Performance Prediction for Random Write Reductions: A Case Study in Modeling Shared Memory Programs *

Ruoming Jin
Department of Computer and Information Sciences
Ohio State University
Columbus, OH 43210
jinr@cis.ohio-state.edu

Gagan Agrawal
Department of Computer and Information Sciences
Ohio State University
Columbus, OH 43210
agrawal@cis.ohio-state.edu

ABSTRACT

In this paper, we revisit the problem of performance prediction on shared memory parallel machines, motivated by the need for selecting parallelization strategy for *random write reductions*. Such reductions frequently arise in data mining algorithms.

In our previous work, we have developed a number of techniques for parallelizing this class of reductions. Our previous work has shown that each of the three techniques, *full replication*, *optimized full locking*, and *cache-sensitive*, can outperform others depending upon problem, dataset, and machine parameters. Therefore, an important question is, *"Can we predict the performance of these techniques for a given problem, dataset, and machine?"*.

This paper addresses this question by developing an analytical performance model that captures a two-level cache, coherence cache misses, TLB misses, locking overheads, and contention for memory. Analytical model is combined with results from micro-benchmarking to predict performance on real machines. We have validated our model on two different SMP machines. Our results show that our model effectively captures the impact of memory hierarchy (two-level cache and TLB) as well as the factors that limit parallelism (contention for locks, memory contention, and coherence cache misses). The difference between predicted and measured performance is within 20% in almost all cases. Moreover, the model is quite accurate in predicting the relative performance of the three parallelization techniques.

1. INTRODUCTION

Predicting performance of a program on a parallel machine has always been a hard problem. A variety of approaches have been taken, including detailed simulations [5], profiling [15], analytical modeling [21], and micro-benchmarking [17]. Profiling and simulation based approaches can be quite time consuming, but usually lead to more realistic predictions. Analytical models can provide results in a short time, but researchers have had only a limited success in validating these results on real machines.

The feasibility and appropriateness of a particular approach depends upon the goal in performing the prediction. Tools and techniques for performance prediction have been developed with many goals, including improving architectural designs, understanding system factors and trade-offs, or deciding suitability of a parallel machine or a particular configuration for an application.

In this paper, we revisit the problem of performance prediction for shared memory parallel machines, with the goal of choosing the best technique among a set of parallelization techniques for a class of problems. Our work is motivated by the need for selecting parallelization technique for *random write reductions*, which arise in data mining algorithms.

1.1 Random Write Reductions

We define a random write reduction loop to have the following characteristics: 1) the elements of an array or a collection may be updated in multiple iterations of the loop, but only using associative and commutative operations (we refer to such a collection or array as a *reduction object*), 2) there are no other loop carried dependencies, except on the elements of the reduction object. Thus, the iterations of the loop can be reordered without affecting the correctness of the computation, and 3) the element(s) of the reduction object updated in a particular iteration of the loop cannot be determined statically or with *inexpensive* runtime preprocessing.

Algorithms for many data mining tasks, including association mining, clustering, classification, constructing artificial neural networks, and learning by analogy involve random write reductions [10]. With the availability of large datasets in many application areas, it is becoming increasingly important to execute data mining tasks in parallel. At the same time, technological advances have made shared memory parallel machines more scalable. Large shared memory machines with high bus bandwidth and very large main memory are being sold by several vendors. Vendors of these machines are targeting data warehousing and data mining as major markets. Thus, shared memory parallelization of random write reductions is of significant interest.

In our previous work, we have developed several techniques for parallelizing random write reductions [10, 11]. One of the tech-

*This research was supported by NSF CAREER award ACI-9733520, NSF Grant ACR-9982087, and NSF Grant ACR-0130437

niques involves creating a copy of the reduction object for each thread and is referred to as *full replication*. The other techniques use locking to avoid race conditions. Among the locking schemes, two have shown particularly promising performance. They are *optimized full locking* and *cache-sensitive locking*. In optimized full locking, there is one lock associated with each reduction element. Further, to avoid additional cache misses because of a large number of locks, this lock is allocated on the same cache block as the reduction element. To reduce the memory requirements for locks in optimized full locking, we have designed the cache-sensitive locking scheme. In this scheme, there is a single lock for all reduction elements on the same cache block, which is also allocated within the same cache block.

Our previous work has shown that each of these three techniques can outperform others depending upon the problem, dataset, and machine parameters. Therefore, an obvious question is, *"Can we predict the performance of these techniques for a given problem, dataset, and machine?"*.

1.2 Contributions and Organization

We have developed a detailed model of the performance of random write reductions on a shared memory machine, parallelized using the three techniques we mentioned. We model a two-level cache, coherence cache misses, TLB misses, locking overheads, and contention for memory. A random write reduction loop is parameterized with the size of the reduction object, the amount of computation in each iteration of the loop, and the size of each reduction element. To obtain the performance prediction from our model, we measure 1) execution time of one iteration of the loop when the reduction object fits in L1 cache, and 2) the latency of L1 and L2 cache misses, TLB misses, coherence cache misses, and memory system service time. Cache miss and TLB miss overheads are added to the measured execution time (factor 1 above) to predict sequential execution times when the reduction object does not fit in L1 cache. Overheads due to memory contention, coherence cache misses, and the cost of waiting for locks are added to predict parallel execution times. Overall, we believe that our modeling and performance prediction methodology has implications well beyond shared memory parallelization of random write reductions.

We have validated our model on two different SMP machines. Our results show that our model effectively captures the impact of memory hierarchy (two-level cache and TLB) as well as the factors that limit parallelism (contention for locks, memory contention, and coherence cache misses). The difference between predicted and measured performance is within 20% in almost all cases. Moreover, the model is quite accurate in predicting the relative performance of the three parallelization techniques.

The rest of the paper is organized as follows. We describe the problem of random write reductions and the proposed parallelization techniques in Section 2. Our analytical model is presented in Section 3. Experimental validation of our performance model is presented in Section 4. We compare our work with related research efforts in Section 5 and conclude in Section 6.

2. PROBLEM STATEMENT AND PARALLELIZATION TECHNIQUES

In this section, we further elaborate on the characteristics of random write reductions. We explain the difficulties in parallelizing them efficiently. Later, we present the parallelization techniques we have developed for this class of codes.

2.1 Random Write Reductions

```
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)   =  Compute(e) ;
        RObj(i)    =  Reduc(RObj(i),val) ;
    }
}
```

Figure 1: Structure of Common Data Mining Algorithms

A canonical random write loop is shown in Figure 1. The function $Reduc$ is an associative and commutative function. Thus, the iterations of the for-each loop can be performed in any order. The data-structure *RObj* is referred to as the reduction object.

The main correctness challenge in parallelizing a loop like this on a shared memory machine arises because of possible race conditions when multiple processors update the same element of the reduction object. The element of the reduction object that is updated in a loop iteration (i) is determined only as a result of the processing.

The major factors that make these loops challenging to execute efficiently and correctly are as follows: 1) It is not possible to statically partition the reduction object so that different processors update disjoint portions of the collection. Thus, race conditions must be avoided at runtime. 2) The execution time of the function $Compute$ can be a significant part of the execution time of an iteration of the loop. Thus, runtime preprocessing or scheduling techniques cannot be applied. 3) In many of algorithms, the size of the reduction object can be quite large. This means that the reduction object cannot be replicated or privatized without significant memory overheads, and 4) The updates to the reduction object are *fine-grained*. The reduction object comprises a large number of elements that take only a few bytes, and the for-each loop comprises a large number of iterations, each of which may take only a small number of cycles. Thus, if a locking scheme is used, the overhead of locking and synchronization can be significant.

