CMU-CS-84-139

Dynamic Decentralized Cache Schemes for MIMD Parallel Processors

by
Larry Rudolph
Zary Segall
Computer Science Department
Carnegie-Mellon University

Abstract

This paper presents two cache schemes for a shared-memory shared bus multiprocessor. Both schemes feature decentralized consistency control and dynamic type classification of the datum cached (i.e. read-only, local, or shared). It is shown how to exploit these features to minimize the shared bus traffic. The broadcasting ability of the shared bus is used not only to signal an event but also to distribute data. In addition, by introducing a new synchronization construct, i.e. the Test-and-Test-and-Set instruction, many of the traditional parallel processing "hot spots" or bottlenecks are eliminated. Sketches of formal correctness proofs for the proposed schemes are also presented. It appears that moderately large parallel processors can be designed by employing the principles presented in this paper.

This research has been supported in part by National Science Foundation Grant MCS-8120270. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF or Carnegie-Mellon University.

Table of Contents

List of Figures

List of Tables

1. Introduction

For many years, manufacturers of high speed uniprocessors have placed a very fast buffer memory (known as a cache) between the processor and main memory in order to reduce the average memory access time. It is well known that during execution of a program most of the memory references are to a small subset of the entire address space of the program. This subset can be dynamically maintained in the cache so that most of the memory references can be quickly satisfied. In uniprocessors, without any help from the programmer, caches have routinely achieved hit ratios (the fraction of all references that are found in the cache) of about 95 percent. There is yet another, commonly agreed advantage of uniprocessor caches -- they are transparent to the user.

It is thus natural to investigate the uses of caches in configurations consisting of many processors sharing a common memory since, in such parallel processors, access to the common main memory is significantly more expensive than in uniprocessors and so the ability to reduce such access cost is even more important. Briggs and Dubois [DUB82] classify cache schemes for parallel processors as either *shared* or *private*, where shared caches are placed near the main memory, usually one per 'memory module', whereas private caches are placed near the processors, usually one per processor or set of processors. (Many others have also studied shared and private caches for various multiprocessor architectures. See, for example, [BEL74], [DUB81], [TAN76], [CEN76], [BRI81], [SMI82], [GOO83], [FRA84].) In this paper, we are concerned only with private caches and their corresponding schemes. It is worth noting that in the rest of the paper we will use the terms private and shared in a different context, as explained below.

We assume a configuration in which the processors perform all their accesses through the cache. In such systems, it is natural to divide data into two classes: *local* (which we sometimes refer to as private) and *shared*. It is also useful to further subdivide these into *read-only* (e.g. code) and *read/write* classes. Note that variables need not be statically classified, e.g. for some period of program execution a data item can be read/write and for some other period it can be read-only.

To motivate our cache schemes consider the data gathered from the execution of parallel programs on the CM*, a parallel processor consisting of 50 processors. Limited cache emulation experiments were performed to ascertain the fraction of references satisfied by the cache. In these experiments, only code and local data were considered cachable and a write-through policy was adopted for local data. Thus writes to local data were counted as cache misses since they caused communication external to the processor/cache. All references to shared (non-code) data also caused a cache miss. Table 1-1 generated by Raskin [RAS78] shows the results for two applications and various cache sizes. The figures indicate the fraction of the total number of references that caused cache misses

classified according to the operation. Note that for the largest cache size, which is small by today's standards, most read references succeeded in finding their value in the cache and that these constituted most of all the references. The figure of 6% read misses is roughly close to that measured on uniprocessors and could be reduced further by employing a larger cache. Although most references are to read-only data, references to shared data still account for 5 to 10% of all memory references. For good overall performance, many of the shared memory references must be satisfied by the private cache.

Cache Size (set size 1 word)	Read Miss Ratio	Local Writes	Shared Read/Write	Total Miss Ratio
256	26.1	8	5	39.1
	25	6.7	10	41.7
512	21.7	8	5	34.7
	28.8	6.7	10	37.5
1024	11.3	8	5	24.3
	10.8	6.7	10	27.5
2048	6.1	8	5	19.1
	5.8	6.7	10	22.5

Table 1-1: Cm* Emulated Cache Results

In this paper we present a cache scheme that, in addition to handling read-only code and private variables, handles shared memory references. At first blush, it seems that one cannot gain much by caching shared data. Upon closer examination, one finds numerous situations in which this is not so. Many references to shared data are for synchronization (i.e. testing and setting locks) and communication (i.e. storing information in shared memory to which many other processes later refer). Under our cache scheme, these situations cause a minimum of external traffic.

Consider another situation in which caching shared data is possible. Since it is assumed that shared variables can be modified at any time, compilers are careful not to leave the contents of such variables in registers for long periods of time.[1] Thus, repeated reference to shared variables cause repeated memory accesses. Often these variables are not modified between fetches and so for short periods of time shared variables act like local variables and, in principle, can be cached. The programmer cannot always detect these situations; indeed, they can depend solely on the timing of the various processors or on the input data. Our cache scheme elegantly processes such situations

[1] A classical error encountered by many novice users of CM* is to busy-wait on a shared variable without explicitly telling the Bliss compiler to refetch the value before each test.

and thereby the performance of the parallel processor can be improved.

The cache schemes presented in this paper are transparent to the user and hence data is not required to be pre-tagged (e.g. as shared read/write). Moreover, since transparency implies consistency, data modified by one processor must be immediately detected by all the other processors and it is the case that a processor always reads the 'latest' value stored (i.e. written by some processor). Ensuring consistency in a decentralized way is very tricky; we formally prove that our scheme satisfies this property.

Although many types of parallel processors have been proposed we concentrate on just one 'generic' type - a collection of processing elements (PE's) connected to a set of memories. We assume that the PE's and memories are connected by a *logically* single bus (although physically this may be a set of buses), allowing for inexpensive broadcasting. In such a design, it is clear that the bus is the critical resource and its use must be carefully limited. The more successful the cache scheme is in reducing bus traffic, the easier it is for the bus to accommodate large numbers of processors . We realize that the use of a single shared bus has limited applicability, and therefore show in section 7 how to adopt our scheme to multiple bus configurations accommodating larger numbers of processors.

