LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-497

ANALYZING MULTIPROCESSOR CACHE BEHAVIOR THROUGH DATA REFERENCE MODELING

Jory Tsai
Anant Agarwal

November 1993

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Analyzing Multiprocessor Cache Behavior Through Data Reference Modeling

Jory Tsai and Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

This paper develops a *data reference modeling* technique to estimate with high accuracy the cache miss ratio in cache-coherent multiprocessors. The technique involves analyzing the dynamic data referencing behavior of parallel algorithms. Data reference modeling first identifies different types of shared data blocks accessed during the execution of a parallel algorithm, then captures in a few parameters the cache behavior of each shared block as a function of the problem size, number of processors, and cache line size, and finally constructs an analytical expression for each algorithm to estimate the cache miss ratio. Because the number of processors, problem size, and cache line size are included as parameters, the expression for the cache miss ratio can be used to predict the performance of systems with different configurations. Six parallel algorithms are studied, and the analytical results compared against previously published simulation results, to establish the confidence level of the data reference modeling technique. It is found that the average prediction error for four out of six algorithms is within five percent and within ten percent for the other two. The paper also derives from the model several results on how cache miss rates scale with system size.

1 Introduction

An early phase in the design of multiprocessor systems is the definition of target applications to be run on the system along with potential hardware configurations. Of the various possible hardware configurations, the configuration that yields the best performance for a given cost is typically selected for the final system design specification. Because the cache is a critical determinant of multiprocessor performance, a simple analytical model of cache behavior that can rapidly yield cache miss rates for various parallel algorithms as a function of system and problem parameters, such as the number of processors and problem size, is extremely desirable during the early definition stage of the design process.

This paper develops a data reference modeling methodology to analyze parallel algorithms and obtain information that can be used to estimate the cache miss ratio in multiprocessor systems. Because the model captures the problem size, the system size, and the cache line size as parameters, it can be used to predict cache miss ratios for different system configurations. However, because the method is based on an analysis of parallel algorithms, it is not suitable for the analysis of complex or irregular applications, where the data reference patterns are hard to discern.

The data reference modeling technique consists of the following steps:

1. Identifying different types of shared data blocks accessed during the execution of a parallel algorithm by each processor. This technique is most convenient when the types of shared blocks accessed by each processor is the same for all processors – a behavior commonly exhibited by data parallel applications (i.e., applications in which each processor executes the same code but operates on different data sets).

2. Capturing in a few parameters the cache behavior of each shared block as a function of the problem size, number of processors, and cache line size. We assume that the partitioning strategy is an algorithm-specific property. The parameters essentially characterize a type of *processor locality* inherent in each type of sharing. Processor locality was described in [2] as the tendency of a processor to repeatedly access a given block of data before an access by a remote processor. A similar form of locality was also measured by Eggers [6] using the notion of write runs, and by Dubois and Wang [5] using

To appear in SIGMETRICS, 1993.

the notion of an access burst.

The three parameters used to capture the processor locality are the number of accesses, a, of a specific type of shared block by a given processor, the number of remote writes, w, to that block that result in cache misses suffered by the given processor, and the number of first-time fetches, f, of that block of data that are not preceded by a remote write. The parameter f contributes to the startup miss cost, and only affects the cache miss ratio in the first iteration of typical iterative algorithms. Therefore, f may be ignored when the algorithm executes many iterations. Notice that the ratio a/w yields a measure of the average number of uninterrupted accesses (i.e., a series of local accesses without any remote write) to a block of data following an eviction from a given processor's cache due to a remote write.

3. Constructing an analytical expression for each algorithm to estimate the cache miss ratio. Because the processor locality parameters w and a are expressed as a function of the number of processors, problem size, and cache line size, the expression for the cache miss ratio can be used to predict the performance of systems with different configurations.

This paper validates the model using six parallel algorithms. We find that the average prediction error for four out of six algorithms is within five percent and within ten percent for the other two. The paper also derives from the model several results on how cache miss rates scale with block size, number of processors and problem size.

This paper first discusses related work in Section 2. Section 3 develops the data reference modeling methodology to estimate a multiprocessor's cache miss ratio, and validates it using previously published simulation data for six parallel applications in Section 4. Results for two out of the six applications are discussed in detail. Section 5 uses the model to study the effects of problem size, cache line size, and number of processors on the cache miss ratio. The problem size is the size of the shared data structures indicated in the parallel algorithm. Section 6 concludes the paper.

2 Previous Work

Several previous studies directly relate to our current research: the independent reference model of Dubois and Briggs [4], the processor locality based model of Agarwal [1], the access burst model of Dubois and Wang [5], the write-run model of Eggers [6], and the directory model of Simoni and Horowitz [9].

To our knowledge, the model by Dubois and Briggs [4] was the earliest effort on analytically obtaining cache miss rates due to invalidation. They estimated miss ratios from a markov model assuming that every shared block was equally likely to be accessed by any processor in the system. Because references to shared memory typically display temporal locality in much the same manner as private references do, the predicted miss rates turn out to be very pessimistic [1].

More recently, researchers have begun using some measure of processor locality in their models. The locality based model of Agarwal used a measure of the processor locality derived from an address trace. Processor locality is derived from a measurement of the interval between references to a given block of shared data by a given processor and the interval between write references to that block of data by other processors. Cache miss rates for systems with other configurations are derived using a simple Markov model. The drawback with this approach is that while the cache miss rate predictions are accurate for the system size from which measurements were made, attempts to extrapolate cache behavior for other system sizes are unsuccessful. The reason for this difficulty lies in our inability to predict data reference patterns arising from algorithmic behavior from a single address trace. An examination of algorithmic behavior, on the other hand, directly reveals this information.