2.2 Parallelization Techniques

We focus on three approaches for parallelizing random write reductions. These techniques are, *full replication*, *optimized full locking*, and *cache-sensitive locking*. For motivating the optimized full locking and cache-sensitive locking schemes, we also describe a simple scheme that we refer to as *full locking*.

Full Replication: One simple way of avoiding race conditions is to replicate the reduction object and create one copy for every thread. The copy for each thread needs to be initialized in the beginning. Each thread simply updates its own copy, thus avoiding any race conditions. After the local reduction has been performed using all the data items on a particular node, the updates made in all the copies are *merged*.

We next describe the locking schemes. The memory layout of the three locking schemes, *full locking*, *optimized full locking*, and *cache-sensitive locking*, is shown in Figure 2.

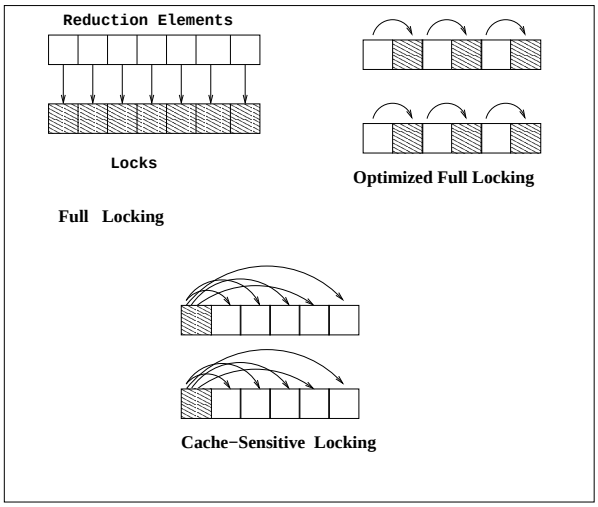

Figure 2: Memory Layout for Various Locking Schemes

Full Locking: One obvious solution to avoiding race conditions is to associate one lock with every element in the reduction object. After processing a data item, a thread needs to acquire the lock associated with the element in the reduction object it needs to update.

In our experiment with apriori, with 2000 distinct items and support level of 0.1%, up to 3 million candidates were generated [11]. In full locking, this means supporting 3 million locks. Supporting such a large numbers of locks results in overheads of three types. The first is the high memory requirement associated with a large number of locks. The second overhead comes from cache misses. Consider an update operation. If the total number of elements is large and there is no locality in accessing these elements, then the update operation is likely to result in two cache misses, one for the element and second for the lock. This cost can slow down the update operation significantly.

The third overhead is of *false sharing* [8]. In a cache-coherent shared memory multiprocessor, false sharing happens when two processors want to access different elements from the same cache block. In full locking scheme, false sharing can result in cache misses for both reduction elements and locks.

Optimized Full Locking: Optimized full locking scheme overcomes the the large number of cache misses associated with full locking scheme by allocating a reduction element and the corresponding lock in consecutive memory locations, as shown in Figure 2. By appropriate alignment and padding, it can be ensured that the element and the lock are in the same cache block. Each update operation now results in at most one cold or capacity cache miss. The possibility of false sharing is also reduced. This is because there are fewer elements (or locks) in each cache block. This scheme does not reduce the total memory requirements.

Cache-Sensitive Locking: The final technique we describe is *cache-sensitive locking*. Consider a 64 byte cache block and a 4 byte reduction element. We use a single lock for all reduction elements in the same cache block. Moreover, this lock is allocated in the same cache block as the elements. So, each cache block will have 1 lock and 15 reduction elements.

Cache-sensitive locking reduces each of three types of overhead associated with full locking. This scheme results in lower memory requirements than the full locking and optimized full locking schemes. Each update operation results in at most one cache miss, as long as there is no contention between the threads. The problem of false sharing is also reduced because there is only one lock per cache block.

One complication in the implementation of cache-sensitive locking scheme is that modern processors have 2 or more levels of cache and the cache block size is different at different levels. Our implementation and experiments have been done on machines with two levels of cache, denoted by L1 and L2. Our observation is that if the reduction object exceeds the size of L2 cache, L2 cache misses are a more dominant overhead. Therefore, we have used the size of L2 cache in implementing the cache-sensitive locking scheme.

3. ANALYTICAL MODEL

This section develops analytical models for the three shared memory parallelization techniques we described earlier.

The code for the canonical random write reduction loop we are modeling was previously shown in Figure 1. All symbols used in our performance model are defined in Table 1. The total number of reduction elements is denoted by E. Each element takes S_e bytes. In our implementation, each lock takes the same number of bytes as the reduction elements, or 8, whichever is smaller. For the purpose of our analysis, we assume that the number of bytes taken by each lock (S_l) is the same as S_e.

We assume that there are a total of t threads executing the reduction loop. The number of iterations or updates performed by each thread is denoted by N. Throughout our analysis, we assume that each thread has an equal probability of accessing any reduction element during an update operation.

The total execution time of the loop is denoted by T_{loop} and the average execution time of each iteration is denoted by $T_{average}$. Clearly,

$$T_{loop} = T_{average} \times N$$

$T_{average}$ comprises three components, the time required for the compute function (denoted by $T_{compute}$), the time required for the reduction function, (denoted by T_{reduc}), and the overhead of the loop (denoted by $T_{loop\_overhead}$). Formally,

$$T_{average} = T_{compute} + T_{reduc} + T_{loop\_overhead}$$

$T_{compute}$ and $T_{loop\_overhead}$ are generally independent of the size of the reduction object. Our focus is on the cost of performing reduction functions. During the execution of the reduction function, a thread may have to wait to acquire a lock, may incur cache misses, TLB misses, or have to stall because of memory contention.

So, we have

$$T_{reduc} = T_{update} + T_{wait} + T_{cache\_miss} +$$

Symbol	Definition
E	Number of reduction elements
$S_e = S_l$	Size of each element or lock
S_b	Size of each cache block
S_c	Total cache size
S_p	Page size
t	Number of threads
N	Number of update operations per thread
M	Memory requirements of reduction elements and locks
p	Number of cache blocks in the cache that hold reduction elements
q	Average number of memory blocks sharing the same cache block
k	Number of reduction elements in a cache block
T_{loop}	Total execution time of the loop
$T_{average}$	Average execution time for one iteration
T_{update}	Execution time of the Reduc function without cache, memory, or waiting overheads
$T_{perfect}$	Execution time of one iteration without cache, memory, or waiting overheads
$T_{compute}$	Execution time for the Compute function
T_{reduc}	Execution time for the Reduc function
$T_{loop\_overhead}$	Loop overhead
T_{wait}	Average waiting time to acquire a lock
T_{cache_miss}	Overhead of cache misses in the Reduc function
T_{tlb_miss}	Overhead of TLB misses in the Reduc function
$T_{memory\_contention}$	Overhead of memory contention in the Reduc function

Table 1: Definition of Symbols Used

$$T_{tlb\_miss} + T_{memory\_contention}$$

Here, T_{update} is the cost of performing reduction operation when there is no contention for the lock and the lock and reduction element accessed are in $L1$ cache. T_{cache_miss} is the overhead because of cache misses and T_{wait} is the time spent waiting to acquire a lock. T_{tlb_miss} is the overhead because of TLB misses, and $T_{memory\_contention}$ is the stall time because of memory contention.