Our scheme is in many ways an extension of the one presented by Goodman [GOO83]. We assume the same underlying architecture and exploit some of the same abilities of the shared bus, although we show how to use bus broadcast capability more efficiently and how to extend our results to a multiple bus architecture. The Goodman scheme may be classified as 'event broadcasting', whereas in our proposed schemes events and data values are broadcast. As will be seen, we assume that each address line in the cache is tagged, however, we can do even more with fewer tags. Moreover, the architecture is further exploited to optimize processing of traditional 'hot spots' or bottlenecks that arise in parallel processing. We also believe that the formal proofs of multiple cache consistency validates their behavior.

The structure of the paper is as follows: the first section explicitly states our assumptions concerning both the pattern of data references as well as the underlying machine architecture and cache capabilities. The main results, two cache schemes, are then presented along with some analysis of their functioning. The schemes can be distinguished by their assumptions concerning the broadcast abilities of the machine. Consideration of synchronization is presented in Section 6 and ways to increase the shared bus bandwidth are shown in Section 7.

2. Assumptions

Before describing the cache schemes it is important to clearly state our assumptions. We first review our assumptions concerning the average pattern of use of data in a parallel processor as described in the introduction:

1. Each data item is referenced more often with a read operation than with a write operation.
2. References to local data and to read-only shared data are more frequent than to read/write shared data.
3. Many shared variables act like local variables for moderately long periods of execution time.

The first assumption indicates that it may be desirable to increase the overhead of the write operation in order to optimize the read operations. The latter two assumptions imply that one should optimize references to local and read-only data.

In regards to the underlying architecture, we make the following assumptions, all of which can be satisfied by existing technologies:

1. There is a *logically* single bus connecting the n PE's and I/O with memory.
2. There is a bus arbitrator that allocates access to the bus.
3. A cache is associated with each PE and communicates both with the PE and with the shared bus.
4. The caches can "listen" to the bus activity and detect the referenced address, the activity (read or write), and the data.
5. The bus cycle time is no faster than the cache cycle time. Thus each cache has time to monitor the bus and take appropriate action before the next bus cycle. Similarly, the PE cycle time should[2] be no faster than the cache cycle time so that the PE does not spend time waiting for the cache to respond.
6. Each cache has the ability to interrupt (i.e. kill) the current bus activity and to replace it with one of its own. The cache is fast enough to first observe a bus action and to then interrupt it.
7. A direct-mapping cache with a one word blocksize is assumed.

Our choice of set size and block size of one has two motivations. First, a high cache hit ratio may not always result in good performance. For example if block size is a page then every cache miss,

---

[2] It need not be.

although infrequent, takes a very long time. Secondly, shared data appears to have different, if any, notions of locality. There is no reason to suspect that nearby address of shared variables will be used by the same processor at the same time. Moreover, the larger the cache size the less important the block size, even for code and local data.

3. The RB Cache Scheme

In this section we describe the workings of the RB cache scheme whose name is derived from the fact that values fetched in response to certain CPU reads are broadcast to all of the caches. We finesse the problem of maintaining read/write shared data in the private caches of the processors by dynamically redefining such data in terms of the other two classes of data. Since concurrent read or write operations to the same data word are serialized (enforced by the use of a shared bus), whenever there is a write to a read/write shared data item, we consider that data item to be a variable local to the PE executing the write. Whenever a read/write shared data item is read, we consider it to be a read-only shared data item. Our goal is to have the caches dynamically decide these properties in a decentralized fashion so that little additional processing overhead is incurred.

Since cache behavior will depend on the current status of the data item, we associate with each address line a set of tag bits indicating the state of the data item associated with that address line. We shall concentrate on a single address line (sometimes referred to as a physical address or a variable) and its corresponding entry in the caches, so that when we say the cache is in the invalid state (or state I), we mean that the cache entry for this address is tagged as invalid. Each PE interacts with its associated cache by issuing a read or a write. Such actions cause the cache to either respond to the PE immediately or cause the cache to first perform a bus read or bus write and, upon successful completion, respond to the PE. Moreover, the cache constantly monitors the bus and as a result may change its state and/or the value of a cache entry.

We begin with a scheme consisting of three states for each address line. The states are 'readable' (R), 'invalid' (I), and 'local' (L). The meaning of the invalid state should be self-evident: the data in the cache is assumed to be incorrect and thus any reference to it will cause a corresponding bus action[3]. Similarly, a readable state means that the data in the cache is valid and consistent with main memory, and can be read immediately from the cache. The local state means that the data can be read or

[3] A read will cause the data to be fetched from the global memory and a write causes data to be stored into shared memory and at the same time to be broadcast to all other caches. This latter action can be accomplished by a bus write to the global memory thereby eliminating the need for a special signal.

written locally causing no bus activity.[4]

We call the collection of cache states for a particular address[5] a *configuration*. In the RB scheme, for each address (and ignoring caches not contain the address) there are only two types of configurations. A variable local to PE_i will be in state L in cache i and in state I in any other cache containing this variable; we call this set of states the *local configuration*. A shared read-only variable is in state R in all caches containing it; we call this the *shared configuration*. Our scheme dynamically assigns these configurations.

We describe the functioning in a case by case manner. First assume the variable, say x, is contained in all caches and recall that we say cache i is in state R to mean that the address line for the physical address representing variable x is tagged with an 'R'.

(i) Let x be in the shared configuration. A read simply fetches the cached value; no bus activity is generated. A write by PE_i to variable x causes the value to be updated in cache i as well as a broadcast of a bus write. The bus write updates the memory and at the same time causes all other caches to change into state I.

