Scientia Iranica, Vol. 8, No. 4, pp 312–321
© Sharif University of Technology, October 2001

SCIENTIA
IRANICA

Research Note

Combination of Scalable Caching Methods for Weakly Coherent Shared Memory Model

K. Zamanifar*, J.M. Nash[1] and P.M. Dew[1]

In this paper, highly scalable caching methods are described for a weakly coherent shared memory model called WPRAM in which explicit synchronization operations are used to guarantee data coherency. The schemes described here are applied to barrier synchronization and a form of pairwise synchronisation employed by the WPRAM and a combination of the two. An example of the barrier caching method application is shown using the simplex method for linear programming. A parallel sorting algorithm is used to demonstrate the combination of barrier and tag caching schemes. Results are based on simulation of a scalable distributed memory machine. An analytical model is used to describe the performance of the algorithm and verify the simulation results.

INTRODUCTION

The emergence of a number of parallel computational models, including the BSP [1], LogP [2] and WPRAM [3] aim to address the problems of writing a parallel software that is both scalable and portable. These models present the programmer with a well-defined cost model to analyze the performance of the algorithms. There is a growing acceptance that general purpose parallel computers need to be based on a scalable shared memory computational model, with the ability to exploit data locality for good performance. Today, this is commonly achieved by mapping the model onto a distributed memory computer with a scalable interconnect (supporting linear increases in bisection bandwidth). This results in a two-level memory hierarchy, in which data is either local or shared across the machine. The next few years will see a trend towards cache coherent multiprocessors, using the techniques employed by machines such as KSR (cache-only memory) and DASH (distributed directories). This will simplify the programming model by presenting a single level memory hierarchy, consisting of a system-wide shared address space. Multiple copies of a shared variable are, then, automatically maintained in a coherent state by the machine.

One of the reasons for the success of sequential computing is that the explicit form of data transfer between different memory hierarchies has been hidden from the programmer through the use of suitable caching techniques. In a parallel system, the main problem to overcome for the introduction of a caching system is the cache coherency problem. This relates to the fact that the copies of a variable resident in the caches of multiple processors must be invalidated when the value of variable is updated, in order to maintain the consistency of the system.

Snoopy-based cache coherency schemes [4] are limited to small-scale multiprocessors, because of the limited bandwidth of the shared bus. Hardware solutions to the cache coherency problem for multiprocessors with point-to-point connections usually employ a directory-based scheme. Due to the increased complexity of hardware solutions to the cache coherency problem, software assisted schemes [5] have been proposed, which are under the supervision of the compiler (static schemes) or supported by the operating system kernel (dynamic schemes). To support scalability for a large number of processors, storage requirements and run-time overhead related to the cache coherency scheme should be constant or grow sublinearly with the total number of processors.

In this paper, a highly scalable and practical cache coherency scheme for barrier and tag synchronizations is described. The next section is devoted to the barrier caching scheme and simplex method as a case study. Then, the tag caching scheme is described. The combination of the two schemes and a parallel sorting

*. *Corresponding Author, Department of Computer Engineering, University of Isfahan, Isfahan, I.R. Iran.*

1. *Scalable Systems and Algorithms Group, School of Computer Studies, University of Leeds, West Yorkshire, LS2 9JT, UK.*

algorithm as a case study are explained followed by the simulation results. Finally, the analytical performance model is discussed very briefly and the paper ends with some conclusions.

BARRIER CACHING METHOD

The cache coherency problem arises from the fact that the copies of a variable resident in the caches of multiple processors must be invalidated or updated when a new value is written to the variable. According to weak ordering semantics [6], the contents of caches and the shared memory should be consistent at barrier synchronization. The barriers divide the computation into a sequence of *supersteps* [1].

Overview of the Method

The cache coherency problem can be solved by variation of the bulk synchronous cache retention method [7]. The method requires that access to a variable can determine if its coherency is to be maintained by the use of barrier operations. This can be solved by providing a matching pair of barrier statements [8]: One to denote the start of the superstep (*barrier_begin*) and one to denote the end (*barrier_end*). This defines the *environment* of data accesses. The barrier_begin statement allows the underlying system to select the associated caching method, for maintaining cache consistency. The barrier_end statement will then synchronize the processors. This preserves the coherency of shared data at the point where all processes are synchronized by execution of *barrier_end* statement.