We denote

$$T_{perfect} = T_{compute} + T_{update} + T_{loop\_overhead}$$

$T_{perfect}$ represents the cost of executing one iteration of the loop with no contention for locks, and no overhead because of the memory hierarchy. We also have

$$T_{average} = T_{perfect} + T_{wait} + T_{cache\_miss} +$$

$$T_{tlb\_miss} + T_{memory\_contention}$$

$T_{perfect}$ can be measured by executing a small problem where all reduction elements and locks fit in the level one (L1) cache, and only one thread is executed. Though it is possible that an analytical model for computing $T_{perfect}$ can be developed, there are several advantages in measuring it directly. First, there is no need to analyze the loop body to count the number and type of operations. Second, we don't need to micro-benchmark every different type of operation. Finally, we don't need to model the instruction level parallelism that the processor supports. Recent work on modeling microprocessors supporting ILP has shown that it can be extremely complicated [2, 18]. Moreover, if $T_{perfect}$ is measured using a small dataset, the time required for performing this measurement will be very small.

3.1 Cost of Waiting

We now compute the time spent waiting to acquire a lock. Obviously, this factor does not arise in the full replication scheme. The analysis presented here is parameterized with the number of locks and applies to both optimized full locking and cache sensitive locking scheme. The total number of locks in the cache sensitive locking scheme is much smaller, which increases the time spent waiting to acquire a lock.

Each thread operates independently during the *compute* phase. Before performing the update on the reduction element, the thread needs to acquire a lock. We can divide $T_{average}$ into three components, the time that each thread executes independently (denoted by a), the time spent holding the lock (denoted by b), and the time spent waiting to acquire a lock (denoted by T_{wait}). Formally,

$$a = T_{compute} + T_{loop\_overhead}$$

$$b = T_{reduc} - T_{wait}$$

T_{wait} is the quantity we need to compute. The total number of threads is t and the number of locks is m.

The problem can be solved using standard results from queuing theory. Each locks can be viewed as an independent queue. For simplifying the problem, we assume that the lock has a fixed service time, b. Then, in queuing theory terminology, each lock has an associated $M/D/1$ queue [13]. If the fraction of the time the lock is acquired by any of the threads (i.e. busy) is U, then the expected

waiting time is given by

$$T_{wait} = \frac{bU}{2(1-U)}$$

If the arrival rate for requests to acquire a particular lock is λ, U is given as

$$U = \lambda b$$

We next compute the arrival rate of requests for acquiring a particular lock. Each thread makes a request to acquire a lock every $a + b + T_{wait}$ units of time. Since there are t threads and each thread has an equal probability of accessing each lock, we have

$$\lambda = \frac{t}{(a+b+T_{wait})m}$$

Thus, we get

$$T_{wait} = \frac{b^2 t}{2m(a+b+T_{wait})} \times \frac{1}{1-(bt)/((a+b+T_{wait})m)}$$

This gives us a quadratic equation in T_{wait}. After solving the equation and making some simplifications, we have

$$T_{wait} = \frac{b}{2(a/b+1)(m/t)-1}$$

This expression matches our intuition that the waiting time increases as the number of threads (t) or the service time (b) increases and decreases as the number of locks (m) or the compute time (a) increases.

3.2 Cost of Cache Misses

This subsection focuses on analyzing the cost of cache misses, including capacity, conflict, and coherence cache misses [8].

In a shared memory parallel machine, cache misses can be classified into four categories: 1) *Cold misses*, which arise when a cache block is accessed for the first time, 2) *Capacity misses*, which occur due to the limited capacity of the cache, 3) *Conflict misses*, which occur due to limited associativity of the cache, and 4) *Coherence misses*, which occur when a cache block is invalidated by other processors.

Since our focus is on long running applications, (i.e. $N >> E$), the overall impact of cold cache misses is not significant. Therefore, we only focus on capacity, conflict, and coherence misses. For simplicity, we only consider direct-mapped cache. This allows us to analyze conflict and capacity cache misses together. We are modeling a two-level cache, where the two levels are denoted by $L1$ and $L2$, respectively.

The sum of probabilities of capacity and conflict misses at the $L2$ level is denoted by $Smiss2$. Here, S stands for cache misses due to a single processor. Similarly, the sum of the probability of capacity and conflict misses at the $L1$ level, when the data is available at the L2 cache, is denoted by $Smiss1$. The probability of a coherence miss during an update operation is denoted by $Cmiss$. A coherence miss involves getting a cache block from other processors' cache or the main memory, and has a different latency than conflict or capacity misses.

Thus we have,

$$T_{cache\_miss} = Smiss1 \times Latency_{L2}$$

$$+Smiss2 \times Latency_{main\_memory}$$

$$+Cmiss \times Latency_{cache\_coherence}$$

3.2.1 Capacity and Conflict Misses

We now individually evaluate the terms $Smiss1$ and $Smiss2$. The total memory requirement of the reduction object and associated locks is denoted by M. Table 2 shows the factor M under the three schemes we are evaluating.

The size of a cache block (or line) at the $L1$ level is S_{b1} and the size of a cache block at the $L2$ level is S_{b2}. The total cache size at levels $L1$ and $L2$ is S_{c1} and S_{c2}, respectively. Whenever there is no scope for ambiguity, we drop the number denoting the cache level.

The factor p denotes the number of blocks in the cache. We have $p1 = \frac{S_{c1}}{S_{b1}}$ and $p2 = \frac{S_{c2}}{S_{b2}}$.

The factor q denotes the number of memory blocks that share the same cache block in a direct-mapped cache. Clearly, $q1 = \frac{M}{S_{c1}}$ and $q2 = \frac{M}{S_{c2}}$. If $q \leq 1$, the reduction object completely fits in the cache and we do not have any capacity or conflict misses at that level.

Consider two consecutive accesses to the same cache block. q memory blocks map to the same cache block. The probability of cache reuse, i.e. the two accesses to the same cache block are to the same memory block is $1/q$. We use this observation to compute $Smiss1$ and $Smiss2$. We first consider the case that $M \leq S_{c2}$, i.e. the reduction object and locks fit in $L2$ cache. In this case, we have $Smiss1 = 1 - \frac{1}{q1}$ and $Smiss2 = 0$.

Next, we consider the case when $M \geq S_{c2}$. Here, we have

$$Smiss2 = 1 - \frac{1}{q2}$$

and

$$Smiss1 = (\frac{1}{q_2}) \times (1 - \frac{S_{c1}}{S_{c2}})$$

Another complication arises in cache-sensitive locking scheme. As we mentioned in Section 2.2, we typically use the size of $L2$ cache block for implementing the cache-sensitive locking scheme. In modern machines, the size of $L1$ cache block is typically smaller. The ratio between the sizes is S_{b1}/S_{b2}. Then, there is a probability $1 - S_{b1}/S_{b2}$ that the reduction element and the lock are in different $L1$ cache blocks. In this case, the probability of $L1$ cache miss is

$$Smiss1 = (1 - \frac{S_{c1}}{S_{c2}}) \times (\frac{1}{q2} + \frac{1}{q2} \times (1 - \frac{S_{b1}}{S_{b2}}) +$$

$$(1 - \frac{1}{q2}) \times (1 - \frac{S_{b1}}{S_{b2}}))$$

which simplifies to

Memory Cost		Number of elements in a cache block	
M_{f-r}	$= E \times S_e \times t$	k_{f-r}	$= \frac{S_b}{S_e}$
M_{o-f-l}	$= E \times S_e \times 2$	k_{o-f-l}	$= \frac{S_b}{2S_e} = k_{f-r}/2$
M_{cs-l}	$= \frac{E \times S_e}{S_b/S_e - 1} \times S_b/S_e$	k_{cs-l}	$= \frac{S_b}{S_e} - 1 = k_{f-r} - 1$
Average number of blocks sharing one cache line			
q_{f-r}	$= \frac{M_{f-r}}{S_c} = \frac{E}{p \times k_{f-r}}$		
q_{o-f-l}	$= \frac{M_{o-f-l}}{S_c} = \frac{2E}{p \times k_{f-r}}$		
q_{cs-l}	$= \frac{M_{cs-l}}{S_c} = \frac{E}{p \times (k_{f-r}-1)}$		

Table 2: The Memory Parameters for Different Strategies

$$Smiss1 = (1 - \frac{S_{c1}}{S_{c2}}) \times (\frac{1}{q2} + (1 - \frac{S_{b1}}{S_{b2}}))$$

In computing cache misses for full replication, the value of M we use is $E \times S_e$, and not $E \times S_e \times t$. This is because each processor accesses only its own copy of the reduction object.