The access burst model is based upon the observation that global shared writable blocks are accessed largely in critical sections. Within a critical section, a block can be assumed to be accessed by a single processor without interruption from other processors. A burst is defined to be a duration of accesses by a processor to a global shared block in an uninterrupted manner. The access burst size is a measure of the processor locality of the shared block. The model also measures the probability that a block is modified during an access burst. The measurements are made for each block size and the problem size is fixed. The model assumes that after a burst, all processors sharing the block are equally likely to access it again, and that the access bursts are independent from one another. Using these assumptions and measurements, a Markov model representing the global state of a shared block is constructed to estimate the likelihood of occurrence of each type of cache event such as an invalidation. The states represent the number of processors sharing the block.

Simoni and Horowitz focused on modeling the performance of limited pointer directory schemes. Their analysis models processor locality by assuming that a *primary* processor is more likely to access a block of data than one of several *secondary* processors. The analysis assumes that the number of processors actively accessing a block of data is fixed. They chose 64 for this number.

Our approach is different from that of Simoni and Horowitz, and Dubois and Wang, in that we analyze the algorithms and derive expressions for a few parameters, as a function of problem size, number of processors and block

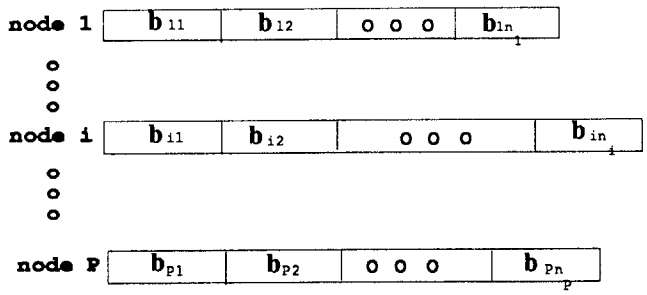

Figure 1: Typical distribution of shared blocks in the caches in a multiprocessor system.

3.2 Notation

On a P-node shared-memory multiprocessor system, shared data blocks are distributed among the caches in the nodes. Figure 1 shows a distribution of various types of shared blocks in the system. Let there be n_i different types of shared blocks in the cache in node i. Blocks with identical access patterns are said to have the same *type*. Miss rates of blocks with the same type can be captured using a single DRM, that is, using a single function of system and problem parameters. Let b_{ij} denote the j-th type of shared block on node i. The types of shared blocks on node i are named b_{i1}, $b_{i2}, \cdots, b_{in_i}$. A DRM is used to present the state of each b_{ij}.

We first introduce the notation for parameters derived from an algorithm. The number of remote-writes on b_{ij} that induce cache misses is denoted w_{ij}, and the number of first references of b_{ij} by processor i that are not preceded by a remote write (i.e., w_o) event is denoted f_{ij}. Let the number of accesses of b_{ij} be captured by a_{ij}. Finally, we will use the variable s_{ij} to denote the probability of accessing b_{ij} on node i; note that s_{ij} is simply $a_{ij} / \sum_j a_{ij}$.

The following are the computed quantities. Let the miss ratio of b_{ij} be called m_{ij}. The miss ratio of a type of shared block is simply the fraction of references of that type of block that results in a miss. Misses are caused both by remote writes followed by local accesses and due to first-time fetches.

Table 1 summarizes the notation used in this paper.

3.3 Processor Cache Miss Ratio

A processor's cache miss ratio is derived from all DRMs on that processor. After all the types of DRMs are identified on the processor, we can determine the n_i, b_{ij}, and w_{ij} from the data partition of an algorithm and the accesses sequence of shared data during the execution. Then, the miss ratios for each b_{ij} is the ratio of the sum of the number of first time misses and the write-invalidate induced misses and the number of accesses of the block.

P	number of processors
N	problem size
B	cache block size
i	index of the node count, $i = 1, 2, \ldots, P$
j	index of the types of shared blocks on node i
n_i	number of types of shared blocks on node i
b_{ij}	jth type of shared block on node i
s_{ij}	probability of accessing b_{ij} on node i
f_{ij}	number of first-time fetches of b_{ij}
a_{ij}	number of accesses of b_{ij}
w_{ij}	number of remote writes that cause cache misses on b_{ij}
m_{ij}	miss ratio of b_{ij}
M_i	cache miss ratio of node i
M	cache miss ratio of system

Table 1: Notation

$$m_{ij} = \frac{f_{ij} + w_{ij}}{a_{ij}}$$

The parameters a_{ij}, w_{ij}, and f_{ij}, are algorithm dependent functions of P, N, and the block size B. A processor's miss ratio (M_i) can be derived from the sum of each b_{ij}'s cache miss ratio times the probability of accessing b_{ij}. That is,

$$M_i = \sum_{j=1}^{n_i} (s_{ij} \times m_{ij})$$

To calculate the average cache miss ratio M of all processors in the system, we use M_i and the processor utilization (U_i) of each node i. The average processor cache miss ratio is then:

$$M = \frac{\sum_{i=1}^{P} (M_i \times U_i)}{\sum_{i=1}^{P} U_i},$$

If all processors have the same utilization U, we can write,

$$M = \frac{\sum_{i=1}^{P} (M_i \times U)}{\sum_{i=1}^{P} U} = \frac{1}{P} \times \sum_{i=1}^{P} M_i$$

In our experimental analysis, we will assume that all processors have the same utilization, so that we can make the above simplification.