Self-invalidation cache coherency schemes are usually software-assisted cache coherency schemes [5,9]. The complexity of hardware-based cache coherency schemes and the limited scalability of bus-based cache coherency schemes have made software-assisted schemes attractive. Software-assisted cache coherency schemes operate at compile-time (statically) or at runtime (dynamically). There are two classes of invalidation schemes: indiscriminate invalidation [10,11] and selective invalidation [9,12]. The indiscriminate invalidation scheme invalidates the entire cache line at certain points in the program. The selective invalidation scheme limits the number of the elements of a cache line to be invalidated. The former scheme is fast but not efficient. This is because some elements within the cache line which are invalidated can potentially be valid, leading to a loss of temporal locality and poor hit ratio. The latter scheme, by keeping some elements of a cache line beyond certain points in the program, increases the hit ratio, however, because of sequential selection and invalidation of data items, it may be much slower than the indiscriminate invalidation scheme. In dealing with the third issue, the approaches used in indiscriminate and selective invalidations are very similar. The ends of some computations, called computational units, and the boundaries of parallel loops [10] are identical. In a cache coherency scheme, the main factors which improve the data locality and support scalability are: retention of data across the caches and maintaining their coherency in a scalable manner.

In the Bulk-Synchronous Parallel (BSP) model [13], memory is consistent at synchronization points. The simple invalidation scheme can be used to cache all the data read during the superstep and to invalidate them at a synchronization point. Although this scheme is scalable, it is not efficient. Limited cache retention scheme [7], which permits some data to be retained across synchronization boundaries, has been used in scalable caching methods for a weakly coherent memory. In this research, which aims to study the use of caching methods in the support of a scalable shared memory for the class of MIMD machines (in particular, a shared memory is weakly coherent, i.e., data is coherent at points of synchronization), the most relevant caching method is the cache retention algorithm. Self-invalidation, retention of data across synchronization points, low storage overhead and the potential to be implemented both in hardware and software are attractive features of this scheme.

The caching scheme is based on the following principle. If a process is currently reading a variable, the process can, then, guarantee that no other process will be updating that variable within that superstep (although other processes may also be reading the variable). Similarly, if a process is updating a variable, no other processes will have access. This leads to the following implication: If a process does not access a variable in the current superstep, another process may be updating its value and so the cached copy must be invalidated before the next superstep begins.

The caching scheme which supports this method is based on the use of a state tag for each variable. The state tag can take the values 0, 1 or 2. When a cache line is initially read from the shared memory, the tag is set to 2. Each time a barrier_end statement is executed, the state of the tag is decremented. The value of the tag will thus decline to 0 if it is not used within the next superstep. A tag value of 0 denotes an invalid variable. Accessing a variable within a superstep will reset the tag value back to 2. The scheme is inherently parallel, since each cache can be updated independently. A potential disadvantage with this scheme is that some cache lines may be unnecessarily invalidated, since although a processor does not access a cache line in a superstep, no other processor accesses the associated variables either.

The contribution of this work is the proposal of a practical and highly scalable cache coherency scheme for a weakly coherent memory, developed from the cache retention method for barrier synchronization. This can be used to support data locality and maintain the coherency of the shared data. The barrier caching method does not suffer the limitations of the broadcasting mechanism of the snoopy-based cache coherency scheme, which limits scalability, nor does it have the serialization problem inherent in directory-based schemes. The barrier caching method is inherently parallel, since the contents of each cache can be invalidated independently. This method is applicable to both hardware and software based schemes, because the related overheads due to validating, invalidating, finding, and updating a variable in a cache line are acceptable in both hardware and software environments. A potential disadvantage with this scheme is that some variables in a cache line may be unnecessarily invalidated in the next superstep, if processes do not access those variables in the current superstep. This affects data locality.

Case Study: Parallel Simplex Method

The simplex method was chosen because of its simplicity (it is implemented using a data parallel approach of a very regular structure) and its non-trivial mapping from a two-level to a one-level memory hierarchy. The method can be used to derive the values of a set of n variables $x_1 \ldots x_n$, which minimize some objective function z of theses variables. This is subject to a set of d constraints on these variables. This can be summarized as follows:

$$\text{Minimize } z = \sum_j (c_j.x_j) \quad \text{for } j = 1 \ldots n$$

Subject to

$$\sum_j (a_{i,j}.x_j) \leq b_i \quad \text{for } i = 1 \ldots d$$

$$x_j >= 0 \text{ for } j = 1 \ldots n \text{ and } d << n.$$

$A = (a_{i,j})$, with b_i and c_j representing the given constants. Parallelism can be exploited by distributing the n columns of A across the processors. The simplex algorithm has been written using three different memory hierarchy models, to demonstrate the issues of performance and program design. The first is based on the two-level memory hierarchy, in which a process explicitly moves data between the shared and local memory. The second method uses the one-level memory hierarchy, in which all data is accessed through the shared memory. The final method additionally employs the caching technique.

