IEEE Transactions on Parallel and Distributed Systems, 3(1):25-44, Jan. 1992.

1 Introduction

Large scale shared-memory multiprocessors where hundreds or thousands of processors and memory modules are interconnected through an "equidistant" [35] multistage interconnection network have recently been designed and/or implemented. A typical "dance-hall" architecture, where a set of processors is lined up on one side of a processor-memory interconnection network and a set of memory modules lined up on the other side, is shown in Figure 1. The memory hierarchy consists of private caches C, local memories LM, and a shared global memory M. Examples of such architectures (although possibly without the complete memory hierarchy) include the University of Illinois Cedar machine [15], the BBN Butterfly multiprocessor [5], the NYU Ultracomputer [18], and the IBM RP3 machine [31].

One of the major problems associated with these architectures is the slow global memory access; thus the efficient management of local memory and private caches is very important. Local memory is generally used to store code and private data although shared data can be temporarily stored in local memory as in [6]. In some sense, we could say that local memory allows single copy caching. The decision on what will be stored in local memory and for how long, and on what will remain in global memory is entirely done at compile-time. On the other hand, caching in its usual sense is a run-time process. It is automatic in hardware-based caching mechanisms and can be *prevented* in some instances in software-assisted schemes. The use of local memory is not incompatible with that of private caches. On the contrary, they can complement each other. However, in this paper, we restrict ourselves to a study of caching of shared variables.

The presence of multiple private caches introduces the well-known cache coherence problem [7]. Hardware based protocols to solve the cache coherence problem are well understood in a shared-bus environment (e.g., [17, 22, 32, 37]). However these solutions cannot be extended to the dance-hall multiprocessors since they make use of the instantaneous broadcast and "snoopy" mechanisms provided by the shared-bus. Software-assisted [10, 25, 27, 33, 38, 40] and directory-based [1, 4, 7, 36, 41] schemes are usually advocated in such an environment.

In this paper, we propose a software-assisted cache coherence scheme which overcomes some of the inefficiencies of previous approaches by using a combination of a compile-time marking of references and a hardware-based local incoherence detection scheme. We also give a performance evaluation of our proposed scheme. In Section 2, we give the notation used throughout the paper. Section 3 reviews previous software-assisted approaches to enforcing cache coherence. In Section 4, a complete description of our approach is given. A correctness proof of our proposed scheme is given elsewhere [29] and is omitted here. Section 5 gives a quantitative comparison of our scheme with previous approaches. Section 6 provides some concluding remarks.

2 Definitions

Programs written for shared-memory multiprocessors may use explicit parallel constructs or may be conventional sequential programs transformed into equivalent parallel ones by a restructuring compiler or a preprocessor like Parafrase [24, 39], PFC [3] or PTRAN [2]. The parallelism is constrained by data dependences : *flow-dependence*, *anti-dependence*, and *output-dependence*[23]. Let r and r' be read operations and w and w' be write operations in a program. r is defined to be flow-dependent on w if the memory location written by w may be read by r. w is defined to be anti-dependent on r if the memory location read by r may be later written by w. w is defined to be output-dependent on w' if the memory location written by w' may be later overwritten by w.

Data dependence relationship among statements in a program can be graphically represented by a labeled directed multigraph called *data dependence graph*. The nodes of the graph are statements in the program and (S_i, S_j, δ) is in the arc set if and only if S_j is δ-dependent on S_i.

In parallel programs, as in sequential programs, most of the execution time is spent in loops. We distinguish between *Serial loops*, *DoAll loops*, and *DoAcross loops*. *Serial loops* are loops with inter-iteration delay equal to the execution time of the whole loop body (i.e., iteration

i cannot begin until iteration $i-1$ finishes). *DoAll* loops are loops with the delay equal to 0 (i.e., all iterations of a *DoAll* loop can be executed completely in parallel). *DoAcross* loops [11] have a delay between 0 and the execution time of the loop body caused by inter-iteration dependences. *DoAll* and *DoAcross* loops will also be called *parallel loops*.

We assume that a parallel program is composed of a set of *epochs* [25] which are either parallel loops or serial regions between them. Execution of an iteration of a parallel loop constitutes an instance of the epoch of type parallel loop. A serial region is a special type of epoch which has only one instance. Initially, we assume that only one level of nested Do loops is to be executed concurrently (i.e., only different iterations of a parallel loop can be executed in parallel on multiple processors). In Section 4.4, we will discuss the case where the parallel program may have nested parallelism.

As an example, consider the program segment shown below. It has two epochs, one consisting of the DoAcross loop i and the other consisting of the serial code after the DoAcross loop (i.e., $S \leftarrow 0$ and the serial loop k). The DoAcross loop has two dependences: a flow dependence δ_1 caused by the dependency between $A(i,j)$ (in S_1) and $A(i,j)$ (in S_2) and a flow dependence δ_2 caused by the dependency between $A(i,j)$ (in S_1) and $A(i-1,j)$ (in S_2).

```
        DoAcross i = 1, n1
          Do j = 1, n2
S1 :        A(i,j) ← ···
S2 :        ··· ← A(i,j) + A(i-1,j) ···
          END Do
        END DoAcross
        S ← 0
        Do k = 1, n3
          S ← S + A(k,1)
        END Do
```

3 Previous approaches to enforcing cache coherence

In [7], Censier and Feautrier defined *a memory scheme to be coherent if the value returned on a read is the value given by the latest store with the same address.* Most of the previous research done on enforcing cache coherence assumed shared resources such as a shared bus [17, 22, 32, 37] or a directory [4, 7, 36, 41]. These shared resources are, however, a hindrance towards scalability. A shared-bus is saturated as soon as the number of processors sharing it exceeds some threshold (certainly below 100 processors). Directories either grow linearly with the number of processor/cache pairs [7, 36, 41] or, if the encoding is more efficient, the protocols rely on broadcasts [4] that should be avoided in a scalable multiprocessor. We are therefore searching for alternatives that do not assume any shared resource but global memory and that do not use broadcast. Our approach, an instance of a "self-invalidation" cache coherence scheme, is based on software assists and local coherence checks.

The simplest software-assisted cache coherence scheme is to disallow caching of shared read/write data for the entire program [40]. This is accomplished by a compile-time marking of shared variables that are writable as non-cacheable. The mechanism is simple but inefficient in the sense that every reference to non-cacheable data is to be forwarded to global memory even though the addressed data could be cacheable during parts of the program where it is read-only or accessed exclusively by a single processor.

To overcome this performance penalty, Veidenbaum [38] proposed a scheme that allows changing of the cacheability of data. He identified the conditions necessary to cause cache incoherence. From those conditions, he showed that all global memory references can be routed through private caches inside DoAll loops and serial regions. But caching is prohibited inside loops with inter-iteration dependencies (DoAcross loops). In addition, caches that may potentially contain stale copies of data are flushed at the boundaries of loops. Although this scheme represents an obvious progress over no shared data caching at all, it still suffers from two inefficiencies. First, caching of read-only shared variables and variables that are exclusively accessed by only one processor inside DoAcross loops is disallowed. Second, the blind invalidations at the boundaries of loops flush out many cache entries that hold up-to-date

copies of data.

The first inefficiency of Veidenbaum's scheme was partially remedied in the schemes proposed by McAuliffe [27] and Lee [25]. Cacheability of variables is determined on an epoch by epoch basis. If a variable is potentially referenced by more than one processor in a given epoch and at least one of these references is a write, the variable is marked as non-cacheable for that epoch. Otherwise, it is marked as cacheable. Although this scheme captures localities within an epoch very well, it still suffers from performance penalties due to the cache flushes at the end of epochs.