4 Applications and Validation

In this paper, we demonstrate the accuracy and relative ease of using the data reference modeling approach by comparing results from simulations and modeling. We will use

the applications studied by Dubois and Wang [5, 7] both because of the availability of simulation results on these applications and due to the simplicity of the implementations of the parallel algorithms. While Dubois and Wang used nine algorithms, we studied six due to the lack of detailed published information on the others. The six algorithms we study are: Jacobi iteration, successive over relaxation, dynamic parallel quicksort, non-shuffling FFT, shuffling FFT, shortest path, and image component labeling.

The above six algorithms can be further divided into two categories:

- Data independent algorithms: Jacobi, S.O.R., Shuffling FFT, and Non-shuffling FFT. Data independent algorithms are those in which the input data set does not affect the shared-data access patterns.
- Data dependent algorithms: Dynamic quicksort and Image component labeling. Data dependent algorithms are those in which the input data sets may affect the access patterns to shared blocks.

This paper presents a detailed analysis of one algorithm from each category: dynamic parallel quicksort and shuffling FFT. For details on the others see [8]. We hope to use these two types of algorithms to demonstrate different DRM analysis approaches.

The examples show that the number of types of shared blocks for many algorithms, especially for those that are iterative, are very few, so the analysis process of constructing the DRMs for each type of shared block is not onerous. Parallel iterative algorithms in which the iterations are distributed among the processors not only allow extracting the DRMs for shared data types on one processor, but also allow us to focus on one iteration of the algorithm.

By analyzing the application code, we identify all data blocks, both shared and private, accessed within each iteration. The number of references can also be determined from the application code for each iteration. Once the DRMs for an algorithm are identified, the cache miss ratio can be formulated as a function of the cache line size, the problem size, and the system size.

After demonstrating that the model has acceptable accuracy for two algorithms, we will focus on analyzing the effects of cache line size, problem size and system size on multiprocessor cache miss rates. Our validation experiments compare the miss ratios obtained through analysis with the simulation results reported by Dubois and Wang [5, 7].

4.1 Shuffling FFT Algorithm

The shuffling FFT algorithm evaluates the discrete fourier transform. Let $s(k),\ k = 0, 1, 2, ..., N-1$ be N samples of a time function. The discrete fourier transform of $s(K)$ is defined to be the discrete function $x(j), j = 0, 1, 2, ...N-1$, where

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

where $i = \sqrt{-1}$.

The *problem size* of the shuffling FFT algorithm is the N-item array $s(k)$; this array is usually divided into P equal sized chunks where P is the number of processors in the system. The implementation uses a single copy of an N-item array, called the *valid* array; the implementation also uses two copies of the data array for temporaries. A one-dimensional shuffling FFT algorithm for N data items is represented by a butterfly graph with $\log_2 N$ stages. These N data items are stored sequentially in the global memory. Each processor is responsible for computing the FFT on its chunk, which contains $\frac{N}{P}$ data items. In this algorithm, computations of partial FFTs alternate with shuffling phases where data are passed among processors. At most two processors can share a block at any time. Coherence activity is significantly reduced since data locality exists.

Figure 2 presents the shuffling FFT algorithm for $N = 16$ and $P = 4$. During the computation phase, each local processor owns $\frac{N}{P}$ data elements and computes in a butterfly fashion. Write operations always occur within the butterfly computation phase and never occur in the shuffling phase, and each shared block can be written by one processor only. Each block is shared by *at most* two processors when the cache line size $B << N/P$ precondition is held. If the cache line size is too large then data elements stored in a block may be accessed by more than two processors, this may cause additional write invalidations than when the precondition is held.

The algorithm requires a total of $2 \times \lceil \frac{\log_2 N}{\log_2 \frac{N}{P}} \rceil$ iterations; each iteration consists of a computation stage and a shuffling stage. Figure 2 shows data partitioning, synchronization, butterfly computation data flow, and shuffling directions. Each shared block is generally shared by only two processors, except the first $\frac{N}{2P}$ data elements on the first processor and the last $\frac{N}{2P}$ data elements on the last processor. The first and last processors processors are different because they have one neighboring processor, unlike others, which have two neighboring processors.

At the beginning of execution, all processors load their N/P data items into their respective local caches. During the computation phase of the algorithm, each data element has to be read and updated locally. Updated data elements are stored into the *valid* array, and synchronization takes place.

During the shuffling phase, the data elements in the *valid* array are copied into a local temporary array, then restored into relative locations of the *valid* array. The relative locations to which a data element must shuffle is dependent