Figure 1 shows a part of the related pseudo code based on the one-level memory hierarchy [14]. b, *pivot_column* and *pivot_index* are shared variables which can be accessed through the cache. Each process holds a unique block of A, denoted by *A_segment*[]. The main body of the algorithm consists of two supersteps which proceed in a series of iterations. In each iteration, all processes search their set of columns for a negative objective coefficient value for c_j. One process can then be selected from the successful processes. This can set the pivot column (together with the index of the pivot row) for the current iteration in first superstep. The other processes, since they do not access these variables, will set the associated state tags to 0 at the end of the superstep. This guarantees that they will read the updated values in the next superstep. In the second superstep, all processes can, then, access these results to update their *A_segment*[]. The algorithm completes when there are no negative objective coefficient values remaining.

```
BARRIER_BEGIN (1);
if chosen process
{
    pivot_column ← A_segment[pivot];
    pivot_index ← index of the pivot row;
}
BARRIER_END (1);
BARRIER_BEGIN (2);
    update A_segment using pivot_column and pivot_index
BARRIER_END (2);
```

Figure 1. One-level memory hierarchy solution.

TAG CACHING METHOD

Pairwise synchronization provides a more flexible mechanism than barriers for designing irregularly structured algorithms. The WPRAM model supports this through tag variables. One process may *set* a tag (*T_set*), which can then cause another process which is waiting on the tag (*T_wait*) to be rescheduled. To support the cache coherency under pairwise synchronization, a mechanism that allows the caching system to decide which appropriate cache coherency scheme to be used is needed. Therefore, the caching method in a pairwise synchronization environment is based on the use of a matching pair of tag statements, i.e., *tag_begin* and *tag_end*. The use of *tag_begin* and *tag_end* statements preserves the coherency of shared data at the point where both processes are synchronized.

It should be noted that in cases where one process wants to update the shared memory and the other one wants to read the updated value, a pairwise synchronization is usually used. Under the proposed caching regime, it is obvious that the reader process must refer to the shared memory to read the updated value which has been updated by the writer process. This is because the cache related to the reader process contains the state value of the data, even though the

Shared Variable A	**P1**	**P2**	**P3**
Barrier_begin 1			
	Read A	**Read A**	**Read A**
	Initialise state (2)	*Initialise state (2)*	*Initialise state (2)*
	Cache A	*Cache A*	*Cache A*
	Assign V=A	*Assign V=A*	*Assign V=A*
Barrier_end 1			
	Decrement state	*Decrement state*	*Decrement state*
	(2) ⟶ (1)	(2) ⟶ (1)	(2) ⟶ (1)
Barrier_begin 2			
	Tag_begin	**Tag_begin**	
	Write A		
	Initialise state (2)		
	Assign A=V		
	Writethrough A	**T_wait**	
	T_set		
		Read A	
		Initialise state (2)	
		Cache A	
		Assign V=A	
	Tag_end	**Tag_end**	
Barrier_end 2			
	Decrement state	*Decrement state*	*Decrement state*
	(2) ⟶ (1)	(2) ⟶ (1)	(1) ⟶ (0)
Barrier_begin 3			
	Read A	**Read A**	**Read A**
	Initialise state (2)	*Initialise state (2)*	*Initialise state (2)*
	Assign V=A	*Assign V=A*	*Cache A*
			Assign V=A
Barrier_end 3			
	Decrement state	*Decrement state*	*Decrement state*
	(2) ⟶ (1)	(2) ⟶ (1)	(2) ⟶ (1)

Figure 2. Combination of the barrier and tag cache coherency schemes.

associated state tag can be 1 or 2. Therefore, the cache coherency scheme for pairwise synchronization environment necessitates all read references to bypass the cache and go directly to the shared memory. Under the barrier caching scheme, when a variable is initially read from or written in the shared memory, the state tag is set to 2, and write references update the shared memory through the cache. According to the explanation for the tag cache coherency scheme, *tag_begin* statement causes all read references to be cached from the shared memory. The *tag_end* statement switches the scheme off.

COMBINATION OF BARRIER AND TAG CACHING METHODS

Combinations of barrier and tag cache coherency schemes, based on a unified caching system, support different kinds of parallelism, such as divide-and-conquer algorithms. In divide-and-conquer algorithms, a problem is partitioned into smaller parts, solutions for the parts are found, and then they are combined into one solution for the whole. In such a class of algorithms, at each level, the processes can be barrier synchronized and within each level, the interprocessor communications occur based on pairwise synchronization. An example of this class of algorithms is the parallel sorting algorithm which is explained in detail in the next section.

Under this integrated scheme, the tag cache coherency scheme will be active in the presence of the barrier cache coherency scheme. While the barrier cache coherency scheme decreases the state tag of the shared variables at barrier synchronization points, after a tag_begin, all subsequent read references will reference shared memory.

Example Code

Figure 2 shows a program example in a combination environment of barrier and tag synchronizations using three processes P1, P2 and P3.

First superstep:

Each of the three processes read a copy of the shared variable A into a private variable V. All of the state tags of the cached copies of A are initialized to 2. At the end of the first superstep, all of the state tags are decremented to 1. The cached copies are valid in the next superstep (because the state tags are 1).