Subsequently Cheong and Veidenbaum [9, 10] proposed the *fast-selective invalidation* scheme which is an improved version of Veidenbaum's scheme. Figure 2 shows the two possible conditions for a stale access [38] on which their reference marking scheme is based. In the first condition, a cache entry is loaded by a write by processor i and becomes stale when another processor j issues a write to the same memory location. The correctness criteria of cache coherence would be violated if processor i were still allowed to access what is now a stale copy in its cache. Similarly a stale access is possible when the cache entry is loaded by a read miss and becomes obsolete by a write from another processor. In their scheme, each read reference is marked at compile-time as either *memory-read* (the reference may potentially access a stale copy in the cache) or *cache-read* (on a hit, it is guaranteed that the cache entry is up-to-date). Notice that since these reference markings are done at compile-time, every reference to a shared variable that could be made after the variable was written by two different processors should be marked as *memory-read* by the compiler (condition 1). Similarly, every reference to a shared variable that could be made after the variable was read by one processor and then written by another processor should also be marked as *memory-read* (condition 2). From the above two conditions, it is easy to see that most read references to shared variables will be marked as *memory-read* by the compiler. In addition to the above compile-time detection scheme for stale accesses, a *change bit* is associated with each word in the cache. Its main purpose is to allow caching of *memory-read* references that occur more than once during an instance of an epoch. The change bits of the entire cache are set at the beginning of each epoch instance. A *cache-read* is executed as usual, irrespective of

the change bit. If a *memory-read* is issued and there is a copy of the corresponding memory block in the cache with the change bit set, the cache controller has no idea of whether the copy was loaded after the last write of the memory block or not. Therefore the controller has to take a conservative approach and the request is serviced by the global memory. However, it is possible that the copy was indeed loaded after the last write and is up-to-date. The extraneous *memory read* is due to the limitation of the compile-time analysis. Many more read requests than necessary will be directed to the global memory (recall that most reads from shared variables are marked as *memory-read*). Therefore the network traffic will be unduly increased and the scalability of the scheme is in question.

A scheme that is very similar to Cheong and Veidenbaum's was independently proposed by Cytron et al. [12]. This scheme uses the same compile-time analysis to detect stale accesses and, therefore, suffers from the same scalability problem. In Cytron's version, there is no change bit. Instead, intra-instance localities are captured by carefully moving around invalidation instructions so that cache entries loaded during the current instance are not needlessly invalidated.

The scheme that we propose in the next section has many similarities with the *version control approach* proposed at the same time, and independently, by Cheong and Veidenbaum [8]. We will briefly compare these two schemes at the end of Section 4.3.

For completeness purposes, we mention Smith's *one time identifier* scheme [33] which is more geared towards the caching of variables in critical sections. Table 1 shows a summary of the software-assisted cache coherence schemes discussed in this section.

4 Timestamp-based cache coherence scheme

4.1 Overview

We propose an extension of the previous methods that has for goal to capture more possibilities for the caching and retention in the caches of shared variables by looking more deeply at

inter-epoch localities. Our approach, like those in the previous section, is based on compile-time analysis and, in addition, on hardware support in the form of counters and tag bits in the cache.

The basic idea is as follows. We associate a "clock" (i.e., a counter) with each sharable data structure (array or scalar) of interest. This clock is incremented at the end of each epoch in which the data structure *may* be modified (a decision that is taken at compile-time). We also associate a timestamp with each word (for the time being we assume that the block size is equal to one word) in the cache. This timestamp is set to the value of the relevant clock + 1 when the word is updated in the cache. A reference to a cache word is valid if its timestamp is equal to or greater than its associated clock value.

As an example, let us consider the program segment and the associated data-dependence graph given in Figure 3. The output-dependence δ_1 is caused by the dependency between $X(f())$ and $X(g())$ and the flow-dependences δ_2 and δ_3 are caused by the dependencies between $X(g())$ and $X(p())$ and between $X(f())$ and $X(p())$ respectively. In Cheong and Veidenbaum's scheme, the reference to array X in S_3 will be marked as *memory-read* and, therefore, will be directed to the global memory. The situation would be about the same in Cytron's scheme. In this scheme, the above reference would be preceded by an invalidation instruction and be eventually serviced by the global memory. These are necessary because when there is a corresponding word in the cache it is not known, at compile-time, whether this cached word is stale (i.e., written in $Loop_{i_1}$) or valid (i.e., written in $Loop_{i_2}$).

In our approach, we would associate a clock with the array X. Its initial value is 0 and it is incremented by 1 at the end of each epoch (here parallel loop) in which the variable X is modified (here $Loop_{i_1}$ and $Loop_{i_2}$). After the first loop, the cache blocks corresponding to $X(f())$ would have a timestamp of 1 (0 (clock value) + 1) and after the second loop the cache blocks corresponding to $X(g())$ would have a timestamp of 2 (1 (clock value) + 1). When the statement S_3 is reached, and if there is a corresponding cache word for a reference to $X(p())$, then this cache reference will be valid if the timestamp is 2, corresponding to $X(g())$, and invalid otherwise.

As a second example, consider the program segment given in Figure 4. Again the reference to array X in S_4 will be directed to the global memory in Cheong and Veidenbaum's scheme since the compiler has to make the conservative assumption that the boolean expression $b()$ may evaluate to true. But if clocks and timestamps are maintained in the same way as in the previous example, the reference to the array X in $Loop_{i_4}$ can be satisfied by the cached words if they are loaded into the cache in either $Loop_{i_2}$ in the *then* case or $Loop_{i_3}$ in the *else* case.

The above two examples show that some of the inefficiencies of previous software-assisted cache coherence schemes can be remedied by history information which can be gathered at execution time.

Figure 5 shows our approach to capturing localities across different epochs. We divide the analysis into two parts: intra-epoch analysis and inter-epoch analysis. The intra-epoch analysis is done at compile-time and results in various markings of references. These markings indicate that (1) for a cache entry to be re-used in future epochs it should be guaranteed that there is no succeeding write reference to the same memory location in the same epoch, and (2) for a read reference to use a cache entry loaded in past epochs it should be guaranteed that the read reference does not have any preceding write to the same memory location in the same epoch. Inter-epoch analysis is performed at execution time using clocks and timestamps as indicated in the previous examples. This inter-epoch analysis detects any intervening write reference to the same memory location between the epoch in which the cache entry was loaded by a processor and the one in which it is accessed by the same processor. The above intra- and inter-epoch analyses, when combined, enable a processor on a read access to a shared variable to detect any write reference to the same variable by other processors since the last update in the associated cache entry in its local cache.

4.2 Support mechanism

Our cache coherence scheme requires the following hardware and software support mechanisms.

4.2.1 Hardware support

- Clock registers

  R_{clock} is the set of n_{clock}-bit clock registers associated with each processor. For each $r_{clock} \in R_{clock}$ and v_{clock} of type subrange $0..2^{n_{clock}} - 1$, the following operations are defined.

  - LOAD-CLOCK r_{clock}, v_{clock} with the semantics $r_{clock} \leftarrow v_{clock}$
  - CLEAR-ALL with the semantics: $\forall\, r_{clock} \in R_{clock}, r_{clock} \leftarrow 0$

- Cache memory. With each word in the cache, we associate:

  - an n_{clock}-bit timestamp.
  - a provisional bit (pb). This bit indicates when the entry has been loaded into the cache ($pb = 1$ loaded during the current instance, $pb = 0$ otherwise).
  - an invalid bit (ib) ($ib = 0$ valid, $ib = 1$ invalid).

Deciding a suitable value for n_{clock} is a tradeoff between the increase in the storage taken up by clocks and timestamps and the reduction in the number of clock or timestamp overflows (cf. end of Section 4.3).

Note that the hardware support is proportional to the cache size while in directory-based schemes, the extra hardware is proportional to the size of global memory. Let us assume a system with N processors and associated caches and N memory modules. There are M memory blocks/module and C blocks/cache. In both directory and timestamped based schemes, we need N more sophisticated controllers (N memory controllers in the directory case, N cache controllers in the timestamped one). The amount of tag bits, i.e., the storage overhead, is between $2NM$ (Archibald and Baer's scheme [4]) and $(N + 1)NM$ (Censier and Feautrier's scheme [7]) in directory-based schemes. In the timestamped case, there are $(n_{clock} + 2)$ extra tag bits per block for a total overhead of $NC(n_{clock} + 2)$. If one assumes M to be one order of magnitude greater than C, the timestamp-based scheme is $10(N + 1)/(n_{clock} + 2)$ times more economical than Censier and Feautrier's scheme in terms

of storage overhead. With $N = 200$ and $n_{clock} = 16$, the storage overhead of Censier and Feautrier's scheme is two orders of magnitude greater than that of the timestamp-based scheme. Notice that the relative space advantage of the timestamp-based scheme becomes greater as the number of processors in the system is increased. Also note that with 16 bit timestamp, the storage overhead of the timestamp-based scheme is comparable to that of Archibald and Baer's scheme, which is most space-economic among the directory-based schemes. (See Section 4.3 for the rationale behind the 16 bit timestamp.)