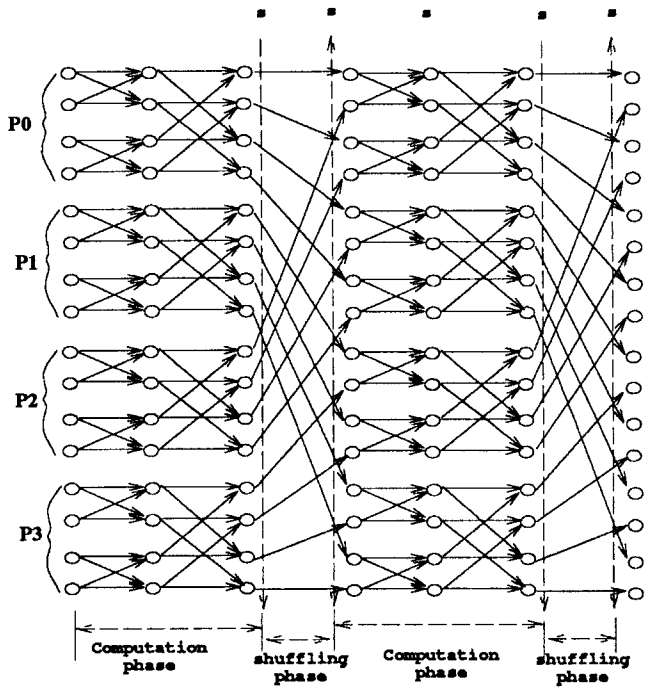

Figure 2: Shuffling FFT algorithm for P=4 and N=16.

on its processor's location. All data elements are virtually divided into two halves: the first half of data elements are shuffled towards the other half, and visa versa. Synchronization is denoted by the dotted line in the figure, and is required before each computation and shuffling phase.

All reads and writes happen locally and no cache misses occur during the computation phase. In the shuffling phase, the order of shuffling operations is remote read, local write, local read, and remote write. Cache misses only occur during the remote read of the shuffling phase. This indicates there is at most one cache miss per cache block, and $w_{ij} = 1$. The only exception is the $N/2P$ data elements mentioned in the first and the last processor, whose $w_{ij} = 0$.

4.1.1 Cache Miss Ratio Validation for Shuffling FFT

In order to calculate the cache miss ratio for the shuffling FFT algorithm one has only to focus on each iteration of both the computation and the shuffling stages. The behavior of each iteration is independent throughout the whole course of execution, which comprises $\lceil \frac{\log_2 N}{\log_2 \frac{N}{P}} \rceil$ stages.

We will first compute the miss ratio of the non-boundary processors, and then adjust the miss rate to account for the first and last processor's access patterns. Within the computation phase, $\log_2(N/P)$ butterfly stages are needed to compute $\frac{N}{P}$ data elements. Each processor performs only two reads and one write on each data element in the computation phase, and two reads and two writes in the shuffling phase. Therefore, the total number of accesses on b_{ij}, namely a_{ij}, during each stage is $(3 \cdot \log_2 \frac{N}{P} + 4) \cdot B$.

Cache misses happen only in the shuffling phase, where $w_{ij} = 1$. Therefore, for the non-boundary processors,

$$n_i = 1, \ \forall i = 2, 3, ..., P-1$$

$$w_{ij} = 1$$

and

$$s_{ij} = 1$$

Therefore,

$$m_{ij} = \frac{w_{ij}}{a_{ij}} = \frac{1}{B \cdot (3 \cdot \log_2 \frac{N}{P} + 4)}$$

And, the cache miss ratio for each processor is

$$M_i = \frac{1}{B} \cdot \frac{1}{(3 \cdot \log_2 \frac{N}{P} + 4)} \quad (1)$$

The effect of the first and the last processor can be included as follows. Recall that processor 1 and processor P have two distinct types of blocks: those that have $w_{ij} = 1$ and those that have $w_{ij} = 0$. Each processor has $N/2P$ blocks of each type. Therefore, the miss ratio of the first processor and the last processor is:

$$M_{1,P} = \frac{1}{2}\frac{1}{B} \cdot \frac{1}{(3 \cdot \log_2 \frac{N}{P} + 4)}$$

The average miss ratio for the whole system is given by,

$$M = \frac{P-2}{P} M_i + \frac{2}{P} M_{1,P}$$

Simplifying, we get,

$$M = \frac{P-1}{PB} \cdot \frac{1}{(3 \cdot \log_2 \frac{N}{P} + 4)}$$

Table 2 compares the cache miss ratios derived from this function and those obtained from simulation results published in [5, 7]. The cache miss ratios are collected based on cache line sizes of one, two, four, eight, and 16 words, number of processors varying from two through eight, and a problem size $N = 65536$. The comparisons are also shown in Figure 3. A constant percentage difference is found between simulated results and modeled results for each number of processors. The observed mean difference is 4.3 percent.

4.2 Parallel Dynamic Quicksort

Dynamic quicksort is a *divide-and-conquer* algorithm, which sorts an array A[1], A[2], ..., A[N] by rearranging it to make the condition that A[1],...,A[j-1] $\leq$ A[j] $\leq$

P	B	DRM Results	Simulation Results	Percent Difference
2	1	0.010204	0.010639	-4.08
2	2	0.005102	0.005319	-4.08
2	4	0.002551	0.002659	-4.06
2	8	0.001275	0.001330	-4.09
2	16	0.000638	0.000665	-4.09
4	1	0.016304	0.017045	-4.34
4	2	0.008152	0.008523	-4.35
4	4	0.004076	0.004262	-4.36
4	8	0.002038	0.002131	-4.36
4	16	0.001019	0.001065	-4.32
8	1	0.020349	0.021341	-4.65
8	2	0.010174	0.010671	-4.65
8	4	0.005087	0.005336	-4.66
8	8	0.002544	0.002668	-4.66
8	16	0.001272	0.001334	-4.66