(ii) Let x be in a local configuration with cache i in the local state. There are two subcases to consider.

a. X is referenced by PE_i. A read of x by PE_i simply fetches the cached value and a write to x just updates the cached value; there is no bus activity generated.

b. Now consider x referenced by PE_j, $j \neq i$. A write updates the cached value in cache j, changes its state to L, and generates a bus write to x with the new value. This bus write causes all other caches to be in state I. The only tricky situation is when PE_j reads x. Since x is in state I in cache j, a bus read is issued which is "seen" by all the other caches. Without intervention, the bus read will fetch the value stored in shared the memory, however, cache i, which is in state L, interrupts the bus read and performs its own bus write updating memory to the correct value. The original bus read will be retried immediately. In addition to fetching the correct value, this bus read is noticed by all the caches which then read the value returned from the read, cache it, and change into state R.

Since the cache size is smaller than the memory size, there must be some mechanism for replacing old cache entries with newly referenced ones. The exact choice of a replacement policy is orthogonal to our scheme. We need only describe the actions to be taken by the cache when it decides to overwrite an old entry. Only those overwritten items that are tagged local need to be written back to the memory. A reference to an item not in the cache behaves exactly as if it were in

---

[4] Implied is that the cache copy is inconsistent with the memory and with all other caches.

[5] We use the terms address, variable, and data item interchangeably. They all refer to a single word of shared memory.

the invalid state; a write (or a read) results in a bus write (or a bus read) and the state is changed to local (or readable).

A few final points need to be addressed. As explained, bus writes caused by write-backs are the only ones that need to update memory, however, for ease of implementation all cache writes should do so. Since shared variables are maintained within the cache, we must ensure consistency when they are referenced by "nonsimple" read or write operations. In particular, we note that read-modify-write operations fit quite well into our scheme. The initial 'read with lock' does not reference the value in the cache; it causes a 'read with lock' bus operation and as a side effect causes all other caches to enter the read state. Any bus writes before the unlock will fail and so it does not matter what the other caches do until the write-with-unlock is broadcast over the bus. This action then sets all the other caches into the invalid state, i.e. a local configuration is assumed.

We make the informal description precise by describing the state transitions for a cache. Once again we are only concerned with a single address line and talk about a cache being in a particular state when we mean that the address of concern in the cache is tagged with this state.

- The cache is in the Read state: In response to a CPU read, the cached value is returned to the processor and in response to a CPU write three things happen: a bus write is generated (this informs the other caches that the variable is now considered local), the cache is updated with the new value, and the cache is tagged as Local (state L). A bus read (generated by some other cache) has no effect on a cache in state R, however, a bus write causes the cache to change its state to Invalid.

- The cache is in the Invalid state: In response to a CPU read, the cache generates a bus read and upon successful completion of the bus read, the data returned is stored in the cache (as well as being returned to the processor), and the cache state is changed to Read. In response to a CPU write similar actions occur: a bus write is generated (informing all other caches of this fact), the cache value is updated to this new value, and the cache state is set to Local. In response to a bus write, a cache in the invalid state will do nothing, whereas, in response to a bus read the following occurs: the value returned in response to the read is stored into the cache and the cache state is changed to Read. (Note that all caches that contain the target address of a bus read, will perform these actions, so that the value read will, in effect, be broadcast to all the processors for future use.)

- The cache is in the Local state: In response to a CPU read, the cached value is returned to the processor and in response to a CPU write, the value in the cache is updated to this new value (no bus activity is generated). Bus writes cause a cache in the local state to change its state to Invalid, whereas, bus reads cause the following actions to occur: The bus read is interrupted and replaced by a bus write of the cached value. The cache state is changed to Read. (The interrupted bus read will be retried on the next cycle causing the new value to be broadcast to all the caches.)

Figure 3-1 details the state transition diagram corresponding to this explanation. The transition lines are marked with the action causing the transition as well as being marked with a possible modifier indicating some action taken by the cache during the transition.

4. Proof of Consistency - Sketch

A cache scheme for a parallel processor is *consistent* if a read by a processor will always fetch the "latest" value written. This section first defines the term "latest" and then shows why the RB cache scheme is consistent.