4.2.2 Software support

The software support consists of variables associated with shared data structures and a marking mechanism.

We associate a variable v_{clock} of type subrange $0..2^{n_{clock}} - 1$ with each shared variable v. Its value is typically loaded into one of the clock registers before referencing v (See the formats for reads and writes given in Section 4.3).

The reference marking scheme is based on a data-dependence analysis of a parallel program. Various attributes are given to read and write operations. These attributes are used to decide which actions to take for the associated operation at execution time.

Marking of write operations

A reference marking scheme for write operations is required in our scheme for both correctness and efficiency purposes. For correctness purposes, a reference marking is necessary for each write operation. This marking states whether the cache entry just written can be re-used in future epochs. The above decision is based on whether the write reference may have a succeeding write reference to the same memory location from other processors in the same epoch. If a given write operation may have such a succeeding write reference, the resultant cache entry cannot be re-used in future epochs and, therefore, the associated timestamp should not be set to the value of clock + 1.

For efficiency purposes, it is advantageous to know whether it is beneficial to load the cache

or to simply bypass it for a given write operation. For example, we would choose the latter alternative if the resultant cache entry cannot be re-used either in future epochs or in the current epoch instance.

From the above considerations, each write operation to a shared variable in a parallel program belongs to zero, one or both of the following two overlapping classes (cf. Figure 6; for a more formal definition of the markings, see [29]).

1. $\mathcal{TW}$ (*timestamped* writes)
2. $\mathcal{PW}$ (*provisional* writes)

A write operation in an epoch belongs to the first class (i.e., is marked as *timestamped-write*, $\mathcal{TW}$) if the memory location written by it cannot be overwritten by writes from different instances of the same epoch. As we will see later, the cache words updated by writes in $\mathcal{TW}$ are targeted for reads in later epochs.

A write operation in an epoch belongs to the second class (i.e., is marked as *provisional-write*, $\mathcal{PW}$) if there is at least one potential read in the same instance of the epoch that may read the cache word updated by the $\mathcal{PW}$ write. Write operations in $\mathcal{PW}$ are targeted for reads in the same instance of the epoch to which they belong.

Write operations which belong to neither of the above two classes are forced to bypass the cache since there are no potential reads from the cache words that might have been updated by them. This not only increases the efficiency of cache storage but also reduces the number of requests the cache controller should handle. The containment relationship among the markings of a write operation is depicted in Figure 6.

As an example of marking of write operations, let us consider the following DoAcross loop.

DoAcross $i_1 = 1, n_1$

$A(i_1) \leftarrow \cdots$

$\cdots$

$A(i_1 - 1) \leftarrow \cdots$

$\cdots$

$\cdots \leftarrow A(g(i_1)) + \cdots$

END DoAcross

For ease of explanation, the synchronizations required to satisfy the inter-iteration dependencies are omitted. The first write to the array (i.e., $A(i_1)$) cannot be marked as *timestamped-write* since the memory location written by it can be overwritten by the write to $A(i_1 - 1)$ in the subsequent iteration. It is marked as *provisional-write* if $A(g(k))$ may denote the same memory location as $A(k)$ for some k, $1 \leq k \leq n_1$ to allow the cache word written by $A(i_1)$ to service the read reference $A(g(i_1))$ generated in the same iteration. The second write to the array A (i.e., $A(i_1 - 1)$) is marked as *timestamped-write* since it is guaranteed that the memory locations written by it will not be overwritten by other writes. If both $A(k-1)$ and $A(g(k))$ may denote the same memory location for some k, $1 \leq k \leq n_1$, $A(i_1 - 1)$ is also marked as *provisional-write*.

Marking of read operations

For read operations, we need a slightly more complicated marking scheme than for write operations to handle both cache misses and cache hits. In the case of a miss, a scheme similar to that for write operations is needed to decide whether the resultant cache entry can be re-used in future epochs and whether it is beneficial to load the cache. In the case when there is a matching word in the cache, we need the same validity analysis as for a write. This leads to the following marking policies for cache loading and cache access.

- **Marking policy on read miss**

  On a read miss, we need a policy to decide whether the word fetched from the global

memory will be placed in the cache or not and, if so, whether the newly loaded cache entry can be re-used in future epochs. For these purposes, each read operation can be marked as *timestamped-loading* (i.e., $\mathcal{TL}$) and/or *provisional-loading* (i.e., $\mathcal{PL}$). The meanings of *timestamped-loading* and *provisional-loading* are analogous to those of *timestamped-write* and *provisional-write*, respectively. A read operation in an epoch is marked as *timestamped-loading* if the memory location read by it cannot be written by writes in other instances of the same epoch and, therefore, the resultant cache entry can be re-used in future epochs. As in *timestamped* writes, the cache words loaded by reads marked as $\mathcal{TL}$ are targeted for reads in later epochs. A read operation in an epoch is marked as *provisional-loading* if there is at least one other potential read in the same instance of the epoch that may access the cache word loaded by the former on a cache miss. In order to use the cache storage effectively, read operations marked as neither *timestamped-loading* nor *provisional-loading* do not load the cache with the word fetched from the global memory on cache misses.

- **Marking on read hit**

  As we mentioned before, for a read operation to utilize cache entries loaded in past epochs, it should be guaranteed that it does not have any preceding write to the same memory location in the same epoch. This test is necessary for correctness purposes. On the other hand, for efficiency reasons, it is advantageous to know whether there could be an up-to-date copy in the cache for a given read reference. For example, if it is decided that the cache cannot have an up-to-date copy for a given read reference, it would be beneficial to simply bypass the cache. This bypassing would reduce the number of requests the cache should service.

  From the above considerations, each read of a shared variable belongs to zero, one or both of the following two sets.

  1. $\mathcal{TR}$ (*timestamped* reads)
  2. $\mathcal{PR}$ (*provisional* reads)

  For a read to be in the set $\mathcal{TR}$, it should be guaranteed that it is not preceded by

any write to the same memory location from different instances of the same epoch. Read operations in $\mathcal{TR}$ are marked as *timestamped-read* and utilize the cache words updated by *timestamped* writes or loaded by reads marked as *timestamped-loading* on cache misses. The cache word is considered to be up-to-date if its timestamp value is equal to or greater than the current clock value of the corresponding variable.

A read operation r belongs to the second class (i.e., $\mathcal{PR}$) if there is at least one reference (write or read) to the same memory location in the same instance of the epoch that can reach r. This indicates whether it is possible that r may be satisfied by the cache word updated by a *provisional* write to the same memory location or loaded into the cache by another read from the same memory location marked as *provisional-loading* in the same instance of the epoch.

Read operations which belong to neither of the above two classes, called as *memory-only* reads, are made to bypass the cache since the request cannot be satisfied by the cache. It cannot be satisfied by cache words updated by *provisional* writes or loaded into the cache by reads marked as *provisional-loading* because there are no such writes or reads in the instance of the epoch to which the request belongs to (otherwise the request would have been marked as *provisional-read*). Neither can it be satisfied by *timestamped* writes or reads marked as *timestamped-loading* because the request can be preceded by a write reference to the same memory location from other epoch instances in the same epoch, that may make the previously loaded cache entry stale. The containment relationship among the markings of a read operation is depicted in Figure 7.

We give below an example of marking of read operations.