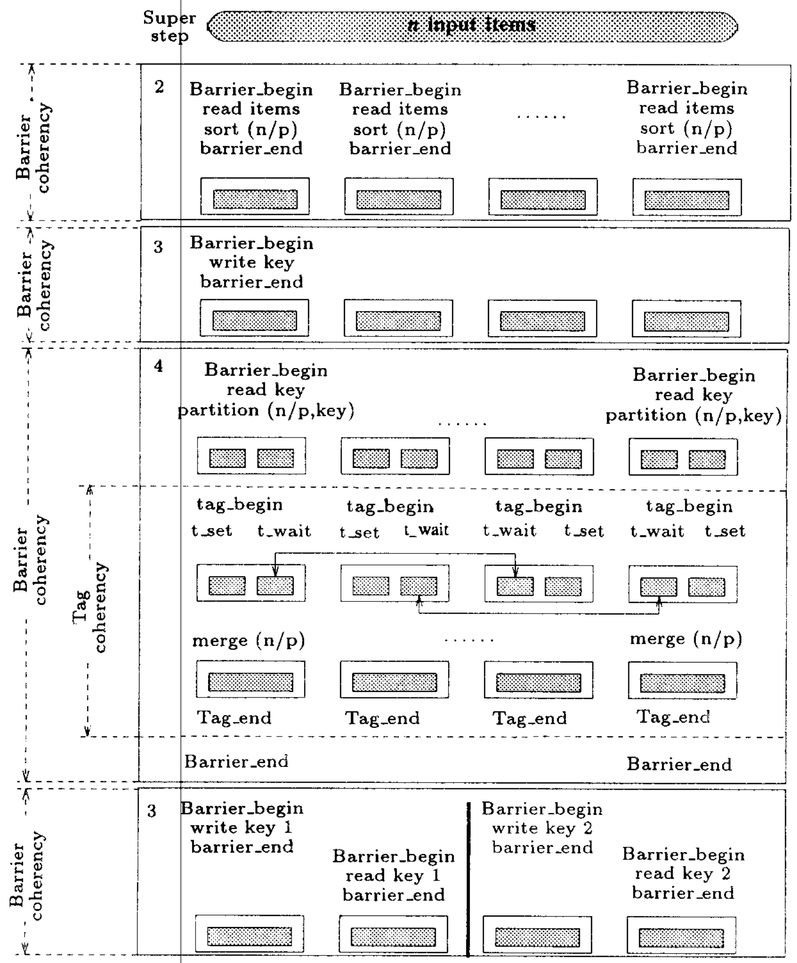

Figure 3. Overview of the parallel sorting algorithm.

Second superstep:

There is a pairwise synchronization between P1 and P2. P2 must be blocked until P1 updates A, then P2 can read the updated value of A. Using the *barrier_begin* primitive, the caching system decides to use the barrier cache coherency scheme. Through using the *tag_begin* primitive, the caching system decides to temporarily switch to the tag cache coherency scheme. In this superstep, P1 updates A in the cache and in the shared memory and unblocks P2. P2 caches A even though the state tag of variable A in its cache is 1. This is because the value of A in the cache P2 is not consistent with the value of A in the shared memory, so tag cache coherency scheme bypasses the cache and causes P2 to read A from the shared memory. The contents of the caches and the shared memory are coherent at the second barrier synchronization. The variable A in the cache P3 is invalid, because it has not been referenced within the previous two supersteps.

Third superstep:

All processes again read A. P1 and P2 can retrieve A from their caches. P3 must access the shared memory.

Case Study: Parallel Sorting Algorithm

The combination of the barrier and pairwise synchronizations, makes this algorithm suitable to be tailored based on the combination of caching methods. The implementation is based on a study reported in [15]. In parallel sorting algorithm, it has been assumed that the input data is randomized. This assumption provides the load balancing of each node. In one-level memory hierarchy implementation, all data movements are carried out in the shared memory through the cache. Figure 3 gives an overview of the parallel sorting algorithm. The algorithm consists of the following computational supersteps:

Superstep 1

Each processor initializes the *counters* and also creates, initializes and unsets a pair of tags.

Superstep 2
The data is sorted. In this superstep, data locality is supported by the barrier caching method.

Superstep 3
The winner processor computes its median element and writes it to the shared global memory, while the loser processors are waiting to access it in the next superstep. The use of the *barrier_end* operation guarantees the consistency of the read and write accesses to this median element.

Superstep 4
This superstep integrates both barrier and tag caching methods. The median element is used by all processors to partition their data in left and right segments. Data locality is maintained by barrier caching method. The processors then exchange their segments so that the first half have the left segments, and the second half have the right segments. The use of the *tag_begin* and *tag_end* statements guarantees the data consistency of the read and write accesses to each segment. Each new pair of segments held by a processor is merged. The processors recursively divide into independent halves. Both halves carry out supersteps 3 and 4 concurrently. The algorithm terminates when the subdivisions contain one processor.