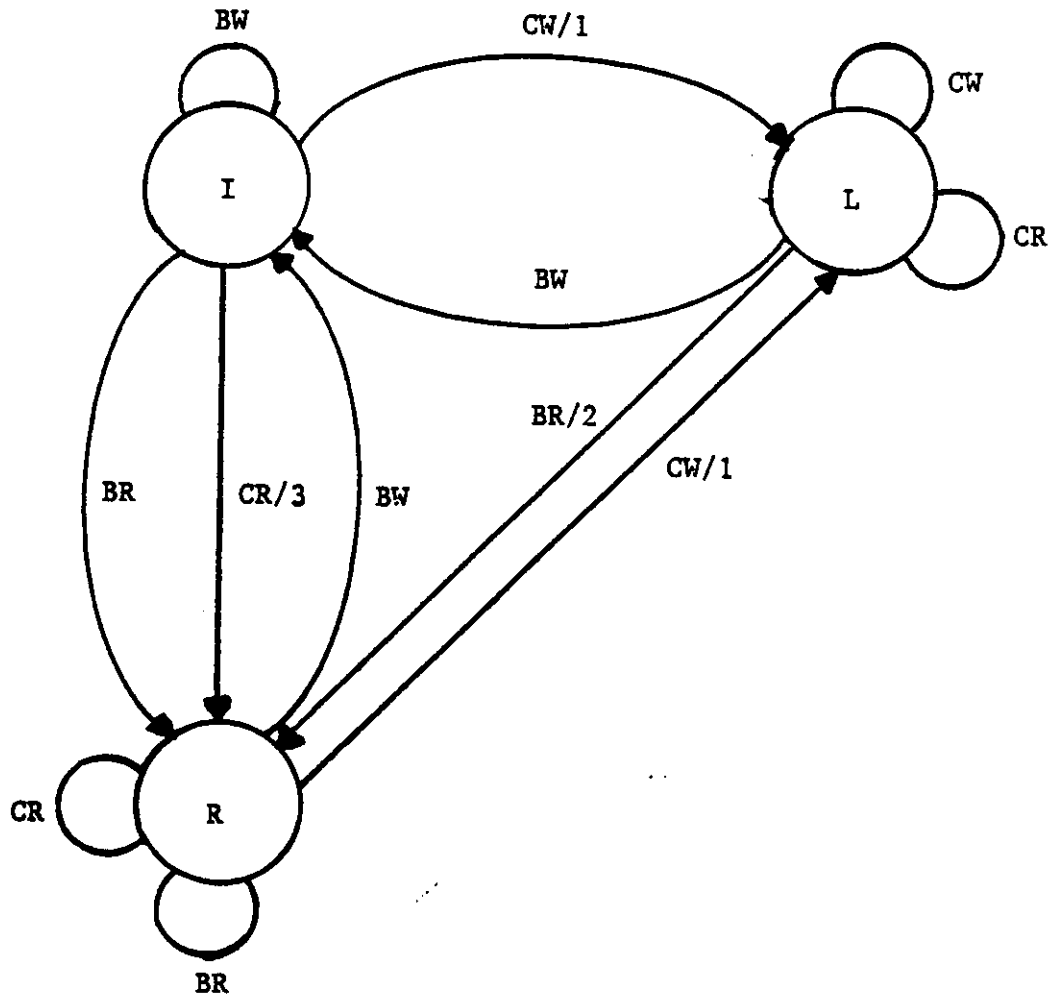
We first review our assumptions. There are N processors connected to N caches. The caches are connected to the memory via a logically single shared bus. In order to define the notion of "latest" value one need only define a serial execution order of all the instructions processed by the machine that is consistent with the parallel executions of the PE's. That is, for each parallel execution of the machine, there must exist a corresponding serial execution of the instructions that produces the same results. The notion of latest value can then easily be defined in terms of this virtual serial execution.

The following shows how to construct one such serial execution order. Assume that each processor can execute a maximum of P_c instructions during each bus cycle. Thus, after t bus cycles, at most $(P_c)(N)(t)$ instructions can be executed. Let $c_{i,t}$ be a number between 0 and P_c-1 used to indicate the cycle number of PE_i after the t^{th} bus cycle (thus, PE_i could have executed up to $(P_c)(t)+c_{i,t}$ instructions by time $c_{i,t}$). Given a parallel execution of T bus cycles we define a serial execution of length at most $(P_c)(N)(T)$. The instruction executed by PE_i, $c_{i,t}$ cycles after the t^{th} bus cycle is given position $(P_c)(N)(t)+(P_c)(t)+c_{i,t}$ in the serial order. If the PE is waiting on the bus or cache, or still completing the instruction begun earlier, then a no-op is assumed. It should be clear that such a serial execution order corresponds to the parallel execution since one processor can only have an effect on another through bus activity.

In the previous section, the cache scheme was described in terms of local and shared configurations. It is important to show that these are indeed the only possible configurations attainable and, in particular, that it is not possible for the same address to be tagged as local in more than one cache at the same time. This is made precise in the following:

Lemma: For each address, the following holds:

- At the start of each bus cycle, for each address, one of the following is true:
    - The address is in the shared configuration: if the address is in the cache then it is in



Legend

CW - CPU Write Request
CR - CPU Read Request
BW - Bus Write Request
BR - Bus Read Request

Modifiers

1 - Generate a BW (i.e. write through)
2 - Interrupt BR and supply the data from the cache
3 - Generate a BR (i.e. cache miss)

Figure 3-1: State Transition Diagram for each Cache Entry for the RB Scheme

state R.

- The address is in the local configuration: the address is in state L in at most one cache and is in state I for all other caches that contain it.

- The latest value written is contained either in some cache that is in state L or else in any cache that contains this variable.

Proof: We will actually only sketch the proof, a full formal version follows directly but adds little to the understanding. Assume that the caches contain the entire address space so that the issue of overwrites can be ignored for the moment. Also, consider only actions associated with a single address (a reasonable assumption since actions involving other addresses are disconnected).

Each cache can be considered a finite state automaton defined by the state diagram in Figure 1. For each value of N (the number of processors), define a *product machine*, M, as the collection of the N finite state automata plus one more to represent the function of the common memory. The validity of the lemma follows directly from the direct examination of this product machine.

The caches will be numbered between 1 and N, and the common memory can be viewed as yet another cache (although somewhat special) which is referred to as number 0. Initially, the memory will be tagged with an L, indicating that the only correct value of the address is contained in t he memory, and all caches tagged as I. Thus, one state of the product machine (the initial state) can be written as $L_0 I_1 \cdots I_N$, where X_i means cache i is tagged with an X (= L,I, or R).

All states reachable from the initial state are now described. Note that these states are identical to the local and shared configurations. We use the notation A:S1 =>S2 to mean that action A (either a cpu read or write) causes the product machine to change from state S1 to state S2.

cpu read_i : $L_0 I_1 \cdots I_N \Rightarrow R_0 \cdots R_N$
cpu write_i : $\Rightarrow I_0 \cdots I_{i-1} L_i I_{i+1} \cdots I_N$

cpu read_j : $I_0 \cdots I_{i-1} L_i I_{i+1} \cdots I_N \Rightarrow R_0 \cdots R_N$
cpu write_j : $\Rightarrow I_0 \cdots I_{j-1} L_j I_{j+1} \cdots I_N$

cpu read : $R_0 \cdots R_N \Rightarrow R_0 \cdots R_N$
cpu write_i : $\Rightarrow I_0 \cdots I_{i-1} L_i I_{i+1} \cdots I_N$

The product machine can be easily extended to include overwrites in a straightforward manner. Each address is still assumed to be in the cache represented by the finite state automaton, however, the automaton can be extended to include a state NP (not present) with the transitions to and from

that state added in the natural way. An overwrite of an address tagged I or R can be modeled by the product machine as simply setting the tag of the address to NP. An overwrite of an address tagged L requires that the address be 'written back' to the memory. This is modeled by having the memory (cache 0) perform a cpu write of this value. The overwritten address in the cache will also be tagged NP and the memory tagged L.