DoAcross $i_1 = 1, n_1$

$A(g(i_1)) \leftarrow \cdots$

$\cdots$

$\cdots \leftarrow A(f(i_1)) + \cdots$

$\cdots$

$\cdots \leftarrow A(p(i_1)) + \cdots$

END DoAcross

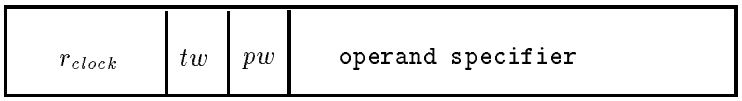
Again we omit the synchronizations to simplify the discussion. The first read operation (i.e., $A(f(i_1))$) is marked as *timestamped-read* if $f(k)$, $1 \leq k \leq n_1$ is not equal to any $g(k')$ for $1 \leq k' < k$. If the above condition is satisfied, the read reference $A(f(i_1))$ can access cache words brought in during past epochs. In addition, the read to $A(f(i_1))$ is marked as *provisional-read* if $f(k) = g(k)$ for some k, $1 \leq k \leq n_1$ to take advantage of the case when the cache word written by the reference to $A(g(i_1))$ is read by the reference to $A(f(i_1))$ executed in the same iteration of the parallel loop. The marking of the second read operation (i.e., $A(p(i_1))$) is done similarly.

For cache block loading purposes, the read to $A(f(i_1))$ is marked as *timestamped-loading* if $f(k)$, $1 \leq k \leq n_1$ is not equal to any $g(k')$ for $k < k' \leq n_1$. In this case, it is guaranteed that the cache words loaded into the cache on read misses on $A(f(i_1))$ remain up-to-date at the end of the parallel loop and may be referenced by *timestamped* reads in future epochs. In addition, it is marked as *provisional-loading* if $f(k)$ is equal to $p(k)$ for some k, $1 \leq k \leq n_1$ to take advantage of the case when the cache word loaded by a read miss on $A(f(i_1))$ satisfies the read reference $A(p(i_1))$ generated in the same iteration of the parallel loop. However, the read reference to $A(p(i_1))$ cannot be marked as *provisional-loading* since there is no possible read reference in the same iteration which can be satisfied by the cache word that might have been loaded into the cache on misses on $A(p(i_1))$.

4.3 Overall scheme

In the following, we assume that each instance of an epoch is executed on a distinct processor and that the block size of the cache is equal to one word. We will discuss the consequences of removing these assumptions later in this section.

In our scheme, each write reference to a shared variable has the following format :

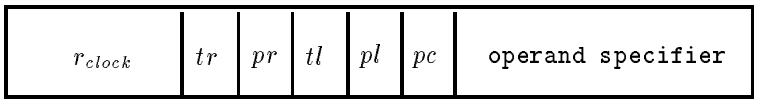


where r_{clock} is the clock register which holds the clock value of the variable being referenced, tw stands for *timestamped* write, and pw for *provisional* write.

Table 2 lists the actions taken for possible combinations of the tw and pw bits for a write to a shared variable. (Recall the tag bits for a cache entry described in Section 4.2.1.) The semantics implies a write-through policy. We could have considered a limited write-back policy where dirty cache words are written back to the global memory at the end of each epoch instance but we would have had to add an enforcement of write-backs for variables involved in data dependencies between epoch instances. In [28], a new global memory update policy, called *write-last* policy, was proposed which combines the advantages of the write-through and write-back policies by updating global memory as soon as possible but avoiding duplicate global memory updates.

Notice that for a non-timestamped write the timestamp field is set to r_{clock}. It is because the corresponding memory location may be overwritten by a different processor in the same epoch, thus making stale the newly loaded cache entry.

A read reference to a shared variable has the following format:



where r_{clock} has the same meaning as before, and tr stands for *timestamped* read, pr for *provisional* read, tl for *timestamped* loading, pl for *provisional* loading, and pc for *preceded*.

The *preceded* bit (pc) is set to 1 at compile-time if the read reference may be preceded by a write to the same memory location in the same epoch. It is set to 0 otherwise.

The condition for a read hit, assuming there is a matching cache block in the cache, is defined as : (the first term in each *and* condition depends on the instruction, i.e., is compiler generated, while the second depends on cache bits set at run-time)

$$(tr = 0 \ \wedge \ pr = 1) \ \wedge \ (ib = 0 \ \wedge \ pb = 1) \ (\textit{provisional} \text{ read})$$

or

$$(tr = 1 \ \wedge \ pr = 0) \ \wedge \ (ib = 0 \ \wedge \ ts \geq r_{clock}) \ (\textit{timestamped} \text{ read})$$

or

$$(tr = 1 \ \wedge \ pr = 1) \ \wedge \ (ib = 0 \ \wedge \ (pb = 1 \ \vee \ ts \geq r_{clock}))(both)$$

If there is a hit and the above Boolean expression is satisfied, the corresponding read reference is satisfied by the cache. Otherwise, the actions taken on a read reference are decided by the loading policy specified by the tl, pl, and pc bits. The pc bit is used to indicate whether on a read marked as *timestamped-loading*, we set the timestamp field of the referenced cache word at $clock + 1$ ($pc = 1$, i.e., the read may have a preceding write to the same memory location in the same epoch) or at $clock$ ($pc = 0$). Table 3 specifies the actions taken on a read miss based on the tl, pl, and pc bits.

An instruction which clears the *provisional* bits of the entire cache is inserted after each DoAll and DoAcross statement. The clock values of the shared variables which may be modified in a parallel loop (i.e., DoAll or DoAcross loop) are incremented by 1 in every processor at the end of the execution of the parallel loop.

At the beginning of a serial region, the processor assigned to execute that region clears the *provisional* bits of the associated private cache. It also increments by 1 the clock values of the shared variables modified by it during the serial region at the end of the serial region. The remaining processors increment the clock values of the shared variables which may be written in the serial region by 1. The instructions doing the above tasks (i.e., checking whether the current processor is the processor assigned to execute the current serial region and incrementing the clocks) are inserted at compile-time.

The invalid bits of the entire cache are set and the clock variables of all the shared variables used in a program are initialized to 0 at the start of the parallel program. In the rare occasion of a clock or a timestamp overflow, the entire contents of the private cache associated with the processor causing the overflow are invalidated and all the clock variables and clock registers in the processor are re-initialized to 0. If we assume n_{clock} to be 16, the above cache flushing and re-initialization would occur once every 2^{16} epoch executions for the worse case.

As mentioned earlier, Cheong and Veidenbaum's *version control approach* [8] and our approach have many similarities. The version control scheme uses a directed graph called *task execution graph* to model the execution of a parallel program. In the graph, each node denotes a task and each directed edge represents a dependency between the two involved tasks. The tasks at the same level in the task execution graph correspond to an epoch in our proposed scheme. Instead of clocks and timestamps, the version control approach uses *current version numbers* and *birth version numbers*. Clocks and version numbers, and timestamps and birth version numbers, are maintained in an analogous way. The main difference between the two schemes is that, by incorporating (and paying the "price" of) more sophisticated reference markings, our proposed scheme has a better chance of capturing localities between dependent tasks that can occur in DoAcross loops.

4.4 Extensions

Parallel programs with nested parallelism

Our scheme is flexible enough to be applied to programs with nested parallel loops. For this

purpose, we define an *era* as a segment of a parallel program delimited by two enclosing *barrier* operations. In this framework, a write is marked as *timestamped-write* if it is guaranteed that the memory location written by it cannot be overwritten in the era to which it belongs by writes from other processes including those that are spawned by itself. A write is marked as *provisional-write* if there is at least one potential read reference of the same memory location that may be issued by the same processor before the next synchronization point. As before, *timestamp-loading* and *provisional-loading* are defined similarly.

For a read to be marked as *timestamped-read*, it should be guaranteed that it is not preceded by any write to the same memory location from other processes in the same era. A read is marked as *provisional-read* if there is at least one preceding reference to the same memory location that may be executed by the same process after the most recent synchronization point.

As an example, consider the program segment, a single era, shown in Figure 8. The read operation from array B (i.e., B(i,n_1)) is marked as *timestamped-loading* since there is no succeeding write to the array. The read from array A (i.e., A(i,1)), however, cannot be marked as *timestamp-loading* since the memory location read by it will be written by one of its children processes spawned to execute the DoAll loop j. For cache access purposes, both reads are marked as *timestamped-read* since they cannot be preceded by any writes to the same locations from other processes in the era. Similarly, the write to array A inside the DoAll loop j (i.e., A(i,j)) is marked as *timestamped-write* since it is guaranteed to be the last write to that particular memory location in the era.

