003508 |
| 4 | 8 | 4096 | 0.980171 | 0.000880 | 0.000561 | 0.006722 | 0.003511 |
| 4 | 8 | 8192 | 0.977927 | 0.000902 | 0.000562 | 0.007631 | 0.003518 |
| 4 | 8 | 16384 | 0.976910 | 0.000908 | 0.000563 | 0.008129 | 0.003524 |
| 4 | 8 | 32768 | 0.000000 | 0.007244 | 0.007491 | 0.043306 | 0.044779 |
| 4 | 8 | 65536 | 0.000000 | 0.007244 | 0.007491 | 0.043306 | 0.044779 |