3.2.2 Coherence Cache Misses

A coherence cache miss occurs when between a processor's two consecutive accesses to the same memory block, another processor invalidates the cache block. Clearly, there are no coherence cache misses in the full replication scheme.

Consider a processor and suppose it accesses a cache block. Let there be k other accesses to cache blocks before this processor accesses the same cache block again. k can range from zero to infinity. The probability of having a particular value of k is

$$(1 - \frac{1}{p})^k \times \frac{1}{p}$$

When the processor accesses the same cache block again, there is a probability $1/q$ that it will be the same memory block it previously accessed. If this is the case, we have the potential of cache reuse, if the cache block has not been invalidated by other processors. Otherwise, we get a capacity or conflict cache miss. The probability that none of the other $t-1$ processors updated the same cache block in the mean time is $(1 - \frac{1}{pq})^{k(t-1)}$.

Putting it together, the probability of cache reuse and no invalidation from other processors is

$$\sum_{k=0}^{\infty} (1 - \frac{1}{p})^k \times \frac{1}{p} \times \frac{1}{q} \times (1 - \frac{1}{pq})^{k(t-1)}$$

This terms simplifies to

$$\frac{1}{p}\frac{1}{q} \times \frac{1}{1 - (1 - 1/p)(1 - 1/pq)^{t-1}}$$

A coherence miss occurs when we have the possibility of cache reuse, but one of the other processors has invalidated the cache block. So, we have

$$Cmiss = \frac{1}{q} - \frac{1}{p}\frac{1}{q} \times \frac{1}{1 - (1 - 1/p)(1 - 1/pq)^{t-1}}$$

3.3 Cost of TLB misses

In modern computer systems, a translation look-aside buffer (TLB) facilitates translation of virtual addresses to physical addresses. A TLB can be viewed as a hardware cache which can hold address translation information for a fixed number of pages. TLB misses can be a source of significant overheads for applications that access a lot of data, and do not have much locality in accesses.

For modeling the cost of TLB misses, we assume that the TLB is fully associative, as is typically the case in most systems. The latency of each TLB miss is denoted by $Latency\_tlb$ and is measured empirically. As defined in Table 1, the cost per iteration of TLB misses is T_{tlb_miss}. The probability of a TLB miss during an update operation is denoted by P_{tlb_miss}. Clearly,

$$T_{tlb\_miss} = P_{tlb\_miss} \times Latency\_tlb$$

We focus on computing P_{tlb_miss}. Note that some processors have separate TLBs for instruction and data, while others have a common TLB. We denoted by $TLB_{effective}$ the total number of TLB entries that can be used for the elements of the reduction object. If the size of each page is S_p, the total data whose page translation information can be stored in a TLB is $S_p \times TLB_{effective}$. If M is the total memory requirement of the reduction object, we do not have any TLB misses (except cold misses) if $M \leq S_p \times TLB_{effective}$. However, if $M > S_p \times TLB_{effective}$, we have

$$P_{tlb\_miss} = \frac{M - S_p \times TLB_{effective}}{M}$$

3.4 Cost of Memory Contention

Contention for memory is a well known factor limiting parallel performance. In random write reductions, as the elements of input dataset are read and processed, cold cache misses occur. This effect is already captured when $T_{perfect}$ is measured. However, when the size of the reduction object is relatively large, an input element that is read can displace a reduction element or lock from the cache. Also, if the reduction object does not fit in L2 cache, capacity misses occur during update operations. Because reduction elements are modified in the cache, such cache misses require a write back operation. A read and write back operation causes significant overhead for the memory system in many machines. This is because the memory subsystem needs some idle cycles between processing a read cache miss and processing a write back operation. Thus, we can say that the memory system has a high latency for processing a *read and write back operation*, which we denote as a RWB operation.

In a SMP machine, RWB requests from multiple processors can

cause memory contention. In this subsection, we model memory contention because of RWB requests. Note that memory contention can also happen because of other factors like coherence misses, which also have a high latency. However, we do not model these currently.

Assume that S is the service time of the memory system to process a RWB operation. S is experimentally measured. We assume that RWB operations are the dominant part of memory system utilization. We model the memory systems as a $M/D/1$ queue with fixed service time S. Let the probability that a loop iteration requires a RWB operation be P_{RWB}.

If the reduction object does not fit into L2 cache, and if l input elements share one cache block, we have

$$P_{RWB} = \frac{1}{l} + Smiss2$$

Let U denote the utilization of the memory system. In a parallel machine with t threads operating on t processors, we have

$$U = \frac{arrival\ rate}{service\ rate} = \frac{t/(T_{average} \times P_{RWB})}{1/S}$$

$$= \frac{S \times t}{T_{average} \times P_{RWB}}$$

Using the standard result for $M/D/1$ queues, we have

$$T_{memory\_contention} = \frac{US}{2(1-U)}$$

By solving the resulting quadratic equation and making some approximations, we have

$$T_{memory\_contention} = \frac{S^2 \times P_{RWB} \times t}{2(\alpha - S \times t \times P_{RWB})}$$

where,

$$\alpha = T_{perfect} + Tcache\_misses + Ttlb\_misses + Twait$$

Note that the above expression for computing $T_{memory\_contention}$ includes the term T_{wait}. Similarly, the expression we derived for T_{wait} earlier includes $T_{memory\_contention}$. Ideally, the two equations should be solved simultaneously to get the values of $T_{memory\_contention}$ and T_{wait}. However, our experience shows that there are no cases where the values of both of these factors are significant. Waiting time is non-negligible only when the number of reduction elements is small, whereas memory contention is significant only when the reduction object does not fit in L2 cache.

For computing $T_{average}$ from $T_{perfect}$, we first compute T_{cache_miss} and T_{tlb}. Both these factors can be computed independent of all other factors. We then compute T_{wait} assuming that $T_{memory\_contention}$ is zero. Finally, we compute $T_{memory\_contention}$.

4. EXPERIMENTAL RESULTS

In this section, we focus on validating our performance model by comparing the predicted execution times against execution times measured on two different real SMP machines. Initially, we describe the SMP machines we conducted our experiments on and the micro-benchmarking experiments we did to evaluate various machine parameters.

4.1 Experimental Platforms and Experimental Design

We used two different SMP machines for our experiments.

The first machine used for our experiments is a 24 processor Sun Fire 6800. Each processor in this machine is a 64 bit, 750 MHz Sun UltraSparc III. Each processor has a 64 KB L1 cache and a 8 MB L2 cache. The total main memory available is 24 GB. The Sun Fireplane interconnect provides a bandwidth of 9.6 GB per second. This configuration represents a state-of-the-art server machine.