Multiple iterations of a parallel loop executed on the same processor.

The possibility that multiple iterations of a parallel loop are executed on the same processor could enhance rather than degrade the performance of our scheme. In this case, it is possible that a read marked as *memory-only-read* issued in an iteration of a parallel loop can be satisfied by the cache word updated by a write marked as *timestamped-write* or a read marked as *timestamped-loading* in an earlier iteration of the same parallel loop executed by the current processor. A *memory-only* read request can be satisfied by the cache if $ib = 0 \wedge ts > r_{clock}$.

This increases the bandwidth requirement of the caches since they must be interrogated even on a *memory-only* read. Caches could be designed so that the interrogation of their entries on a *memory-only* read can be enabled or disabled under program's control at run-time. The default option would be disabled. Enabling would occur if the ratio of the number of processors allocated to a parallel loop over the number of iterations of the loop is small enough.

Block size of the cache larger than one word.

There are both advantages and disadvantages in having a cache block size larger than one word. The main advantage of a larger-than-one-word block size is the implicit prefetching of the other words in the block on a cache miss [34]. The prime disadvantage of having a cache with a block size larger than one word in a shared-memory multiprocessor is *over-invalidations* due to false sharing [14]. In an invalidation-based cache coherence scheme, an invalidation of one word in a cache block invalidates other words in the block as well. This may cause a cascade of invalidations if multiple words in the same block are written by different processors at about the same time. In fact, as observed in [26], large block sizes can penalize performance in the type of architectures that we are studying.

If the block size were larger than one word, all the words in a cache block loaded on a read miss, except the requested one, will be assigned values of $pb = 1$ and $ts = r_{clock}$ if they come from the same variable (e.g., array) and the variable is not modified in the current epoch. Otherwise, the invalid bits of those words are set to 1. It would be an easy task for a compiler to allocate shared variables so that they do not cross the block boundaries. Note that for caches with block size larger than one word, each word in a cache block has its own pb and ib bits. On a write miss, we advocate a variation of a *store-allocate-non-fetch* policy in which a block is allocated, the write is reflected in the allocated block only on the referenced words, and the invalid bits of the other words in the block are set to 1.

5 Evaluation of software-assisted cache coherence schemes

The use of private caches is advocated in many proposed and/or implemented shared-memory multiprocessors to reduce both the average memory access time and the network traffic. As we briefly mentioned in Section 1, there are two general approaches to enforcing cache coherence in large-scale shared-memory multiprocessors: directory-based and software-assisted. In a directory-based cache coherence scheme, a directory entry, which is kept in the memory controller, is associated with each memory block. This entry encodes the state of the block. The state is used to decide whether there is a need for invalidations on a given write transaction to the block and if so, to locate the private caches which have a copy of the block to be invalidated. It is also used to tell whether the corresponding memory block is stale or not and if so, to locate the private cache which is guaranteed to have the most current copy of the block. In addition to the state in the global directory, a local state is usually associated with each cache block in private caches. This local state is used to allow a private cache to service most requests from its associated processor without incurring any global actions.

Even though cache coherence schemes based on directories can be quite efficient in yielding a high hit ratio because of their ability to dynamically keep track of the status of each block, the network traffic generated for invalidation requests and for the manipulation of local and/or global state information can be substantial. In our proposed scheme, cache entries are invalidated solely by local processors and no globally-manipulated information is associated either with cache blocks or with memory blocks. This eliminates the extra network traffic at the expense of less efficient caching. An initial performance comparison between our proposed timestamp-based scheme and a directory-based scheme was made in [30]. The results indicated that the timestamp-based scheme generally yields miss ratios comparable to those of the directory-based scheme with less network traffic for parallel programs written in the SPMD (single program multiple data) model of parallel programming where parallelism is expressed in terms of DoAll loops. Detailed results from the comparison are not repeated here and interested readers are referred to [30].

In our comparative study [30], we have used the most efficient (in terms of hit ratio and

network traffic) directory scheme. In order to have a fair comparison, we needed to determine the most efficient software-assisted scheme. Therefore, in this section, we evaluate the relative effectiveness of various software-assisted cache coherence schemes. Lee's [25], Cheong and Veidenbaum's [9, 10], and our own timestamp-based cache coherence schemes are chosen since they can exploit an increasing amount of program localities. Notice that Cheong and Veidenbaum's scheme being evaluated in this paper is the fast-selective scheme [9, 10], not the version control scheme [8]. We expect that the performance of the version control scheme would be essentially the same as that of the timestamp-based scheme for parallel programs with only DoAll loops. Our evaluation methodology is trace-driven simulation.

The method used to get parallel traces and the simulator structure are described in Section 5.1. Section 5.2 discusses the sample parallel programs traced. Section 5.3 presents our simulation results.

5.1 Methodology

As shown in Figure 9, the simulation consists of three steps: generation of serial traces with markers, preprocessing, and actual trace-driven simulation.

5.1.1 Generation of serial traces

Trace data used in the experiment is obtained by preprocessing traces from serial execution of parallel programs with marking instructions embedded in them. Markers are placed on events of interest such as start of an epoch, end of an instance of an epoch, etc. Since the programs used to generate the traces were written using explicit parallel constructs (e.g., DoAll, END DoAll), the insertion of marking instructions was straightforward. The original serial traces are gathered by *tracer*, a trace generating program run on the VAX architecture under the Ultrix V2.3 operating system [19]. For each memory reference the corresponding trace record contains fields for the type of reference (e.g., read instruction, read data, write data, etc), storage segment involved (i.e., data segment, stack segment or instruction segment), the size

of the item being referenced, and the memory address.

5.1.2 Preprocessing

The original serial traces are restructured through the preprocessing step based on various types of markers as shown in Figure 10. Attributes required during the actual simulation are assigned to each reference during this step. These include markings of references used in the cache coherence schemes simulated, the value of the associated clock register, etc. (for a complete list, see Table 4). The markings represent an upper bound on the accuracy of data dependence information that could be obtained from a parallelizing compiler. For example, in the program segment shown in Figure 11 the memory location corresponding to A(2) is marked as cacheable in our experiment for Lee's scheme if the *then* path is taken at execution time during the third iteration of the DoAcross loop. However, the compiler would mark it as non-cacheable since the compiler should make the conservative assumption that the *else* path could be taken. This slightly increases the hit ratio of Lee's scheme. These kinds of optimistic markings are used throughout but do not bias the experiment since about the same degree of favor is given to the three schemes evaluated.

In order to remove the degree of freedom brought upon by the sophistication of the register allocation techniques used in the compiler, two different parallel traces from the same serial trace were simulated. The first trace (*unfiltered* version) contains all the references that the original serial trace has. In the second trace (*filtered* version), all the *registerable* references are filtered out. A read reference is defined to be *registerable* if it has at least one preceding reference (either read or write) to the same memory location in the same epoch instance whereas a write is defined to be so if it has at least one succeeding write reference to the same memory location in the same instance. These registerable references represent an upper bound of references that can be captured by registers. Even though the unfiltered trace is more realistic, especially when we do not know how many registers will be used in a compiler, the filtered one allows us to see the importance of inter-epoch locality since the intra-instance locality is captured by the registerable references. We think that the

evaluation based on filtered traces becomes more important with the existence of increasingly sophisticated register allocation techniques.

5.1.3 Trace-driven simulation

Traces restructured by the preprocessing step are used to drive a cache simulator in order to obtain various performance measures such as the hit ratio and the amount of network traffic for various cache organizations. The input parameters to the simulator are the cache size, set associativity, type of cache coherence scheme, global memory update policy, replacement policy, number of processors allocated, and processor scheduling policy. A direct-mapped cache with one-word block size, so that false sharing effects are eliminated, is assumed throughout the experiment. Table 5 shows the ranges of the parameters which define the space actually explored by our experiment. Only data references (both private and shared data references) are simulated since requests for instructions can be handled similarly in the three schemes. The output from the simulator includes the miss ratio for each type of reference and the amount of network traffic. Reasons for cache misses such as block miss, timestamp mismatch, etc. are also provided. We report the above performance figures for only one processor. This is sufficient because of the self-invalidating nature of the schemes under evaluation.