Table 2: DRM versus simulation for shuffling FFT

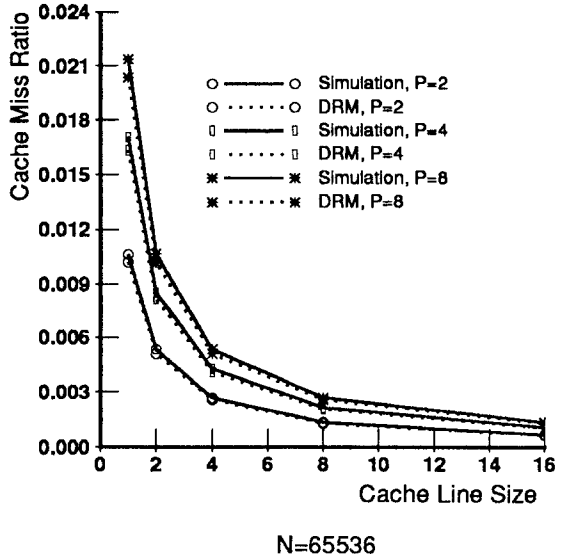

Figure 3: DRM versus simulation for shuffling FFT.

A[j+1],...,A[N] holds for some j, and a splitting process to place the two subarrays into a global job queue. By recursively applying the same procedure to the subarrays A[1],...,A[j-1] and A[j+1],...,A[N], the entire array is sorted.

In its parallel implementation, at the end of each splitting phase, the larger subarray is processed by the same processor and a descriptor of the smaller subarray is stored into a global job queue for potential distribution to other processors. The larger subarray is retained to maximize data locality. When a processor is idle it checks the global job queue and picks up a subarray's descriptor if the global job queue is not empty. If the size of the subarray is one then the processor writes the subarray back to shared array and change its state to idle. The algorithm is completed when all processors are idle and the job queue is empty.

The *problem size* of the dynamic quicksort algorithm is the N item array A. In this algorithm, every data item in this array may be accessed by all processors during the course of execution. The implementation of this algorithm uses a single copy of the array. There is no system wide synchronization required for this algorithm during run time. The only restriction on data sharing is that shared data are accessed mutually exclusively while a processor splits the array or the subarray.

Figure 4 shows how shared data elements are accessed during the execution of the algorithm, for $N = 32$ and $P = 4$. Each square represents a data element in the array. Squares grouped by a rectangle in the graph shows those data elements are allocated and sorted by a processor within that stage. The dotted line shows data locality after the initial loading. For simplicity, we assume that the subarrays are of equal size.

During run time, the first processor (say, P1) allocates all data elements initially, and releases less than half of those elements at the end of stage 1. The second processor (say, P2) then picks a subarray from the job queue and sorts the subarray, and other processors follow the same task. After stage 1, there are $(N-1)/2$ unsorted elements on each processor on average, and the N element array has been divided into 2 subarrays. After stage x, there are $(N - 2^{x-1}+1)/(2^{x-1})$ unsorted elements on each processor, and the array has been divided into 2^{x-1} subarrays. When the number of unsorted elements on each processor reaches one, the execution is completed. The average run time of the algorithm is $\log_2 N$ stages ($\lceil \log_2(N-1) \rceil$ iterations).

4.2.1 Cache Miss Ratio and Model Validation

Cache misses in the quicksort algorithm have a high degree of data dependency. That is, it is impossible to figure out exactly the number of cache misses without complete knowledge of the data being sorted. However, we can proceed to formulate a DRM for this type of algorithm by making suit-

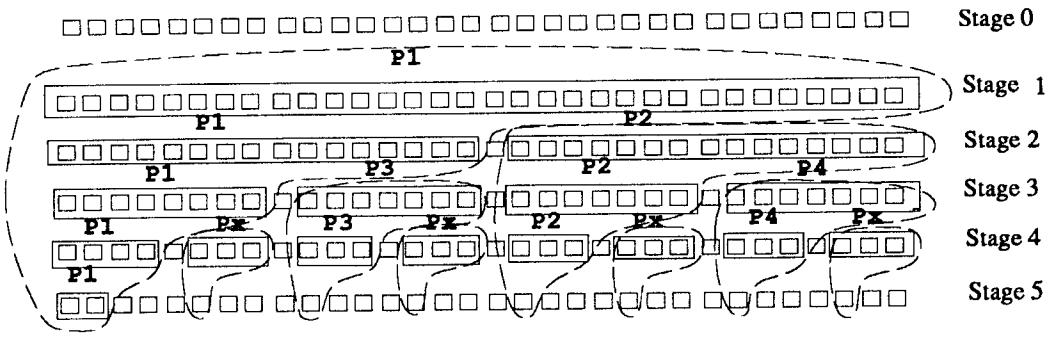

Figure 4: Shared data handling in dynamic quicksort, $P = 4$ and $N = 32$

able assumptions about the data distribution. For simplicity, we make the assumption that the array at each stage is evenly divided into subarrays, since we do not know the exact size of the two sorted subarrays. The assumption of even division represents the worst-case scenario; a 65-35 split is more likely if the numbers in the array are generated randomly. Under this assumption, we can derive DRM parameters and then estimate the cache miss ratio for the algorithm.

At stage x, the N element array has $2^{x-1} - 1$ sorted data elements, an average queue size of $2^{x-1} - P$, and 2^{x-1} subarrays. The average size of a unsorted subarray is therefore $\frac{N-2^{x-1}+1}{2^{x-1}}$ elements. To derive the cache miss ratio for a processor in this algorithm, since only one type of shared block exists, it is convenient to consider the whole course of the execution and derive expressions for f, a, and w for this type of shared block. Clearly, the value of n_i is one for all i, and the value of s is also one, since only one type of shared block exists.