SIMULATION RESULTS

A simulation of the WPRAM exists on the Sun Sparc workstation. The simulator supports the weak memory coherency model of the WPRAM and also the caching methods. A well-defined cost model based on the performance figures of the IMS-T9000 processor and IMS-C104 packet router [16] has been implemented in the simulator. Figure 4 shows the effect of varying the number of processors on the completion time and speed-up for the simplex method. In the case of 64 processors and 6400 matrix columns, adding the caching scheme increases performance by 74%, with a drop of 37% from the solution using a two-level memory hierarchy. In Figure 4, n is the number of variables and d is the number of constraints in the simplex method. In a two level memory hierarchy, a processor explicitly moves data between shared and private memory. Figure 5 depicts the results for the

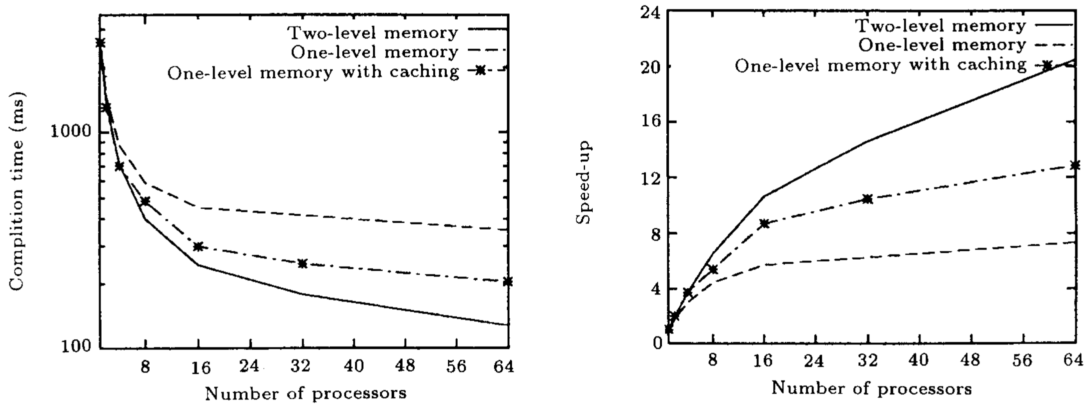

Figure 4. Simulation results for parallel simplex method, $n = 6400$ and $d = 8$.

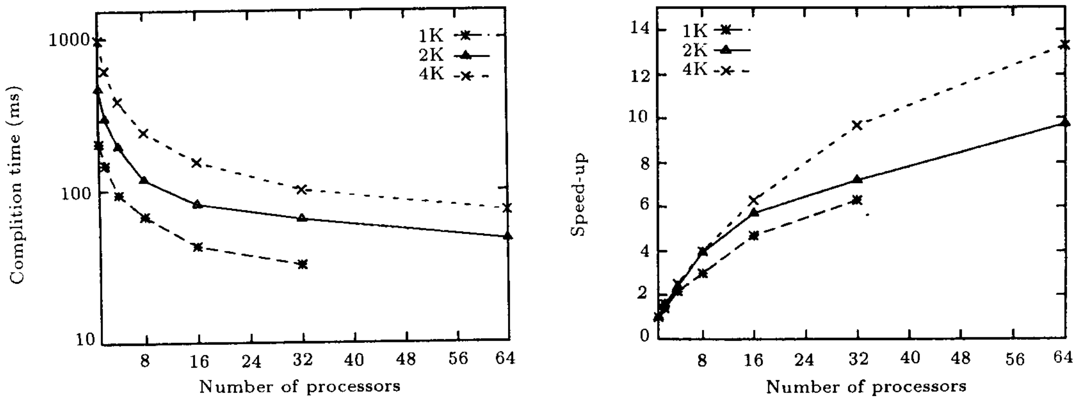

Figure 5. Simulation results for the parallel sorting algorithm.

parallel sorting algorithm. In the case of 64 processors and 4K problem size, the speed-up is around 13.5.

PERFORMANCE MODEL

In this section, a performance model for the WPRAM is described, which can be used to estimate the completion time of an algorithm using the caching technique [17]. An assumption, here, is the use of a scalable interconnection network, giving a linear increase in the bisection bandwidth as more processors are added. The performance model consists of three levels:

- Machine model: The set of operations at the machine level which dominate performance,
- WPRAM model: The operations within the WPRAM model,
- Workload model: A high level characterization of the cache access patterns.

Machine Model

The costing of the machine operations is divided into two groups: Those which perform local operations and those which have access to the network. It is assumed that the network packetizes data accesses into fixed size packets and that these packets contain the same number of data bytes as a cache line. A cache reads miss-results in a cache line being obtained from the shared memory, with the associated packet sent through the network. A write to the cache will also generate a single packet.

The network is characterized by the following parameters:

- D: The network latency. This is the delay between a processor sending a packet and it being received at the destination;
- g: The granularity. The minimum sustained time period between a processor sending consecutive packets. Under the assumption of a scalable network, this is a constant value;
- L: The line/packet size (byte).