The second machine is a Sun Microsystem Ultra Enterprise 450, with 4 250MHz Ultra-II processors and 1 GB of 4-way interleaved main memory. The size of L1 cache is 16 KB and the size of L2 cache is 1 MB. This configuration represents a common SMP machine available as a desk-top or as part of a cluster of SMP workstations.

We have conducted experiments with three major goals. They are: 1) Evaluating how well our performance model predicts the performance with 1 thread, as the size of the reduction object is scaled. Here, the focus is on determining the model's effectiveness in capturing the impact of memory hierarchy. 2) Evaluating how well our model predicts the speedups achieved by the three techniques. Here, we focus on determining the efficacy in modeling waiting time for locks, coherence cache misses, and contention for memory, and 3) Evaluating how successful the model is in predicting when a parallelization technique outperforms others, i.e, can our model we used as a basis for choosing a parallelization technique.

4.2 Micro-benchmarking

We now describe the micro-benchmarking experiments we did and the machine parameters we obtained. The parameters we obtained are shown in Table 3.

To measure the size and miss penalty of $L1$ and $L2$ cache, we used the *LMbench tool-set* [14]. To measure the size and miss penalty associated with the TLB, we used the method described by Saavedra *et al.* [16]. The key idea in this method is to measure the time required for accessing arrays of increasing size, accessed using a stride that equals the page size.

For computing cache coherence miss latency, we use a method similar to the one described by Hristea *et al.* [9]. Finally, we also computed the service time of the memory system in processing a read and write back (RWB) request. The method we used is again similar to the one developed by Hristea *et al.* [9]. Our experience with random write reductions showed that memory contention is insignificant on Sun Fire 6800. Therefore, the RWB service time was evaluated only for Sun Enterprise 450.

4.3 Results on the Large SMP Machine

We now present experimental results on Sun Fire 6800. We were able to experiment with up to 12 threads. In all experiments, the

		Sun Fire 6800	Sun Enterprise 450
$L1$ Cache	size	64KB	16KB
	miss penalty	18 ns	40 ns
$L2$ Cache	size	8MB	1MB
	miss penalty	273 ns	267 ns
Main Memory	page size	8KB	8KB
	RWB service time	n/a	122 ns
TLB	num. of entries	512-I&D	64-D
	miss penalty	127 ns	167 ns
Cache Coherency	miss penalty	355 ns	406 ns

Table 3: Parameters Extracted from Micro-Benchmarking Experiments

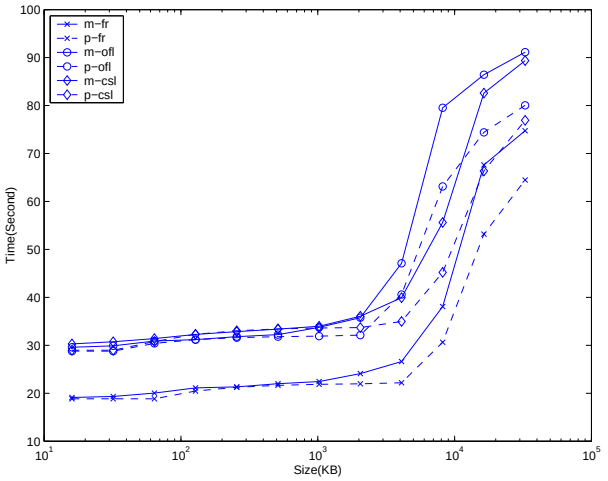

Figure 3: Predicting performance with 1 thread as reduction object size is scaled: all 3 techniques, Sun Fire 6800

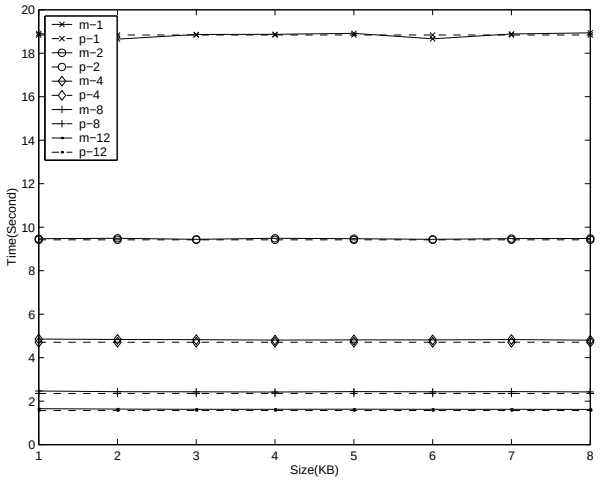

Figure 4: Speedup of full replication with small reduction object sizes, Sun Fire 6800

size of the input data set was 1 GB, comprising 128 million elements.

As we stated earlier, we compute the term $T_{perfect}$ by executing the loop on a small reduction object that fits in L1 cache. Sequential execution times when the reduction object sizes are larger are predicted by adding the overheads because of cache misses and TLB misses. In Figure 3, we evaluate the effectiveness of our model in predicting sequential execution times for all three techniques. We show measured (denoted by m) and predicted (denoted by p) execution times for full replication (denoted by fr), optimized full locking (denoted by ofl) and cache sensitive locking (denoted by csl). By sequential execution, here we mean execution with a single thread where locking is still used. The reduction object size is varied from 16 KB to 32 MB. Note that the amount of computation in the loop does not change as the reduction object is changed, because it only depends upon the size of the input dataset.

The model is very accurate in predicting performance when the reduction object fits in L2 cache. When that is not the case, the model still predicts the performance within 20%. An important observation is that the model accurately predicts the relative performance of the three techniques. With a single thread, full replication is always the winner, because it involves no memory or computational overheads associated with locks. When the overhead of memory hierarchy is not significant, optimized full locking outperforms cache sensitive locking. This is because cache sensitive locking requires more computation in finding the address of the lock. However, when the overhead of memory hierarchy is significant, cache sensitive locking performs marginally better than optimized full locking.

We next focus on evaluating the model's accuracy in predicting performance with more than 1 thread. Two graphs are presented for each technique, one with small reduction objects (up to 8 KB) and another with larger reduction objects (16 KB to 32 MB). The results from full replication are presented in Figures 4 and 7, those from optimized full locking are presented in Figure 5 and 8, and those from cache sensitive locking are presented in Figure 6 and 9. All figures show measured and predicted times with 1, 2, 4, 8, and 12 threads.

When the size of the reduction object is small, the overheads of coherence misses and waiting for locks are large. Therefore, the locking schemes only achieve relatively low speedups, i.e., less than 6 on 12 threads. Our model does a good job in predicting this. Predicted parallel execution times are lower than the measured times, showing that our model is underestimating the parallelization overheads. With up to 8 threads, the disparity is at most 20%. The measured performance with 12 threads on cache sensitive locking is quite different from the predicted performance, particularly on 1 KB, 2 KB, and 5 KB reduction objects. We believe that when a large number of threads share such a small reduction object, the machine behavior is hard to explain or capture in an analytical model.