Here, we can ignore the work queue's data blocks, since it has the same access pattern as the shared data blocks, and hence will not change the cache miss ratio. However, we need to taking the work queue into account if we are estimating the cache miss count,

There are three types of cache misses that occur during the execution of this algorithm:

- initial loading misses
- loading misses from the job queue, which correspond to cache misses that occur when a subarray is stored in the unsorted job queue and requested by a processor
- invalidation and false-sharing misses within a cache block on account of accesses by different processors.

The sum of the first two comprise the f component of misses, and the third type constitutes the w component.

Given a cache line size B, the number of initial loading misses is simply $\lceil N/B \rceil$.

The loading misses from the job queue as computed as follows. After $\log_2 P$ stages, all processors on the system become active and the job queue size remains empty, because an idle processor grabs a sorted subarray from the job queue as soon as a sorted subarray is inserted by an active processor. At stage x, there are a total of $2^{x-1} - P$ subarrays in the queue, with an average size of $(N - 2^{x-1} + 1)/(B \cdot 2^{x-1})$ blocks. The probability that a subarray is inserted and taken again from the queue is $1/P$. Consequently, the probability that reloading a subarray misses in the cache is $(P-1)/P$. Overall, from stage $\log_2 P + 2$ to stage $\log_2 N$, the number of such reloading misses is:

$$(\frac{P-1}{P}) \cdot (\frac{N - 2^{x-1} + 1}{B \cdot 2^{x-1}}) \cdot (2^{x-1} - P).$$

The sum of the above two components is f.

As the algorithm completes, if the size of a subarray is smaller than the cache line size, then write operations to a subarray may cause write invalidations on other subarrays within the same cache block. These misses caused by false sharing within a cache block usually occur when the size of subarrays is very small. A total of

$$(\frac{N}{B}) \cdot \sum_{y=1}^{\log_2 B+1} [(\frac{P-1}{P}) \cdot (2^{y-1} - 1)]$$

misses occur from stage $\log_2 N - (log_2 B + 1)$ to stage $\log_2 N$. These comprise the w component of misses.

An approximate total cache access count in stage x is $11 \cdot (N + 2^{x-1} - 1)/(2^x)$, which involves 7 shared data accesses and 4 local variable accesses for each data element in the subarray. The constant access of 11 may be vary depends on the implementation of algorithm. The sum of these values over all the stages, namely, $\sum_{x=1}^{\log_2 N} [11 \cdot \frac{N+2^{x-1}-1}{2^x}]$, is the value of a.

Since there is only one type of shared blocks in the system, $m_{ij} = M_i = M$. Thus M is simply $(f + w)$ divided by the total cache access count a. Therefore,

$$M_i = \frac{f + w}{a}$$

$$M_i = \frac{\frac{N}{B} + \sum_{x=\log_2 P+2}^{\log_2 N} [\frac{P-1}{P} \cdot \frac{N-2^{x-1}+1}{B \cdot 2^{x-1}} \cdot (2^{x-1} - P)] + \sum_{y=1}^{\log_2 B+1} [\frac{N}{B} \cdot \frac{P-1}{P} \cdot (2^{y-1}-1)]}{\sum_{x=1}^{\log_2 N} [11 \cdot \frac{N+2^{x-1}-1}{2^x}]}$$

which is simplified to yield,

$$M_i = \frac{\frac{N}{B} + \frac{P-1}{B \cdot P} \cdot [(N+P-1) \cdot \log_2 \frac{N}{B} + 4P - 1 + \frac{(N-1) \cdot 2P}{N}] + \frac{N \cdot (P-1)}{B \cdot P} \cdot (2B - \log_2 B - 2)}{11 \cdot [\frac{\log_2 N}{2} + \frac{(N-1)^2}{N}]}$$

Table 3: Expression for the miss rate in quicksort.

P	B	Simulation Results	DRM Results	Percent Difference
2	1	0.076779	0.090916	18.41
2	2	0.043321	0.048300	11.49
2	4	0.025626	0.028412	10.87
2	8	0.018372	0.019179	4.39
2	16	0.018343	0.014917	-18.67
4	1	0.115593	0.122175	5.69
4	2	0.063170	0.065350	3.45
4	4	0.036758	0.039069	6.28
4	8	0.025725	0.026993	4.93
4	16	0.024854	0.021489	-13.53
8	1	0.138924	0.130718	-5.90
8	2	0.074812	0.070332	-5.98
8	4	0.043398	0.042625	-1.77
8	8	0.030225	0.030015	-0.69
8	16	0.028873	0.024332	-15.72

Table 4: Simulation versus DRM result listing for quicksort

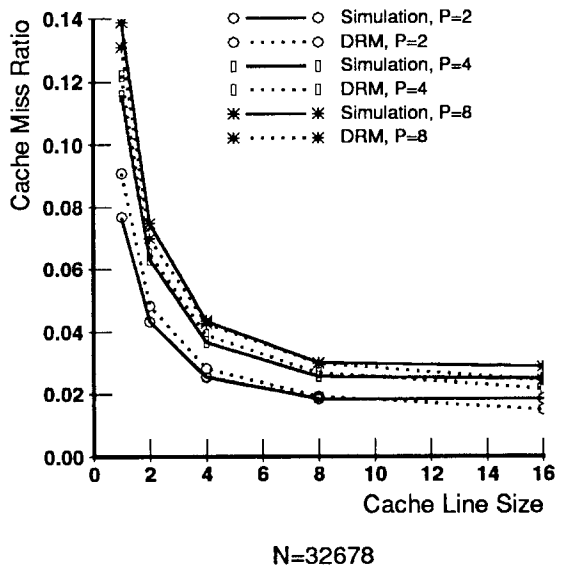

Figure 5: DRM versus simulation for quicksort.

and, substituting for f, w, and a, we obtain the miss rate as depicted in Table 3.