We can now present our consistency theorem.

Theorem: Each PE always reads the latest value written.

Proof There are various cases to consider based on the class of the variable, local or shared, and upon the previous pattern of use of a shared variable. Assume PE_i is to read variable X.

Case X is local variable: This is the easiest case. Since local, the latest value of X is the one written by PE_i. The value fetched will either be from the i^{th} cache or from memory. The cache behaves just like a uniprocessor cache.

Case X is shared and last written by PE_i: In this case it should also be clear that the latest value is fetched. By the lemma, at most one cache can be in local state and that cache contains the latest value. If some other PE has read the value in the meantime, the i^{th} cache will be in the read state. In any event, the latest written value is either in the cache or has been evicted from the cache and is thus correctly fetched from memory.

Case X is shared and last written by PE_j ($j \neq i$):

Subcase PE_i is first non PE_j to read x: The read will cause a bus read and the value will be fetched from that cache. Since cache i will then enter state R, so will all the other caches.

Subcase PE_i reads x after other PE's read x: By the previous subcase, all the caches that contain the variable will contain its latest value. □

In the next section, a Read-Write Broadcast scheme will be introduced. A proof of its correctness is a straightforward adaptation of the previous proof.

5. The RWB Cache Scheme

In the RB scheme, the caches note the occurrence of bus reads and bus writes as well as the data returned in response to bus reads, whereas in this section, an improved scheme, the RWB scheme, is presented in which the caches also note the data part of the bus writes. The result is that the RWB scheme has improved performance, although, at the extra cost of a new type of bus action and an additional state. The broadcasting abilities of the shared bus are further exploited to optimize certain patterns of references common to parallel programs. Although these reference patterns only represent a small fraction of all memory references, a large portion of the PE's are often concurrently involved and so efficient processing is important.

The RWB scheme differs from the RB scheme in the way in which the caches change from the local configuration to the shared configuration. Assume a variable X is in the local configuration and that PE_i was the last to modify its value. In the RB scheme, a read by some PE_j ($j \neq i$) was the only operation that caused X to change to the shared configuration. We now propose that this configuration change occur whenever a PE_j ($j \neq i$) references X (including a write to X).

The motivation is as follows: Variables are initially assumed to be in the local configuration and the first write will cause a change to the shared configuration. Only when a variable is used exclusively by one PE does it reenter the local configuration[6]. Consider the initialization of an array that is much too large to fit in a cache. Under the RB scheme, there would be two bus writes for each item; one for the first CPU write initializing the element and one again later as a writeback when the addressline is reused (i.e. and overwrite). In RWB, there will be only one bus write per item.

Many shared variables tend to be referenced in the cyclical pattern: written by some one PE and then read by others. In such cases, the bus write caused by a PE writing to a variable in the shared configuration simply broadcasts the new value to all interested caches. Subsequent read references will cause no bus activity. A case in point are the read-modify-write operations. Upon completion of such operations, the RWB scheme will leave the caches in a shared configuration so that subsequent reads cause no bus activity. Finally, the RWB scheme allows for a more robust memory management; if the value of a variable is corrupted while in memory or in some cache, there is a higher probability that some cache contains a correct copy.

---

[6]There is considerable latitude in deciding when a variable is assumed to be consider local to a PE. For expository purposes, we assume that two writes to a variable with out any intervening references to the variable by any other PE is enough to indicate local usage. Straightforward modifications are possible if one wishes at least k uninterrupted writes to indicate local usage.

The details are as follows: A new state called first-write (state F) is added, as well as a new bus signal called invalidate (i.e. BI). The first write to a variable, say by PE_i, in shared configuration causes all caches to remain in state R except for the i^{th} cache that goes into state F. A subsequent write by PE_i then confirms the fact that the variable is to be assumed local. Cache i enters state L and broadcasts an invalidate signal causing all other caches to enter state I (i.e., assume a local configuration). While still in this intermediate configuration (cache i in state F and the rest in state R), all reads have no configuration effect and data can be fetched from any cache. A write by some other PE_j will cause cache j to change to state F and cause a bus write to occur. The data written is read by all caches and they in turn enter state R. Figure 5-1 makes this precise.

A read causing a cache miss, will in turn generate a bus read. If the variable is in the local configuration then the actions are identical to the previous scheme. All other configurations will be unchanged. On the other hand, a bus write caused by a cache miss will be treated as above causing all other caches to assume state R and this cache state F.