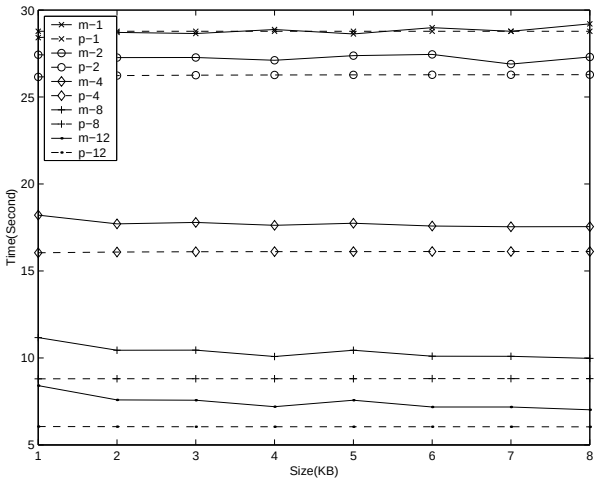

Figure 5: Speedup of optimized full locking with small reduction object sizes, Sun Fire 6800

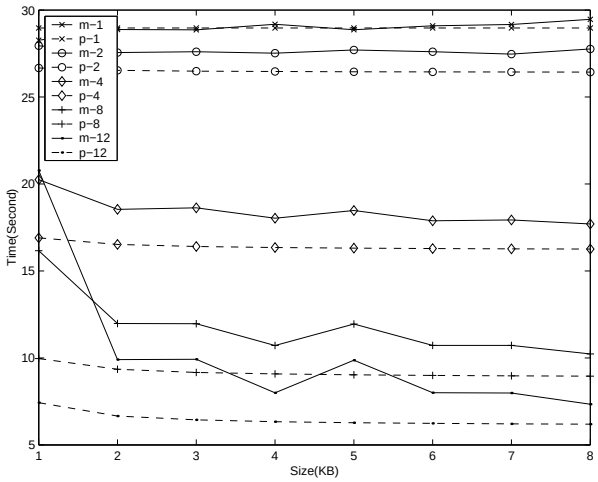

Figure 6: Speedup of cache sensitive locking with small reduction object, Sun Fire 6800

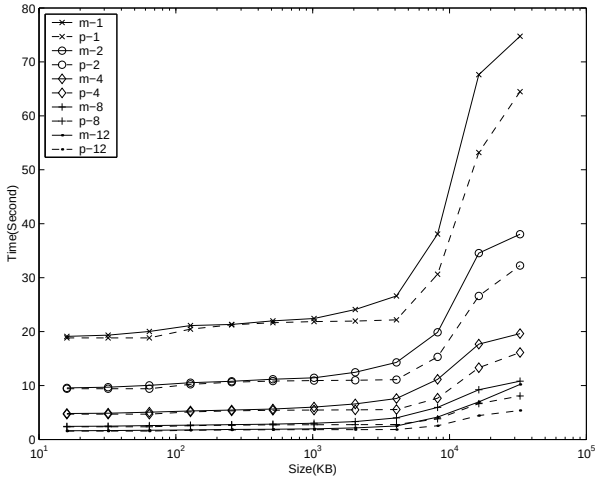

Figure 7: Speedup of full replication with large reduction object sizes, Sun Fire 6800

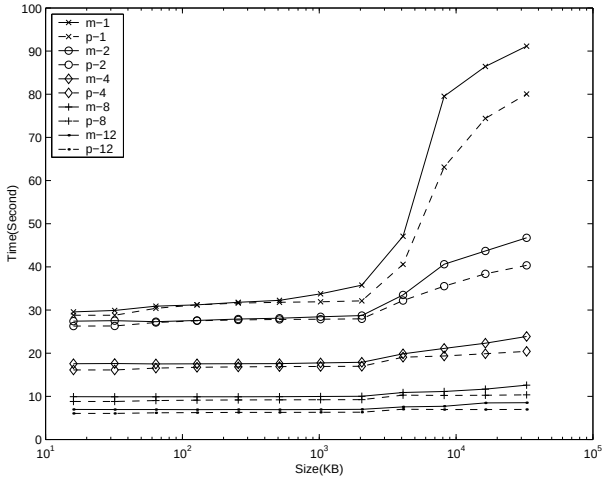

Figure 8: Speedup of optimized full locking with large reduction object sizes, Sun Fire 6800

As the size of reduction object increases, the probabilities of coherence misses and waiting for locks reduce. Higher speedups are seen from all schemes. Again, this is predicted very well by our model.

Our final goal in developing analytical model of performance was to predict when a parallelization technique will outperform other. In Figure 3, we compared the measured and predicted performance for the three techniques when 1 thread is used. In Figure 10, we compare the measured and predicted performance for optimized full locking and cache sensitive locking when 12 threads are used. With the reduction object sizes we have used, the replicated reduction object does not exceed main memory even with 12 threads. Therefore, full replication always gives the best performance. Our model is able to predict this. Therefore, we focus on comparing optimized full locking and cache sensitive locking. The locking schemes outperform the full replication scheme when the size of the replicated reduction object exceeds the available main memory, as we have shown in our previous work [11]. Our performance model currently does not include the overheads of memory thrashing. So, we have not done experiments with cases where the replicated reduction object is out-of-core.

In Figure 10, we show measured and predicted times for the two schemes with 12 threads, and for 12 different reduction object sizes. Approximately 20% difference is seen between predicted and measured execution times. However, in 9 out of 12 cases, the model accurately predicts which scheme is the winner. When the combined size of reduction object and locks doesn't exceed L2 cache, optimized full locking is the winner. This is because cache sensitive locking requires more computation in finding the address of the lock. When the reduction object takes 4 MB, cache sensitive locking becomes the winner. This is because the number of pages required for the reduction object and the locks exceeds the TLB size with optimized full locks, but not with cache sensitive locking. Our model accurately predicts this transition. However, when the size of the reduction object is 8 MB, 16 MB, or 32 MB, our model doesn't accurately predict which scheme is the winner. Our model shows that optimized full locking will have lower execution time, whereas the measurements show that cache sensitive locking does

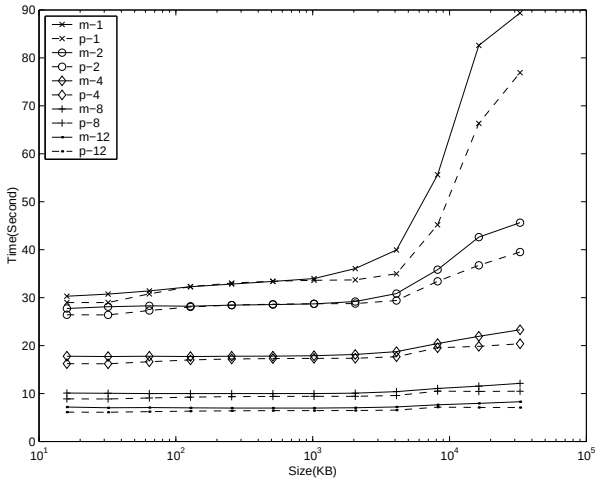

Figure 9: Speedup of cache sensitive locking with large reduction object sizes, Sun Fire 6800

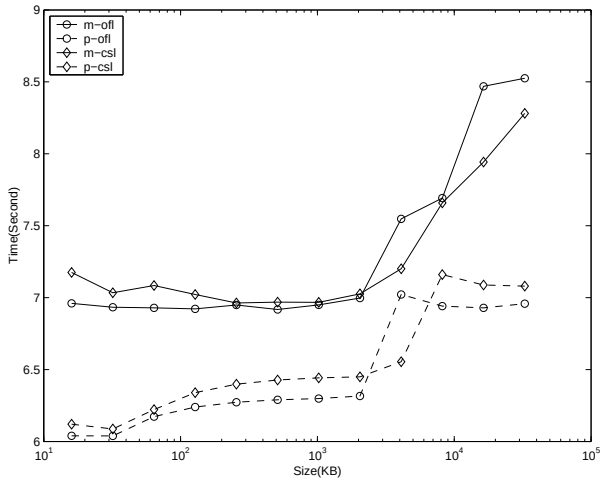

Figure 10: Comparing optimized full locking and cache sensitive locking, 12 threads, Sun Fire 6800

better. We believe this is because our model underestimates the overheads associated with memory hierarchy.