The actual scheduling of parallel iterations on processors, based on the number of allocated processors, is done when the restructured trace is processed by the simulator. Two different types of scheduling policies are used: pre-scheduling and random scheduling. In the pre-scheduling policy the i^{th} iteration of a parallel loop is executed on $processor_{i \ mod \ \mathcal{P}}$ where $\mathcal{P}$ is the number of processors allocated. In this policy, it is also assumed that one designated processor called MASTER always executes all the serial regions in the program. In the random scheduling case, whenever there is an instance of an epoch for execution (including a serial region), a processor is randomly selected to execute it.

5.2 Parallel programs traced

Three parallel application programs are used to generate the traces. They follow the same parallel programming paradigm, namely: SPMD (single program multiple data).

The first program, *sim* [20], simulates a multistage interconnection network which serves vector load/store requests from processors to memory modules. The second program, *gauss* [13], uses the Gauss elimination technique to solve the following linear system of equations.

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$
$$\cdots\cdot$$
$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$$

The last application, *mhd*, is a program for the numerical solution of two-dimensional magnetohydrodynamic (MHD) differential equations [16]. The first two programs are written in *pcp*, a parallel extension of the C programming language [21]. The last one is a slightly modified version of the program originally written in HEP FORTRAN. These programs do not contain any DoAcross loop; this slightly favors the cache coherence schemes with little or no ability to capture inter-iteration locality. Table 6 shows a summary of the characteristics of the three traces used in the experiment. In the table, a value (or contents of a memory location) is considered shared if it is accessed by more than one process.

5.3 Results

There are three types of temporal localities in parallel programs: intra-instance, inter-instance, and inter-epoch localities. The main difference among the three schemes evaluated here is in their ability to exploit the above types of localities. All three schemes can exploit the first two types, although the amount of inter-instance locality captured by Lee's and Cheong and Veidenbaum's schemes is limited. It is because caching is restricted to references to read-only shared variables in the former case and to read references marked as *cache-read*

in the latter. Only Cheong and Veidenbaum's and the timestamp-based schemes can explore inter-epoch locality since in Lee's scheme caches are flushed at the end of each epoch. As we mentioned in Section 4.1, the amount of inter-epoch locality captured by Cheong and Veidenbaum's scheme is very limited.

This qualitative assessment is validated by the data shown in Figure 12 where the miss ratios of the three schemes are shown along with the miss ratio of the serial execution case when the parallel programs happen to be executed on a single processor. The serial execution always yields the highest hit ratio since no cache entry is needlessly invalidated.

For small caches there is not much difference among the three schemes. It is because the three schemes respond well to intra-epoch locality (i.e., intra-instance + inter-instance) which can be captured even with small caches. But the effect of inter-epoch locality becomes the dominant factor once all of the intra-epoch locality is captured by the cache. This is the main reason why Cheong and Veidenbaum's and the timestamp-based schemes yield better hit ratios than Lee's scheme for larger caches with the exception of the *gauss* program (cf. Figure 12). In the *gauss* application, however, Lee's scheme yields better hit ratios than Cheong and Veidenbaum's scheme. This is because Lee's scheme can capture more effectively the localities caused by read-only sharing of shared memory locations, in this case read-only sharing of pivot rows, across different iterations of a parallel epoch. Another interesting point to notice is the rapid drop in miss ratios in the *gauss* program when the cache size is increased from 256 words to 512 words in Lee's and the timestamp-based schemes. A careful inspection of the corresponding trace shows that 256 words is the threshold cache size beyond which the localities due to the above read-only sharing are captured by the caches.

The above trends are more apparent if we consider the miss ratios for the same configuration for filtered traces as depicted in Figure 13. (Recall that in filtered traces the intra-instance localities are captured by registers.) The remaining sole source of locality for Lee's scheme is inter-instance localities due to read-only sharing of shared variables across different instances of the same epoch. This is the reason why Lee's scheme yields miss ratios near one for the *mhd* program in which there is little such read-only sharing. Cheong and Veidenbaum's

scheme still captures some inter-instance and inter-epoch localities but to a limited degree as we can see in the figure. The timestamp-based scheme, however, consistently manages to maintain miss ratios comparable to those of the serial execution case.

The same analysis holds in the multiprocessor simulations for the *sim* and *mhd* programs under a pre-scheduling policy (see Figure 14 for *sim*; similar data for *mhd* can be found in [28]). This is a consequence of the pre-scheduling which allows shared memory locations written by one processor to be subsequently read by the same processor. In the *sim* program, the rows of the multistage interconnection network are simulated in parallel for each network cycle and, with pre-scheduling, the same row is always simulated by the same processor. This increases the chance of the reuse of the cache contents before they become stale. A similar behavior occurs in *mhd*. These characteristics provide the timestamp-based scheme ample opportunities to exploit inter-epoch localities.

In general, however, the chances for shared-memory locations written by a processor to be referenced by the same processor before they become stale are decreased as more processors are allocated in the random scheduling case. This is shown in Figure 15 for the *sim* application. Miss ratios in the random scheduling case are worse than their pre-scheduling counterparts for the *sim* and *mhd* applications. Furthermore, the difference in miss ratios between the timestamp-based scheme and the better of the other two schemes is reduced as more processors are allocated in the random scheduling case as we can see in the figure. One interesting point to notice is that for the timestamp-based scheme the miss ratios of caches are lowered as more processors are allocated for the *sim* and *mhd* applications (cf. Figure 12 and Figure 14 for *sim*). It is because, as more processors are allocated, the amount of shared data to be cached per processor is reduced, thus reducing the number of misses due to replacements.

On the other hand, as the number of processors is increased, the miss ratios for the *gauss* application are not lower as we can see in Figure 16. It is because the amount of inter-instance locality, which is one of the main components of cache hits in the *gauss* application, decreases. Also, the hit ratios under pre-scheduling are worse than those obtained under ran-

dom scheduling for some cache sizes (cf. Figure 16 and Figure 17). This seemingly anomalous behavior can be explained by looking carefully inside the Gauss Elimination algorithm. The algorithm consists of two major steps: reduction and back substitution. The code segment corresponding to the reduction step is given in Figure 18. It starts with a parallel reduction with a_{11} as a pivot, i.e., the elements of the first column are zeroed out and the other elements are modified adequately on a row by row basis. The process is then repeated for the (new) second column and so on until the matrix A is reduced to an upper triangular matrix.

Accesses to each row of the matrix A by processors for each invocation of the DoAll loop are shown in Figure 19, assuming that three processors are allocated and that they are pre-scheduled naively, i.e., processor i executes the i^{th} instance of the DoAll loop on every invocation of the DoAll loop. In the figure, we can notice that each processor reads the row of the matrix A which was last written by some other processor in the previous invocation of the DoAll loop. This drastically reduces the chance of capturing inter-epoch locality since most of the cache contents associated with the matrix A become stale whenever a new step of the reduction is started. Therefore none of the three cache coherence schemes work well in this situation; this fact is well illustrated in Figure 20 in which the reasons for misses are shown for the unfiltered gauss trace when eight processors are pre-scheduled (See also Figure 16 which shows an overall miss ratio of 0.2 to 0.4 depending on the cache size for 8 processor case). As can be seen, most misses for large caches are due to timestamp mismatch rather than block misses. On the other hand, if we use random scheduling, there is some chance, although small, that elements of the matrix A written by a processor are read by the same processor during the next invocation of the DoAll loop. This is the main reason why, in the *gauss* program, the random scheduling yields slightly better hit ratios than the pre-scheduling for some cache sizes.

Naturally, better performance could be achieved by tailoring the scheduling to the application by using some form of *data-constrained* scheduling. Figure 21 shows such a schedule for *gauss* assuming three processors are allocated. In this schedule, the processor that has executed the k^{th} instance of the DoAll loop during the previous invocation is assigned to execute the $k-1^{th}$ instance to maximize the reuse of the cache entries associated with the array A. The results

from the new simulation for 8 processors are compared to those from the pre-scheduling case in Figure 22.