Each processor is modelled by a small number of local operations:

- o_s: The send overhead. The time that the sender is engaged in the transmission of a packet;
- o_r: The receive overhead. The time that the sender is engaged in the reception of a returned packet;
- t: The time that the receiver is engaged in receiving a packet of information and sending back a reply;
- s_i: The cost of an integer operation;
- s_f: The cost of a floating point operation;
- c: The time to read/update a cache line.

Table 1 provides example costs for the T9000/C104. The costs include a number of lower level issues for the T9000 processor, one example being the dynamic creation and termination of local processes when issuing messages.

WPRAM Model

The performance model for the WPRAM includes the cost of performing local computation and the access of a line of data, either locally from the cache, or accessed through the network. Table 2 presents an overview of the costs. The read access to the shared data will either find a valid cache line or require remote access, resulting in a packet being sent through the network. A

Table 1. Machine costs for the IMS-T9000/C104 architecture (time in ns).

D	g	o_s	o_r	t	s_i	s_f	c	L
$414 * \log P + 2600$	10,000	2872	1656	1390	10	20	160	16

Table 2. WPRAM costs.

WPRAM Operation	Status	Cost
Read()	hit	$R_{hit} = c$
Read()	miss	$R_{miss} = \max(c + o_s, g) + 2D + \max(t, g) + o_r$
Write()	hit	$W_{hit} = \max(c + o_s, g) + 2D + \max(t, g) + o_r$
Write()	miss	$W_{miss} = \max(c + o_s, g) + 2D + \max(t, g) + o_r$
Read&add()		$R_{opg} = \max(o_s, g) + 2D + \max(t, g) + o_r$
Barrier_begin()		$B_b = s_i$
Barrier_end()		$B_e = 2 \log P(\max(o_s, g) + D + o_r)$

read hit simply uses the cost c, defined at the machine level. A read miss contains the following costs:

- $\max(c + o_s, g)$: The delay in sending the packet into the network is either the cost of initially checking the cache for the data (c) and the overhead of sending the packet o_s, or the network granularity cost g, if this is greater;
- $2D$: The network latency incurred by accessing the remote destination;
- $max(t, g)$: The delay in the destination processor accessing the data and generating a reply;
- o_r: The overhead in receiving the resulting packet.

This is summarized in Figure 6. A write hit and miss have the same cost. Both involve accessing the cache to check for a valid line of data (and updating the data if present) and then writing the data to the shared memory. The *read&add* concurrent operation does not involve any cache access, since the variable is always directly updated in the shared memory. The beginning of a barrier is simply marked by setting an appropriate flag for that process (an integer operation). The end of a barrier involves the synchronization of the processors. This can be accomplished by the use of a balanced spanning tree. The weakly coherent memory model also allows the pipelining of data accesses. The method for costing multiple remote data accesses is demonstrated in Figure 6. The completion time of the two operations shown in the figure can be derived by analyzing the critical path. After the overhead incurred by generating the two packets, the second packet must progress through the network, incur an overhead at the destination, and finally travel back to the destination processor, with a final overhead for receiving the result. This is under the assumption that the receiving overhead o_r is not greater than the overhead incurred at the destination processor. The access of x such packets can, then, be modelled as $x \max(c + o_s, g) + 2D + \max(t, g) + o_r$.

Workload Model

The workload model aims to characterize the patterns of cache hits and misses, so that an estimation of the completion time of an algorithm can be generated from the related WPRAM operation costs.

Table 3. Workload model.

m_{rhit}	Probability of a read hit
m_{rmiss}	Probability of a read miss
m_{whit}	Probability of a write hit
m_{wmiss}	Probability of a write miss

Characterizing the Cost of Shared Data Access

A shared data access operation can be parameterized as to whether it is a read or write access, and the probability of the access being a cache hit or miss. The latter will depend on issues such as the size of the cache line and the size of the data set being accessed by a processor compared to the size of the cache. The relative sizes of the data set and cache will effect the likelihood of a valid cache line being replaced by a new line of data, due to address conflicts. Table 3 summarizes these parameters. The cost of reading n bytes in shared memory can, then, be costed as:

$$n(m_{rhit}R_{hit} + m_{rmiss}R_{miss}).$$

Similarly, the cost of writing n bytes is costed as:

$$n(m_{whit}W_{hit} + m_{wmiss}W_{miss}).$$

Estimating the Probability of a Cache Hit

The probability of a read or write hit can depend on two factors:

- The relative size of the data type being accessed and the cache line. Given a variable of size x bytes, each remote access of a new line of data will also retrieve another $L/x - 1$ similar variables, where L is the size of the cache line in bytes;
- The size of the total data set being accessed compared to the size of the cache. It is assumed here that a line of data is stored in a random location in the cache, for example, using some simple linear cash function on the shared address [18]. Figure 7 shows the cumulative probability distribution function of such a conflict occurring, $prob_{conf}$, for different ratios of the data set and cache sizes. In this paper, a reasonably large cache size compared to the data set is assumed, resulting in a negligible probability of a