4.4 Results on the Small SMP Machine

We next present the validation results we obtained on a four processor Sun Enterprise 450. All of our experiments were done using a 512 MB input dataset comprising 64 million elements. We were able to use up to 3 threads for our experiments.

Figure 11 shows measured and predicted performance with 1 thread, as the size of the reduction object is increased. The size of reduction object is varied from 16 KB to 16 MB. The predicted performance is within 5% of the measured performance in all cases. Moreover, the model is also very successful in predicting the relative performance of the three schemes. This shows that our model and micro-benchmarking have been effective in capturing the memory hierarchy of this machine.

Figures 12, 13, and 14 show parallel performance with reduction objects of size 1 KB to 8 KB using full replication, optimized full locking, and cache sensitive locking, respectively. As there are no waiting costs or coherence misses in this scheme, full replication gives perfect speedups. Because of the small reduction object, speedups from the locking schemes are limited. The difference between measured and predicted performance is relatively large (consistently around 20%) with 3 threads. We believe that the large number of coherence cache misses in this case cause bus contention. While we capture coherence cache misses in our model, the resulting bus contention is not captured.

Figures 15, 16, and 17 show the measured and predicted parallel performance with reduction objects of size 16 KB to 16 MB using full replication, optimized full locking, and cache sensitive locking, respectively. With larger reduction objects, the probability of coherence misses and waiting for locks decreases. As a result, all techniques have high speedups. The predicted performance is at most 15% different from the measured performance.

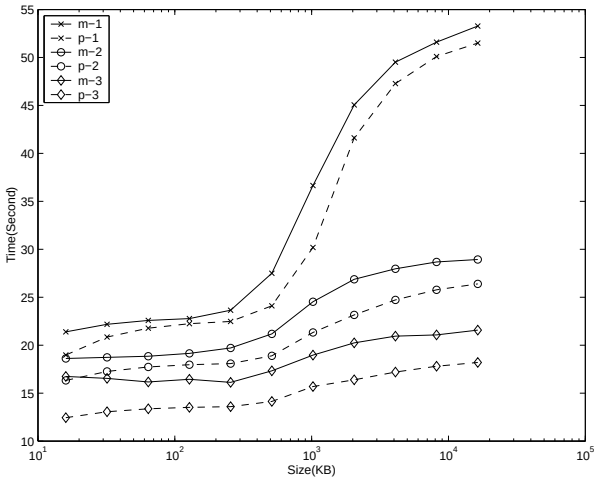

Figure 17: Speedup of cache sensitive locking with large reduction object sizes, Sun Enterprise 450

5. RELATED WORK

The area of performance modeling, prediction, and measurement of parallel programs is a very broad one. It is not possible for us to list all efforts. A variety of approaches have been taken and a number of different system factors have been modeled.

Analytical modeling of shared memory machines has been a popular topic. Recently, researchers have focused on modeling shared memory systems with ILP processors [2, 18]. In comparison, our work focuses on analyzing parallelization techniques for a specific type of programs, but develops a more detailed model, including waiting for locks, coherence misses, and TLB misses. Karlsson *et al.* have developed an analytical model for memory hierarchy performance in decision-support systems [12]. They model cold and capacity cache misses for data intensive applications. Again, our model is significantly more detailed in modeling multiple layers of memory hierarchy, locking overheads, and contention for memory.

The LogP model employs parameterized cost models of the application characteristics and system overheads to model performance on distributed memory machines [4]. This model has also been extended to model the effects of contention [6]. A variant of LogP

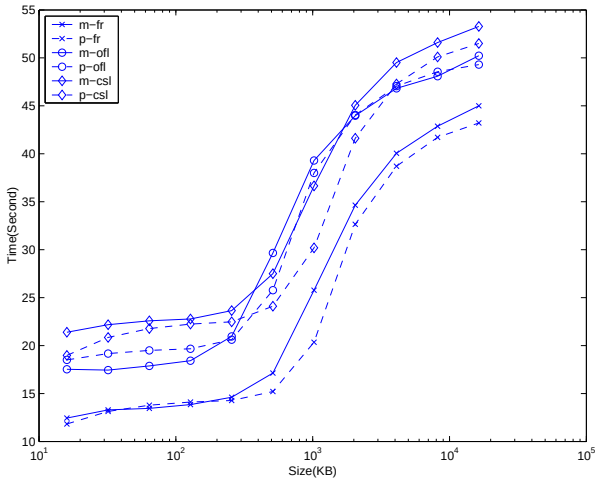

Figure 11: Predicting performance with 1 thread as reduction object size is scaled: All 3 techniques, Sun Enterprise 450

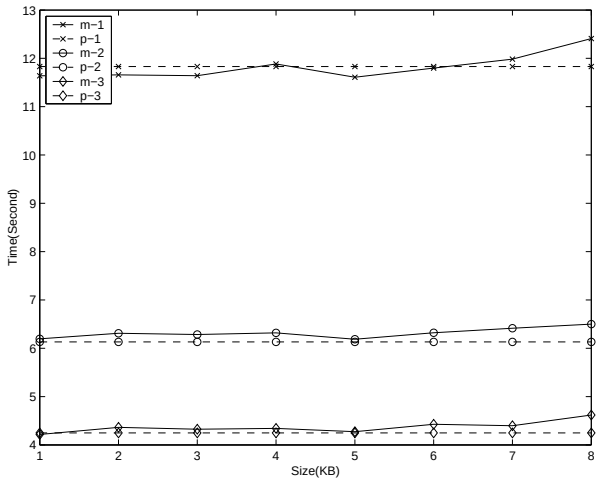

Figure 12: Speedup of full replication with small reduction object sizes, Sun Enterprise 450

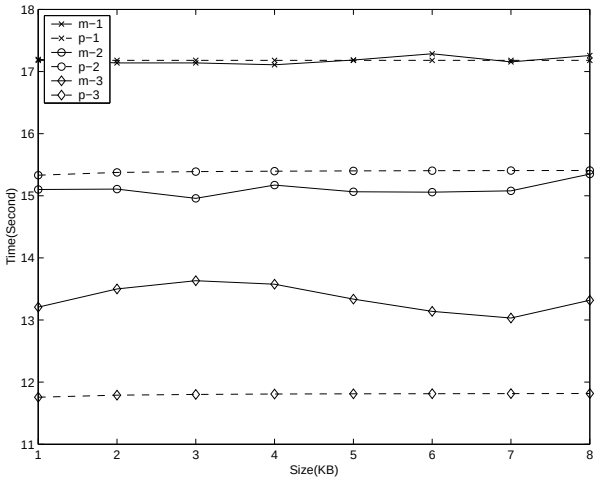

Figure 13: Speedup of optimized full locking with small reduction object, Sun Enterprise 450

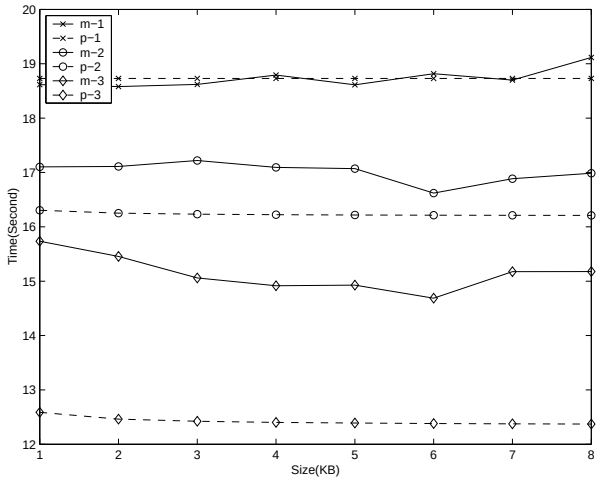

Figure 14: Speedup of cache sensitive locking with small reduction object sizes, Sun Enterprise 450

(LogGP) has been very successfully applied for predictive analysis of a wavefront application [19].