The results from the data-constrained scheduling case are quite consistent with our expectations. With the new schedule, the timestamp-based scheme yields better hit ratios than in the pre-scheduling and random scheduling cases especially for large caches. Most of the improvements result from the fact that most of the cache entries associated with the matrix A are now re-used over different invocations of the DoAll loop in the new schedule. On the other hand, the miss ratios from the other two schemes are barely improved because of their limitation to exploit inter-epoch locality. This enlarges the performance gap between the timestamp-based scheme and the other two schemes. One interesting point to note from the figure is that the miss ratios for the pre-scheduling case are better than those for the data-constrained scheduling for small caches ($<$ 4K words). This is because, in the new schedule, p_1 (from which we measured the miss ratios) executes fewer iterations of the DoAll loop during the reduction step than in the pre-scheduling case. This, in turn, reduces the amount of inter-instance locality due to the read-only sharing of the pivot rows for p_1, thus yielding lower hit ratios than in the pre-scheduling case. Such an effect would have been minimized if we used a much larger A matrix than that used in the experiment or if the miss ratios were averaged over all processors.

The data gathered by the traces allows us to also provide some information on the potential network traffic. Figure 23 and Figure 24 depict the ratios of write-back traffic to write-through traffic for different cache sizes for $p = 1$ and $p = 64$ respectively. The three schemes evaluated here are assumed to have the same write-back traffic because global memory update policy used in one of the schemes can be equally used in the others. The result shows that, except for *sim* the ratio is almost insensitive to cache size. This indicates that few memory locations are written in a single epoch instance in our experiment. The exception for the *sim* program (from 0.33 to 0.26 for $p = 1$ and from 0.34 to 0.28 for $p = 64$) occurs when the cache size is increased from 1 Kwords to 2 Kwords because of access to some large data structures larger than 1024 words. This can be also seen in Figure 12, Figure 14, and Figure 15 in which hit ratios for the *sim* program are significantly improved when the cache size is increased from

1 Kwords to 2 Kwords independently of the number of allocated processors and the scheduling policy. The high write-back ratios of the *gauss* application arise because most of the writes are to shared memory locations and these locations are rarely written more than once inside a single instance.

The above results indicate that the write-back policy may reduce the write traffic substantially at the possible expense of transient fluctuations in the network traffic at the end of epoch instances. This, in turn, provides a ground for our write-last global memory update policy that tries to minimize these fluctuations without increasing the network traffic.

6 Conclusions

The efficient enforcement of coherence of multiple private caches is essential to the effective performance of dance-hall type architectures. In this paper, we propose a self-invalidating cache coherence scheme which overcomes some of the inefficiencies of previous software-assisted schemes. Our approach is based on compile-time marking of read/write operations and on execution time incoherence detection which makes use of locally maintained clocks and timestamps. We show, through trace-driven simulation, that our timestamp-based cache coherence has a better performance (hit ratio) than previous schemes especially when the processors are carefully scheduled so as to maximize the re-use of cache contents. We also investigate the effects of register allocation on cache performance through trace filtering and show that the ability to capture localities across different epochs, the main strong point of our approach, becomes relatively more important as more sophisticated register allocation techniques are applied.

There are numerous open areas for future research. One such area is an integration of the timestamp-based cache coherence scheme into a parallelizing compiler such as Parafrase [24], PFC [3], and PTRAN [2]. We think that such an integration also provides a ground for a fair comparison of software-assisted schemes with directory based cache coherence schemes and user-controlled local memory. It remains to assess (via simulation of parallel programs)

which scheme or which combination of the three schemes is most advantageous in terms of processor utilization and overall throughput.

Another interesting topic for future research is an investigation of multilevel cache hierarchies in which the first level cache employs a software-assisted cache coherence scheme whereas the second level uses a directory-based scheme. This configuration seems to be a nice match since the first level cache will not be disturbed unduly by unrelated coherence events and the second level cache will have a high hit ratio because it is directory-based.

ACKNOWLEDGMENT

The authors thank C. Ebeling, S. Eggers, and R. Cypher for their many helpful discussions. We would also like to thank R. Henry for allowing us to use *tracer* program. Thanks also to E. D. Brooks III for providing us with the parallel programs used to generate the parallel traces. Finally, we want to thank the anonymous referees for their constructive comments.

References

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

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the 1987 International Conference on Supercomputing*, June 1987.

[3] J. R. Allen and K. Kennedy. PFC: A program to convert FORTRAN to parallel form. Technical Report MASC Technical Report 82-6, Dept. of Math. Sciences, Rice University, March 1982.

[4] J. Archibald and J.-L. Baer. An economical solution to the cache coherence problem. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 355–362, June 1985.

[5] BBN. *Butterfly Parallel Processor Overview*, version 1 edition.

[6] D. Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Rice University, April 1987.

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

[8] H. Cheong and A. Veidenbaum. A version control approach to cache coherence. In *Proceedings of the 1989 International Conference on Supercomputing*, pages 322–330, June 1989.

[9] H. Cheong and A. V. Veidenbaum. Stale data detection and coherence enforcement using flow analysis. In *Proceedings of the 1988 International Conference on Parallel Processing, Vol. I Architecture*, pages 138–145, August 1988.

[10] H. Cheong and A. V. Veidenbaum. A cache coherence scheme with fast selective invalidation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 299–307, June 1988.

[11] R. Cytron. Doacross: Beyond Vectorization for Multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844. IEEE, August 1986.

[12] R. Cytron, S. Karlovsky, and K. P. McAuliffe. Automatic management of programmable caches (extended abstract). In *Proceedings of the 1988 International Conference on Parallel Processing, Vol. II Software*, pages 229–238, August 1988.

[13] G. A. Darmohray and E. D. Brooks III. Gaussian techniques on shared memory multiprocessor computers. Unpublished Technical Report.

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

[15] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh. Cedar - a large scale multiprocessor. *Computer Architecture News*, 11(1):7–11, March 1983.

[16] W. Gentzsch. *Vectorization of computer programs with applications to Computational Fluid Dynamics*, volume 8 of *Notes on Numerical Fluid Mechanics.* Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig, 1984.

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

[18] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing a MIMD, Shared-Memory Parallel Machine. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 27–42, April 1982.

[19] R. R. Henry. Address and instruction tracing for the VAX architecture. Unpublished Technical Report, August 1983.

[20] E. D. Brooks III. Performance of the butterfly processor-memory interconnection in a vector environment. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 21–24. IEEE, August 1985.

[21] E. D. Brooks III and G. A. Darmohray. A parallel extension of C that is 99 % fat free. Unpublished Technical Report.

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

[23] D. J. Kuck. *The Structure of Computers and Computation.* Wiley, New York, NY, 1978.

[24] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe. The structure of an advanced vectorizer for pipelined processors. In *Computer Software and Applications Conference (COMPSAC80)*, pages 709–715. IEEE, October 1980.

[25] R. L. Lee. The effectiveness of caches and data prefetch buffers in large-scale shared memory multiprocessors. Technical Report CSRD Report. No. 670, Center for Supercomputing Research and Development, University of Illinois, May 1987.

[26] R. L. Lee, P.-C. Yew, and D. H. Lawrie. Multiprocessor cache design considerations. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 253–262, June 1987.

[27] K. P. McAuliffe. *Analysis of Cache Memories in Highly Parallel Systems.* PhD thesis, New York University, May 1986.

[28] S. L. Min. *Memory Hierarchy Management Schemes in Large Scale Shared-Memory Multiprocessors.* PhD thesis, University of Washington, 1989.

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

[30] S. L. Min and J.-L. Baer. A performance comparison of directory-based and timestamp-based cache coherence schemes. In *Proceedings of the 1990 International Conference on Parallel Processing, Vol. I Architecture*, pages 305–311, August 1990.

[31] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771. IEEE, August 1985.