Table 4 compares the cache miss ratios derived from this function and the miss rates from simulations. Simulations use cache line sizes of one, two, four, eight, and 16 words, processor numbers of two, four, and eight, and a problem size of 32678. The comparisons are also shown in Figure 5. An average difference of 8.5 percent was found between simulated results and modeled results. We believe the difference is primarily due to the unpredictability of the input data set, as discussed in Section 4.2.1. Nevertheless, we find that the analysis is still reasonable for this type of algorithm with suitable assumptions about data distributions.

4.3 Summary of Results for Six Algorithms

Table 5 lists the differences between the analytically obtained miss ratios and the miss ratios obtained through simulations for the six algorithms studied. We observe that the data reference modeling approach is fairly accurate in estimating the cache miss ratios for all studied cases, and is generally significantly more accurate than the Access Burst Model [5] for the same set of benchmarks. For example, the data reference modeling approach yields average and maximum errors of 0.63 and 1.30 percent respectively for Jacobi, while the Access Burst Model yields average and maximum errors of 10.03 and 17.82 percent respectively.

The average is the absolute difference divided by the simulation data, abs(model-sim)/sim. Maximum is the absolute maximum difference.

Since our approach essentially does a mental simulation of algorithms, why are the differences not zero? There are several reasons for the mismatch between simulations and analysis. In algorithms where the cache miss ratios is data dependent (quicksort and image labeling), the errors are relatively larger than the others because of the lack of fealty in the assumptions about data distributions. In the other algorithms, the differences are much smaller; we believe these can be attributed to our incomplete knowledge of the specific implementations of the algorithms.

5 Predicting The Cache Miss Ratio

Armed with the knowledge that the data reference modeling method is acceptably accurate in predicting cache miss ratios for several system configurations, we can now apply the

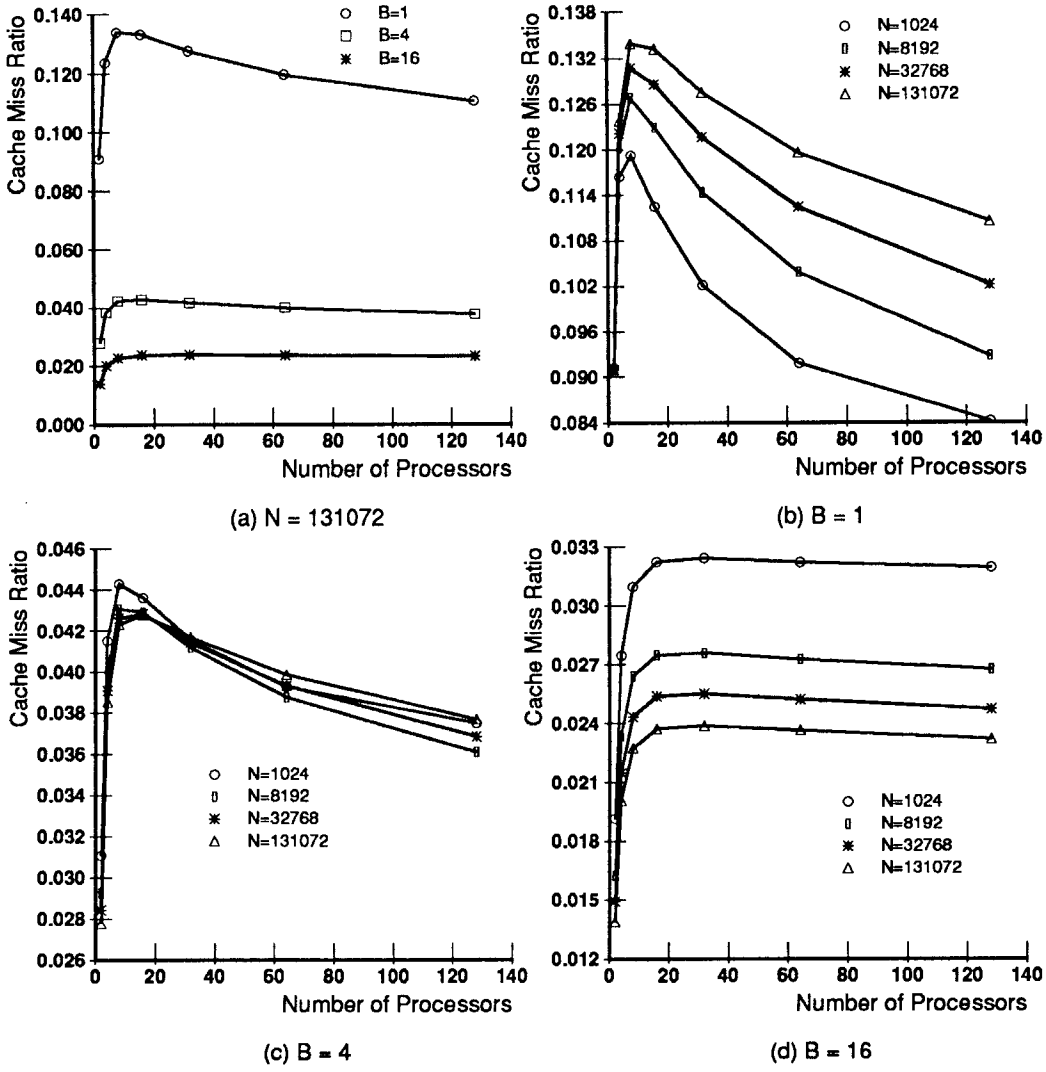

Figure 8: Cache miss ratios for quicksort.