Micro-benchmarking is a popular technique used as a basis for performance prediction. The initial work in this area focused on sequential programs and did not consider memory hierarchy [17]. The work has been since then been extended for parallel machines [16, 22]. We combine micro-benchmarking with probabilistic models to capture the effects of memory hierarchy, contention and locking. We have used several ideas from Hristea *et al.* [9] and Saavendra *et al.* [16] in our micro-benchmarking work.

Detailed analytical models of cache have been developed. Agarwal *et al.* reported an analytical model for uniprocessor cache [1]. In comparison, we model a two-level cache and coherence misses, but focus on a limited application class and do not model associativity. Tsai and Agarwal model data reference patterns to predict cache misses on a cache-coherent multiprocessor [20]. Our treatment of coherence cache misses is specialized for random-write reductions.

Static analysis of cache misses is another well-studied topic [3, 7]. In comparison, we include coherence misses and TLB misses, but again are considerably restricted in the set of programs we model.

6. CONCLUSIONS

In this paper, we have identified a new application for parallel performance prediction. We have developed a detailed analytical model for predicting performance and choosing the best parallelization strategy. Within a uniprocessor, the model captures the impact of memory hierarchy, including a two-level cache and TLB. For predicting parallel performance, we model the overheads because of coherence cache misses, memory contention, and waiting for locks. We are not aware of any previous work on analytical modeling that includes all of the above factors. In our performance prediction approach, memory hierarchy and parallelization overheads are added to the average execution time obtained when the problem fits in first-level cache. This approach alleviates the need for modeling instruction-level parallelism, which can be very complicated.

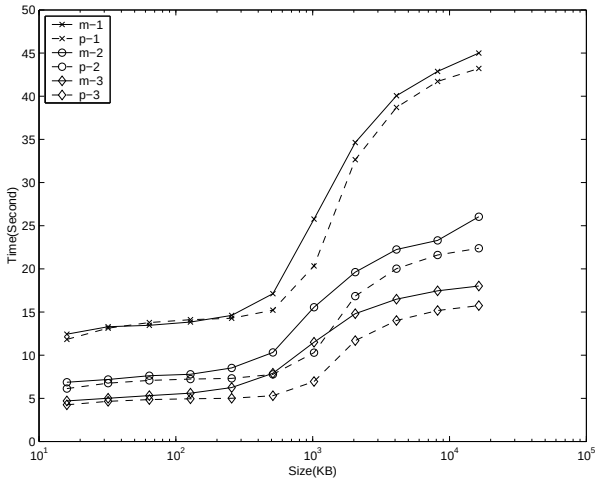

Figure 15: Speedup of full replication with large reduction object sizes, Sun Enterprise 450

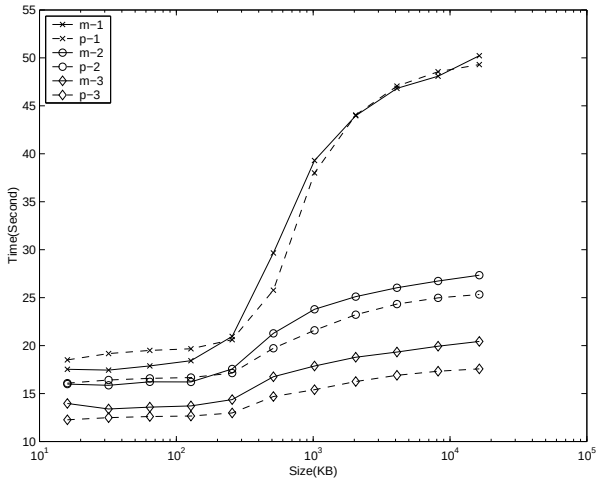

Figure 16: Speedup of optimized full locking with large reduction object sizes, Sun Enterprise 450

We have validated our model on two different SMP machines. Our results show that our model effectively captures the impact of memory hierarchy (two-level cache and TLB) as well as the factors that limit parallelism (contention for locks, memory contention, and coherence cache misses). The difference between predicted and measured performance is within 20% in almost all cases. Moreover, the model is quite accurate in predicting the relative performance of the three parallelization techniques.

7. REFERENCES

[1] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *ACM Transactions on Computer Systems (ACM TOCS)*, 7:184–215, May 1989.

[2] D. H. Albonesi and I. Koren. An Analytical Model of High Performance Superscalar-Based Multiprocessors. In *Proceedings of Conference on Parallel Architectures and Compilation Technology (PACT)*, pages 194–203, June 1995.

[3] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the Conference on Programming Language Design and Analysis*, pages 286–297, June 2001.

[4] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[5] P. M. Dickens, P. Heidelberger, and D. M. Nicol. Parallelized direct execution simulation of message-passing parallel programs. *IEEE Trans. on Parallel and Distributed Systems*, 7(10):1090–1105, 1996.

[6] M. I. Frank, A. Agarwal, and M. K. Vernon. LoPC: Modeling contention in parallel algorithms. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.

[7] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: An Analytical Representation of Cache Misses. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Inc., San Francisco, 2nd edition, 1996.

[9] Cristina Hristea, Daniel Lenoski, and John Keen. Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks. In *Proceedings of SC 97*, 1997.

[10] Ruoming Jin and Gagan Agrawal. A middleware for developing parallel data mining implementations. In *Proceedings of the first SIAM conference on Data Mining*, April 2001.

[11] Ruoming Jin and Gagan Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the second SIAM conference on Data Mining*, April 2002.

[12] M. Karlsson, F. Dahlgren, and P. Stenstrom. An Analytical Model for the Working-Set Sizes in Decision-Support Systems. In *Proceedings of ACM SIGMETRICS*, pages 275–285. ACM Press, 2000.

[13] L. Kleinrock. *Queueing Systems, Vol. I*. John Wiley and Sons, 1975.

[14] Larry W. McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[15] David Ofelt and John L. Hennessy. Efficient Performance Prediction for Modern Microprocessors. In *Proceedings of ACM SIGMETRICS 2000*, pages 229–239, June 2000.

[16] R. Saavedra and A. Smith. Measuring cache and tlb performance and their efect on benchmark run times. *IEEE Transactions on Computing*, pages 1223–1235, 1995.

[17] Rafael H. Saavedra-Barrera and Alan J. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. *ACM Transactions on Computer Systems*, 1996.

[18] Daniel Sorin, Vijay Pai, Sarita Adve, Mary Vernon, and David Wood. Analytical Evaluation of Shared-Memory Systems with ILP Processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 380–391. ACM Press, 1998.

[19] David Sundaram-Stunkel and Mary K. Vernon. Predictive Analysis of a Wavefront Application Using LogGP. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPOPP)*. ACM Press, June 1999.

[20] Jory Tsai and Anant Agarwal. Analyzing multiprocessor cache behaviour through data reference modeling. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1993.

[21] M. Vernon, E. Lazowska, and J. Zahoran. An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols. In *Proceedings of 15th International Symposium on Computer Architecture (ISCA)*, pages 192–202, June 1988.

[22] Stephen J. Von Worley and Alan J. Smith. Microbenchmarking and Performance Prediction for Parallel Computers. Technical Report 95–873, University of California, Berkeley, May 1995.