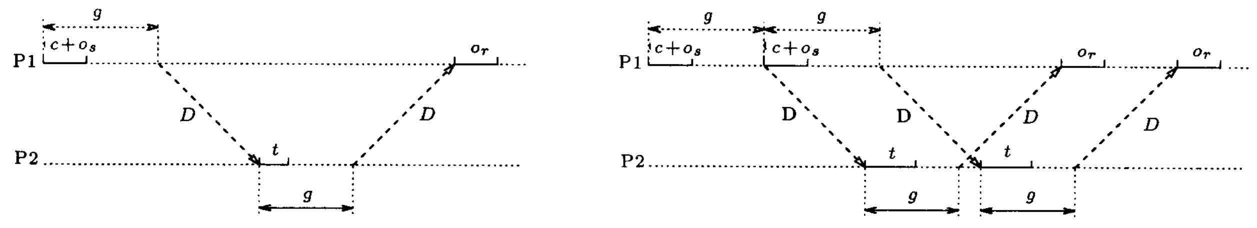

Figure 6. Costing single and multiple remote access memory.

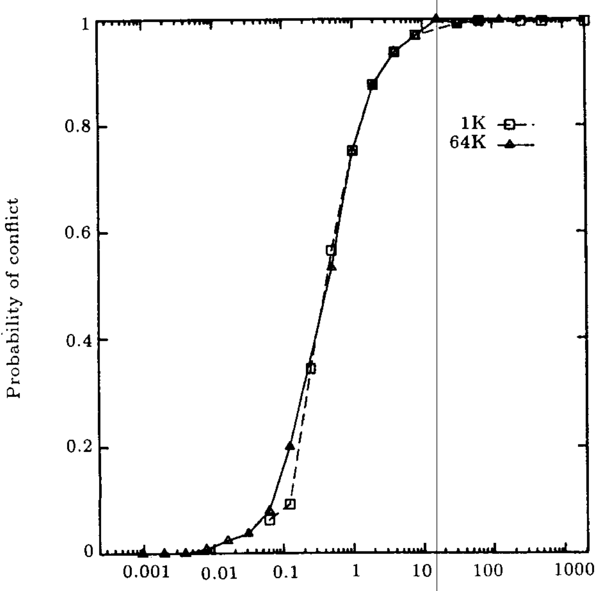

Figure 7. The probability of an address conflict for different ratios of data set size and cache size.

conflict. A more detailed analysis could estimate the probability of a conflict by fitting a normal density function to this result.

Bringing these terms together, the probability of a hit for a read access is $m_{rhit} = (1-x/L)(1-prob_{conf})$, under the assumption that the cost is amortized across all of the variables within the cache line. The probability of a write hit is directly related to the probability of a conflict, giving $m_{whit} = 1 - prob_{conf}$.

COMPARISON OF THE SIMULATION AND ANALYTICAL RESULTS

Figure 8 shows the estimated completion time of the simplex method based on T9000/C104 platform and the related simulation results. The correspondence between the two is close, considering the reasonably simple set of costs used in the performance model. Furthermore, the estimated completion time of the parallel sorting algorithm and the related simulation results (for n = 4 K) are illustrated in Figure 8. There is a good agreement between the analytical and simulation results for parallel sorting algorithm. The reason for some variation in the case of 64 processors and 4096 items is the reduced effectiveness load balancing.

CONCLUSION

This paper is concerned with the potential use of caching techniques within a multiprocessor. The ideas are based on the use of a weakly coherent shared address space, employing barrier and pairwise operations to support data consistency. This type of environment is supported by the WPRAM computational model, developed at Leeds University, and has been used as the basis for the study in this paper. The barrier caching method is based on the idea that a cache line can be in one of the three states. A change in the state occurs when a barrier operation is executed and can be carried out independently by each processor. It is a practical and highly scalable cache coherency scheme for a weakly coherent memory.

A complementary caching method called the tag caching scheme is also proposed for a form of pairwise synchronization employed by the WPRAM model. The integrated scheme allows tag synchronization between barriers, supporting more dynamic forms of parallelizm. In other words, it provides explicit support for MIMD algorithms which aim to provide efficient implementation. A study, using the simplex method and parallel sorting algorithm, has shown the practicality of this method, using simulation results based on an IMS-T9000/C104 platform. An associated performance model has been developed, which corresponds closely with the simulation studies.