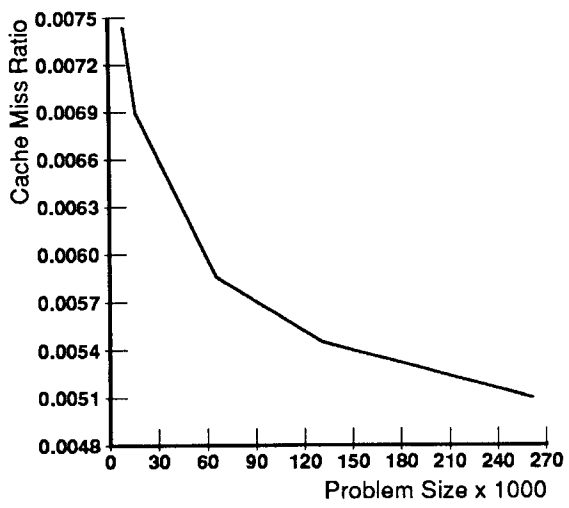

Figure 7: Cache miss ratio for shuffling FFT, B=4 and P=16.

One of the most interesting features of these graphs is the relationship between cache miss ratio and the problem size for a given cache line sizes. Figure 8(b), corresponding to B = 1, shows that when the problem size increases the cache miss ratio also increases. Figure 8(c), corresponding to B = 4, shows that the cache miss ratio is largely insensitive to the problem size. Figure 8(d), corresponding to B = 16, shows that when the problem size increases the cache miss ratio decreases.

The reason for this behavior is that a larger cache line size (greater than four) reduces the cache miss ratio significantly during the early stages of algorithm execution, when the first-time miss cost is most significant. On the other hand, a larger cache line size pays a higher cache miss penalty toward the end of the execution, when coherence related invalidations abound. Notice that for the B=16 case, the flat miss ratio curves indicate that the cache miss ratio is less sensitive to the number of processors than when the cache line size is large. These results suggests that a cache line size greater than or equal to four is preferable to smaller line sizes.

6 Conclusions

This paper presented a data reference modeling approach to computing the cache miss ratios of multiprocessors for different algorithms. The method involves a mental simulation of the algorithm. The approach is validated by comparing analytically obtained results with those from simulations.

The modeling approach analyzes algorithms to derive a few parameters that capture the processor locality in applications. These parameters are expressed as a function of the problem size, the number of processors, and the cache line size. By analyzing the algorithm, we can accurately capture the impact of the system configuration and problem size on the cache miss ratio. This approach is different from methods developed by others in that it does not analyze an address trace. We found that using parameters measured from address traces alone allows us to study applications behavior under a similar system environment where the trace are derived. Furthermore, even when the application behavior is similar, we have found it to be virtually impossible to predict the miss rate as various system parameters are changed without considering algorithm characteristics.

After showing that the data reference modeling approach is accurate and is not too onerous to use, we used the model to predict the cache miss ratios for different system configurations.

7 Acknowledgments

The research reported in this paper is funded by NSF grant # MIP-9012773 and DARPA contract # N00014-87-K-0825. Jory Tsai was supported by Digital Equipment Corporation.

References

[1] Anant Agarwal A Locality-Based Multiprocessor Cache Interference Model. *MIT VLSI memo 89-565*, October 1989.

[2] Anant Agarwal and Anoop Gupta. Memory-Reference Characteristics of Multiprocessor Applications under MACH, In Proceedings of ACM SIGMETRICS, May 1988.

[3] Anant Agarwal, Mark Horowitz, and John Hennessy. *An Analytical Cache Model. ACM Transactions on Computer Systems, Vol. 7, No. 2*, Pages 184-215, May 1989.

[4] Michel Dubois and Faye A. Briggs. Effects of Cache Coherence in Multiprocessors. In *Proceedings of the 9th International Symposium on Computer Architecture*, pages 299–308, IEEE, New York, May 1982.

[5] Michel Dubois and Jin-Chin Wang. *Shared Block Contention in a Cache Coherence Protocol. IEEE Transactions on Computers, Vol. 40, No. 5*, May 1991.

[6] Susan J. Eggers. *Simplicity Versus Accuracy in a Model of Cache Coherency Overhead. IEEE Transactions on Computers, Vol. 40, No. 8*, August 1991.

[7] Jin-Chin Wang. Analytical modeling of shared block contention in cache coherence protocol. *Ph.D. Dissertation*, University of Southern California, Dec. 1990.

[8] Jory Tsai. *Cache Modeling for Very Large Multiprocessor System.* Master thesis, 1992, LCS, Massachusetts Institute of Technology.

[9] Richard Simoni and Mark Horowitz. Modeling the Performance of Limited Pointers Directories for Cache Coherence. In *Proceedings 18th Annual International Symposium on Computer Architecture*, IEEE, 1991.