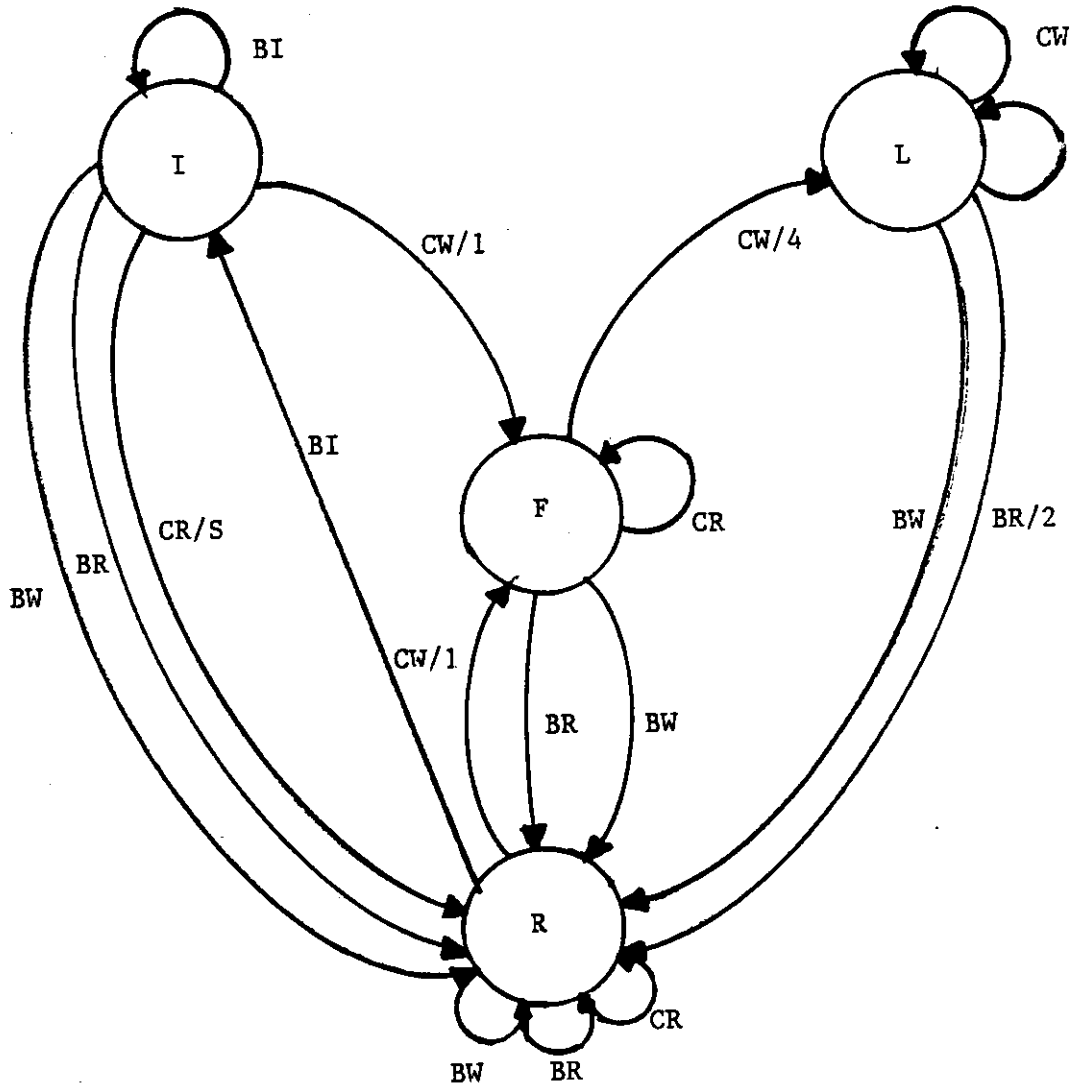
The special bus invalidate signal can be implemented by reserving one value from the range of values assumed by any data word to indicate an invalidate signal.

6. Synchronization Using Caches

The behavior of a parallel computation can be characterized as a series of parallel actions alternated by phases of communication and/or synchronization. Although some of the parallel actions may cause resource contention (e.g. access the same memory module), the use of large private caches as well as appropriately designed connection hardware alleviates most of these problems. Unfortunately, the steady-state assumptions concerning memory traffic and resource contention often are greatly disturbed by the phases of synchronization (also know as parallel computation "hot spots"). It is desirable to eliminate, or at least reduce, the effects of the hot spots.

Although there have been many types of synchronization primitives proposed for parallel processing (see, for example [Rud81]), we concentrate on the classic test-and-set primitive (TS). The standard parallel computer implementation of the test-and-set instruction generates a read-modify-write operation between processing element and the shared memory. That is, a special bus read operation is generated that locks the appropriate shared memory location[7], returns the value back to the PE, the PE then performs some operation on the value that may modify it, and finally the modified

[7] In some implementations all of memory is locked, in others only sections of memory. It is generally considered too expensive to associate a lock with each memory address.



Legend

CW - CPU Write Request
CR - CPU Read Request
BW - Bus Write Request
BR - Bus Read Request
BI - Bus Invalidate Request

Modifiers

1 - Generate a BW (i.e. write through)
2 - Interrupt BR and supply the data from the cache
3 - Generate a BR (i.e. cache miss)
4 - Generate a BI

Figure 5-1: State Transition Diagram for each Cache Entry for the RWB Scheme

value is stored back into the shared memory cell and the lock removed. Accordingly, if many PE's simultaneously test-and-set the same memory location, as in the case of a lock for shared data structure, high bus traffic and memory contention will result. We show how our cache scheme can reduce these ill effects.

The analysis of bus traffic may consist of two components. One is the bus traffic generated by successful synchronization operation (i.e. getting a lock) and the second is the traffic generated by unsuccessful (spinning and waiting) operations. The latter category introduces an unnecessary burden on the bus traffic and memory contention. We propose a variation of the traditional synchronization primitive, called a test-and-test-and-set (TTS). Recall that a test-and-set instruction is an atomic operation on a variable with the following implementation of the test and set semantic:

```
Test-and-set  V  to  X:
     If  V = 0 Then V := X
               Else nil
```

The test-and-test-and-set operation could be seen as a combination of a test instruction followed by a test-and-set instruction and can be described as follows:

```
Test-and-Test-and-Set   V to X:
     If V = 0 Then nil
     Else
       IF  V = 0 Then  V := X
                 Else nil
```

The advantage of the test-and-test-and-set is that a read-modify-write cycle will be generated only when the initial test has succeeded. The bus traffic component connected with unsuccessful synchronization attempts can therefore be minimized. Indeed, the initial test part of the instruction could be executed in the local cache, without generating bus traffic. Only when the test succeeded (indicating a good chance that the lock will be free) will the test-and-set part be executed.

The test-and-test-and-set can be implemented in hardware by creating a new primitive instruction, or alternatively, in software by preceding each test-and-set instruction with a simple test instruction We consider this latter possibility as a definite advantage since it enables the use of off-the-shelf processors. Much existing software could be used with minor modifications.