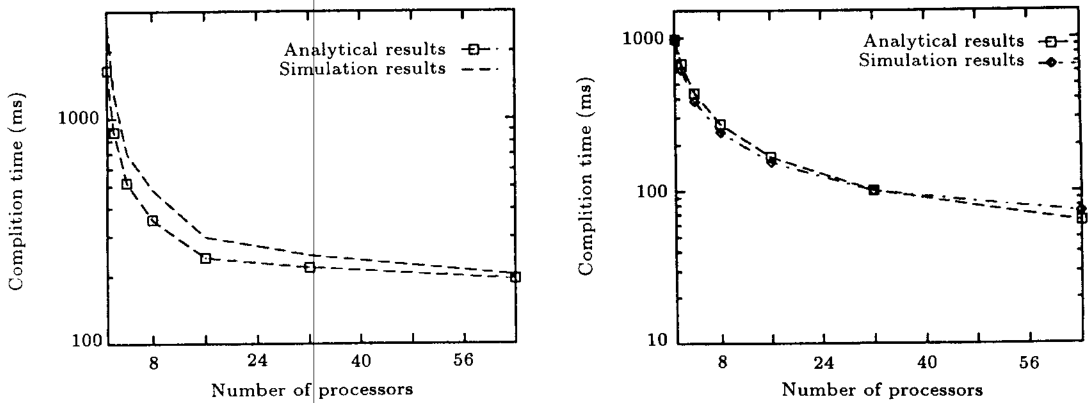

Figure 8. Simplex method and sorting algorithm

REFERENCES

1. Valian, L.J. "A bridging model for parallel computation", *Communications of the ACM*, **33**(8), pp 103-111 (1990).
2. Culler, D.E., Karp, R.M., Patterson, D.A., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R. and von Eicken, T. "LogP: Towards a realistic model of parallel computation", in *Proc. of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp 1-12, San Diego, CA (1993).
3. Nash, J.M., Dew, P.M. and Dyer, M.E. "Scalable parallel algorithm design", in *Proc. of the General Purpose Parallel Computing*, British Computer Society, Parallel Processing Specialist Group (1993).
4. Archibald, J. and Baer, J.L. "Cache coherence protocols: Evaluation using a multiprocessor simulation model", *ACM Transactions on Computer Systems*, **4**(4), pp 273-298 (1986).
5. Cheong, H. and Veidenbaum, A.V. "Compiler-directed cache management in multiprocessor", *IEEE Computer*, **23**(6), pp 39-47 (1990).
6. Dubois, M. and Scheurich, C. "Synchronisation, coherence and event ordering in multiprocessors", *IEEE Computer*, **21**(9), pp 9-21 (1988).
7. Barnaby, C. "Scalability of the cache on a bulk synchronous multiprocessor", Technical Report, INMOS Ltd, 1000 Aztec, West, Bristol, UK (1991).
8. Zamanifar, K., Nash, J.M. and Dew, P.M. "Designing scalable caching methods for the WPRAM model", In *Proc. of the First Annual CSI Computer Conference (CSICC '95)*, pp 21-28, Sharif University of Technology, Tehran, Iran (1995).
9. Min, S.L. and Baer, J.L. "A performance comparison of directory-based and timestamp-based cache coherence schemes", In *Proc. of 1990 International Conference of Parallel Processing*, pp 305-311 (1990).
10. Veidenbaum, A.V. "A compiler-assisted cache coherence solutions for multiprocessors", In *Proc. 1986 International Conference Parallel Processing*, pp 1029-1036 (1986).
11. Owicki, S. and Agarwal, A. "Evaluating the performance of software cache coherence", in *Proc. of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pp 230-242 (1989).
12. Smith, A.J. "CPU cache consistency with software support and using one time identifiers", in *Proc. of the Pacific Computer Communication'85*, K.H. Kim, K. Chon and C.V. Ramamoorthy, Eds., pp 153-161 (1985).
13. Valiant, L.J. "Bulk synchronous parallel computers", Technical Report MA 02138, Harvard University, Aiken Computation Laboratory, Cambridge, USA (1990).
14. Zamanifar, K., Nash, J.M. and Dew, P.M. "Scalable caching techniques for a weakly coherent memory", In *Proc. of Abstract Machine Models for Parallel and Distributed Computing*, M. Kara, J.R. Davy, D. Goodeve and J. Nash, Eds., pp 63-77 (1996).
15. Nash, J.M., Dyre, M.E. and Dew, P.M. "Designing practical parallel algorithms for scalable message machines", in *Proc. of the WTC'95 World Transputer Congress*, pp 529-544 (1995).
16. May, D. and Thompso, P. "Transputers and routers: Components for concurrent machines", Technical Report, INMOS Ltd, 1000 Aztec, West, Bristol, UK (1990).
17. Zamanifar, K. "An analytical performance model for a weakly coherent shared memory model", in *Proc. of the Forth Annual CSI Computer Conference (CSICC'98)*, pp 21-28, Sharif University of Technology, Tehran, Iran (1998).
18. Mehlhorn, K. and Vishkin, U. "Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories", *Acta Informatica*, **21**, pp 339-374 (1984).