Next, we will consider the implication of the proposed cache consistency schemes to the elimination of hot spots. We assume off-the-shelf processing elements and hence we will use the software implementation of test-and-test-and-set class of instructions.

6.1. Synchronization Using RB Scheme

In order to study the behavior of the RB cache scheme we will use an example of synchronization between M processes (1 process per processor) using a shared data structure lock S. The lock S is 1 if the data structure is currently reserved for use by some process and is 0 if the data structure is not in use.

Figure 6-1 demonstrates the cache behavior when using test-and-set instruction. The first *m* columns relate to the *m* caches. Each entry in these columns indicates the state (R,L, or I) and the value (0, 1, or -) that is in the cache for the lock S. The $m + 1^{st}$ column gives the value of S in memory. Each row represents the situate at a point in time with successive rows representing the progression over time. Note that an attempt to get the lock via a test-and-set instruction causes a read-modify-write cycle. We treat such an instruction as a non-cachable read when the test-and-set fails and as a write when it succeeds.

P_1 Cache	P_2 Cache		P_m Cache	S	Observation
R(0)	R(0)	- - -	R(0)	0	Initial State
I(-)	L(1)	- - -	I(-)	1	P_2 Locks S
R(1)	R(1)	- - -	R(1)	1	Others try to get S
.	.		.		(Bus Traffic)
.	.		.		
.	.		.		
I(-)	L(0)	- - -	I(-)	0	P_2 releases S
L(1)	I(-)	- - -	I(-)	1	P_1 get the S
R(1)	R(1)	- - -	R(1)	1	Others try to get S

Figure 6-1: Synchronization with Test-and-Set for RB Scheme

Note that during the period when the lock is reserved for P_2 all attempts by other processors to reserve the lock will be unsuccessful and will continuously generate bus traffic and memory contention.

The use of the TTS primitive eliminates much of the bus traffic during the period when the lock is reserved for PE_2. Figure 6-2 demonstrates the states resulting when TTS is used in place of TS.

Note in this case the unsuccessful attempts to get the lock are spins in the cache and do not

P_1 Cache	P_2 Cache		P_m Cache	S	Observation
R(0)	R(0)	- - -	R(0)	0	Initial State
I(-)	L(1)	- - -	I(-)	1	P_2 locks S
R(1)	R(1)	- - -	R(1)	1	Others try to get S
.			.		(No Bus Traffic)
.			.		(Load from Caches)
.			.		
.			.		
I(-)	L(0)	- - -	I(-)	0	P_2 releases S
R(0)	R(0)	- - -	R(0)	0	A Bus Read to S
L(1)	I(-)	- - -	I(-)	1	P_1 get the S
R(1)	R(1)	- - -	R(1)	1	Others try to get S

Figure 6-2: Synchronization with Test-and-Test-and-Set for RB Scheme

generate bus traffic.

6.2. Synchronization Using RWB Scheme

We now consider the cache synchronization behavior for the RWB scheme and use the same example as above. Again the TTS primitive will be used and Figure 6-3 presents the results.

When compared to RB scheme note the substantial minimization of cache invalidation, along with the reduction of bus traffic due to unsuccessful synchronization attempts. Further examples of the RWB scheme can be found in [RUD84].

7. Shared Bus Bandwidth

In a multiprocessor, even one using our cache schemes, bus utilization increases with increasing numbers of processors. Eventually, bus saturation occurs. In this section, we try to estimate the bus bandwidth required to avoid saturation for a given number of processors. We analyze bus traffic and concentrate on the average case since the main purpose of caches is to improve average case performance; the analysis will not consider peak values.

Assume the following notation:

x - number of accesses per second generated by a processor in Million Accesses per Second (MACS)

P_1 Cache	P_2 Cache		P_m Cache	S	Observation
R(0)	R(0)	- - -	R(0)	0	Initial State
R(1)	F(1)	- - -	R(1)	1	P_2 locks S
.			.		Others try to get S
.			.		(No Bus Traffic)
.			.		(Load from Caches)
.			.		
I(-)	L(0)	- - -	I(-)	0	P_2 releases S
R(0)	R(0)	- - -	R(0)	0	A Bus Read to S
F(1)	R(1)	- - -	R(1)	1	P_1 get the S
.			.		
.			.		Others try to get S
.			.		

Figure 6-3: Synchronization with Test-and-Test-and-Set for RWB Scheme

1/h - the cache miss ratio

m - number of processors on the shared bus

Then the shared bus bandwidth (SBB) satisfies the following:

$$SBB > mx/h$$

Consider the following example:

1/h = 10%

n = 128

x = 1 MACS

=> SBB = 12.8 MACS

If it is not possible to build a shared bus capable of accommodating the desired shared bus traffic (i.e. SBB is too large), then a multiple shared bus could be employed. The question is then, how to extend our proposed cache schemes to function correctly in a multiple shared bus configuration.

Figure 7-1 shows a dual shared bus configuration. The private caches and the shared memory are divided into two memory banks using the least significant address bit. Each part of the divided cache will generate, on average, half of the traffic that would otherwise be produced by an undivided cache.

Hence, the required bandwidth for each shared bus will be about half. Using this method, the cache scheme proposed could easily be extended for a small number of multiple shared buses. Initial evaluation shows that by using the proposed cache schemes, relatively large parallel processors having as many as 32 to 256 processors could be economically built using existing off-the shelf components and today's technology.

8. Conclusion

We have presented two cache schemes for shared memory parallel processors. Their main feature is that local and read-only shared variables are dynamically detected and classified to allow the caches maintain consistency while greatly reducing memory access time. A second feature is the use of the broadcasting ability of a shared bus to not only signal an event but to distribute the data. It appears that moderately large parallel processors can be designed in this fashion.

Our scheme exploits the ability of the caches to 'listen' to the bus traffic, hence, our ideas may not be applicable to other parallel architectures. In particular, it is not clear how they can be applied to architectures consisting of a set of processors connected to a set of memories via a logarithmic interconnection network. The advantages of our scheme (e.g. complete transparency to the user), however, make it desirable to explore the possibility of applying it to other architectures.

Two questions raised in our presentation appear to be promising for further research. The first is how to extend our scheme to hierarchical structures more amiable to large scale parallel processing. The second is the exploitation of replicated values in the various caches to improve the reliability of the memory.

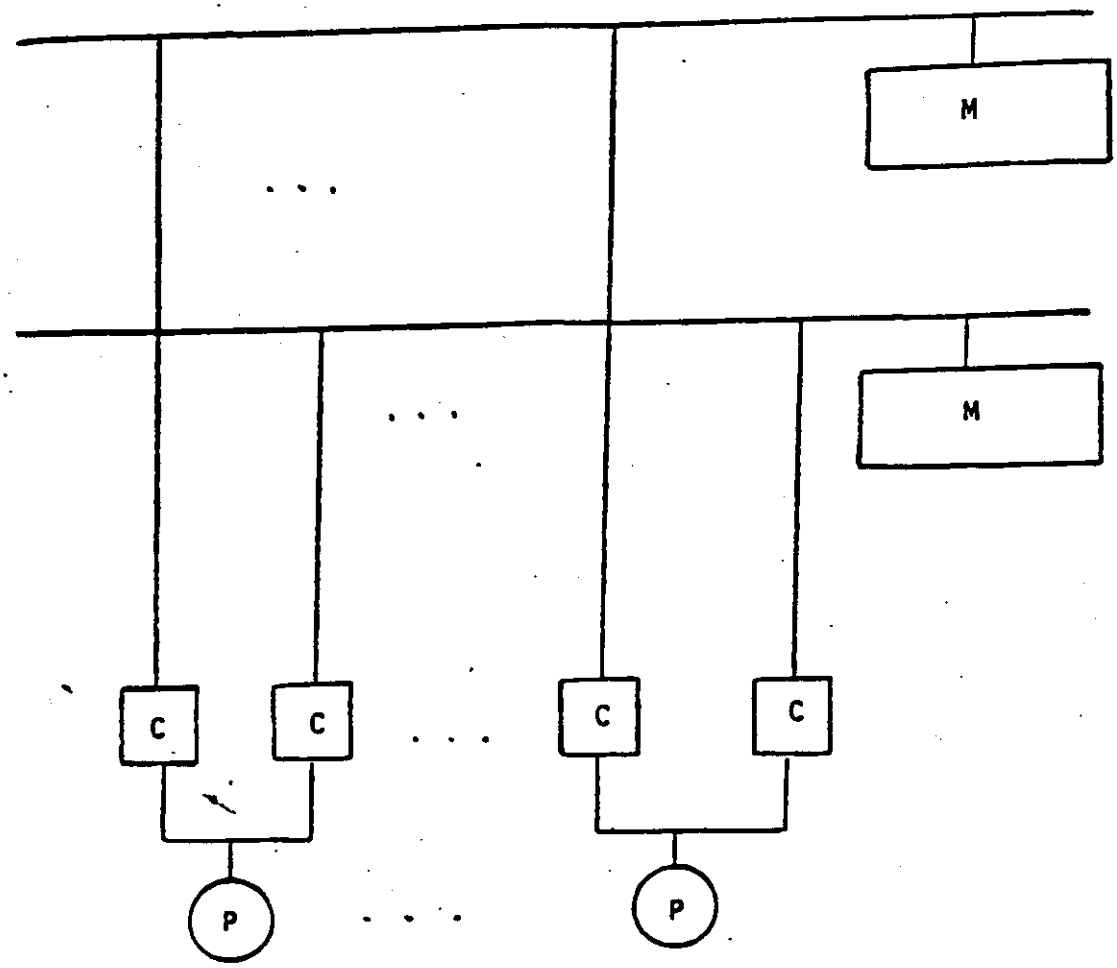


Figure 7-1: Multiple Shared Bus Cached Based Parallel Processor

References

[BEL74] J.D. Bell et al., "An Investigation of Alternative Cache Organizations," IEEE Transactions on Computers, Volume C-23, No. 4, April 1974.

[BRI81] F.A. Briggs, M. Dubois, K. Hwang, "Throughput Analysis and Configuration Design of Shared-Resource Multiprocessor Systems: PUMPS," The 8th Symposium on Computer Architure, 1981.

[CEN78] L.M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," IEEE Transaction on Computers, Vol., C-27, No. 12, December 1978.

[DUB81] M. Dubois and F.A. Briggs, "Efficient Interprocessor Communication for MIMD Multiprocessor Systems," Proceedings of the 8th International Symposium on Computer Architecture", May 1981.

[DUB82] M. Dubois and F.A. Briggs, "Effects of Cache Coherency in Multiprocessors," The 8th Annual Symposium on Computer Architecture, 1982.

[FRA84] S. J. Frank, "Tightly coupled multiprocessor system speeds memory-access times," Electronics, Vol. 57, No. 1, Jan. 1984, pp. 164.

[GOO83] J.R. Goodman, "Using Cache Memory to Reduce Processor - Memory Traffic," 10th International Symposium on Computer Architecture, 1983.

[GOT83] A. Gottlieb, R. Grishman, C.P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," IEEE Transactions on Computers. Volume C-32, February 1983.

[RAS78] L. Raskin, "Performance Evaluation of Multiprocessor Systems," Ph.D. Thesis, Carnegie-Mellon University, August 1978.

[RUD81] L.S. Rudolph, "Software Structures for Ultraparallel Computing," Ph.D. Thesis New York University, December 1981.

[RUD84] L.S. Rudolph, "Executing Systolic Arrays by MIMD Multiprocessors," Department of Computer Science, Carnegie-Mellon University, 1984.

[SMI82] A.J. Smith, "Cache Memories," Computing Surveys, September 1982.

[TAN76] C.K. Tang, "Cache System Design in the Tightly Coupled Multiprocessor System," Proceedings of the AFIPS, 1976.