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

[33] A. J. Smith. CPU cache consistency with software support and using "one time identifiers". In *Proceedings of the Pacific Computer Communications Symposium*, pages 22–24, October 1985.

[34] A. J. Smith. Line (block) size choice for CPU caches. *IEEE Transactions on Computers*, C-36(9):1063–1075, September 1987.

[35] L. Snyder. Type architectures, shared memory, and the corollary of modest potential. *Annual Review of Computer Science*, 1:289–317, 1986.

[36] C. K. Tang. Cache design in the tightly coupled multiprocessor system. In *AFIPS Conference Proceedings National Computer Conference*, pages 749–753, 1976.

[37] C. P. Thacker and L. C. Stewart. Firefly : a multiprocessor workstation. In *Proceedings Second International Conference on Architectural support for Programming Languages and Operating Systems*, pages 164–172, October, 1987.

[38] A. V. Veidenbaum. A compiler-assisted cache coherence solution for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Proessing*, pages 1029–1036, August 1986.

[39] M. Wolfe. Optimizing compilers for supercomputers. Technical Report UIUCDCS-R-82-1105, Department of Computer Science, University of Illinois at Urbana-Champaign, October, 1982.

[40] W. A. Wulf and C. G. Bell. C.mmp - A multi-mini processor. In *Proc. Fall Joint Computer Conference*, pages 765–777, Montvale, New Jersey, December 1972.

[41] W. C. Yen, D. W. L. Yen, and K.-S. Fu. Data coherence problem in a multicache system. *IEEE Transactions on Computers*, C-34(1):56–65, January 1985.

Figure 1: Dance-hall type architecture

Figure 2: Conditions for a stale cache access

Figure 3: Sample program segment which exhibits the inefficiencies of the previous approaches

Figure 4: Another sample program segment showing the inefficiencies of the previous approaches

Figure 5: Overall approach to capture inter-epoch localities

Figure 6: Containment relationship among various write markings

Figure 7: Containment relationship among various read markings

Figure 8: Example of parallel program with nested parallelism

Figure 9: Overall structure of simulation process

Figure 10: Restructuring of a serial trace into a parallel trace

Figure 11: Example of an optimistic marking

Figure 12: Miss ratio for single processor case for the original (unfiltered) traces

Figure 13: Miss ratio for single processor case for the filtered traces

Figure 14: Miss ratio for the *sim* application for $p = 8$ and $p = 64$ (pre-scheduling case)

Figure 15: Miss ratio for the *sim* application for $p = 8$ and $p = 64$ (random-scheduling case)

Figure 16: Miss ratio for the *gauss* application for $p = 8$ and $p = 64$ (pre-scheduling case)

Figure 17: Miss ratio for the *gauss* application for $p = 8$ and $p = 64$ (random-scheduling case)

Figure 18: The reduction step of the Gauss Elimination algorithm

Figure 19: Accesses to matrix A by processors when $p = 3$

Figure 20: Decomposition of misses according to sources for $p = 8$ for the unfiltered gauss trace (pre-sched)

Figure 21: Accesses to matrix A by processors in the data-constrained schedule

Figure 22: Miss ratio for $p = 8$ for the unfiltered gauss trace (data-constrained sched)

Figure 23: Ratios of write-back traffic to write-through traffic when $p = 1$

Figure 24: Ratios of write-back traffic to write-through traffic when $p = 64$

	Wulf Bell	Veidenbaum	McAuliffe	Lee	Veidenbaum Cheong	Cytron et al.	Smith
coherence enforcement unit	each variable	all variables	each variable	each variable	each reference	each reference	each page
coherence enforcement region	program	loop	computational unit (CU)	epoch	program	program	critical section
detection of incoherence	usage of variables in a program	loop nesting structure	usage of variables in a CU	usage of variables in an epoch	flow analysis	data-dependence analysis	OTI
invalidation place	— —	boundary of loops	boundary of CU's	boundary of epochs	boundary of loops	each reference	end of critical section
global memory update policy	—	write-through	write-back	write-back	write-through	hybrid scheme	write-through (for shared writable data)

Table 1: Summary of software-assisted cache coherence schemes.

Type of write		Actions taken
Memory-only write	$tw = 0$ & $pw = 0$	Update the global memory.
Provisional write	$tw = 0$ & $pw = 1$	if *block miss* then allocate a new cache entry. Update both the global memory and the associated cache entry. $ts \leftarrow r_{clock}$ $pb \leftarrow 1$ if $ib = 1$ then $ib \leftarrow 0$
Timestamped write	$tw = 1$ & $pw = 0$	if *block miss* then allocate a new cache entry. Update both the global memory and the associated cache entry. $ts \leftarrow r_{clock} + 1$ $pb \leftarrow 0$ if $ib = 1$ then $ib \leftarrow 0$
Timestamped & *provisional* write	$tw = 1$ & $pw = 1$	if *block miss* then allocate a new cache entry. Update both the global memory and the associated cache entry. $ts \leftarrow r_{clock} + 1$ $pb \leftarrow 1$ if $ib = 1$ then $ib \leftarrow 0$

Table 2: Actions taken for a write to a shared variable

Type of read		Actions taken
No loading	$tl = 0$ & $pl = 0$	Fetch from the global memory and do not cache.
Provisional loading	$tl = 0$ & $pl = 1$	Fetch from the global memory and load into the cache. $ts \leftarrow r_{clock}$ $pb \leftarrow 1$ if $ib = 1$ then $ib \leftarrow 0$
Timestamped loading	$tl = 1$ & $pl = 0$	Fetch from the global memory and load into the cache. if $pc = 1$ then $ts \leftarrow r_{clock} + 1$ else $ts \leftarrow r_{clock}$ $pb \leftarrow 0$ if $ib = 1$ then $ib \leftarrow 0$
Timestamped & *provisional* loading	$tl = 1$ & $pl = 1$	Fetch from the global memory and load into the cache. if $pc = 1$ then $ts \leftarrow r_{clock} + 1$ else $ts \leftarrow r_{clock}$ $pb \leftarrow 1$ if $ib = 1$ then $ib \leftarrow 0$

Table 3: Actions taken on a read miss

Attribute	Scheme	Type of reference	Condition
non-cacheable	Lee's scheme	read & write	if and only if the associated variable is referenced by more than one process in the epoch to which it belongs and at least one of these references is a write
memory-read	Cheong & Veidenbaum's scheme	read	after the corresponding location is written more than twice in different epochs
cache-read	Cheong & Veidenbaum's scheme	read	if and only if it is not *memory-read*
timestamped-write	timestamp-based scheme	write	if and only if it has no succeeding write to the same location in the same epoch
provisional-write	timestamp-based scheme	write	if and only if there is at least one succeeding read reference to the same location in the same instance
timestamped-read	timestamp-based scheme	read	if and only if it has no preceding write to the same location in the same epoch
provisional-read	timestamp-based scheme	read	if and only if there is at least one preceding reference to the same location in the same instance
timestamped-loading	timestamp-based scheme	read	if and only if it has no succeeding write to the same location in the same epoch
provisional-loading	timestamp-based scheme	read	if and only if there is at least one succeeding read reference to the same location in the same instance
preceded	timestamp-based scheme	read	if and only if there is at least one preceding write reference to the same location in the same epoch
clock value	timestamp-based scheme	read & write	the value of the associated clock register

Table 4: Various attributes of references gathered during the preprocessing step

Parameter	Range
Scheme	Lee, Cheong and Veidenbaum, timestamp-based
Scheduling policy	pre-sched, random-sched
Trace type	unfiltered, filtered
Number of processors	1 2 4 8 16 32 64
Cache sizes	128 256 512 1K 2K 4K 8K 16K 32K ∞ (infinite cache) (in words)

Table 5: Range of parameters used in the experiment

	sim		*gauss*		*mhd*	
	unfiltered	filtered	unfiltered	filtered	unfiltered	filtered
number of instructions traced	7973367	7973367	4470574	4470574	5626036	5626036
number of data references	5855877	1010014	4499205	2096148	1533562	640969
number of data reads	4689713	707285	3792278	1409077	1087221	385378
number of data writes	1166164	302729	706927	687071	446341	255591
number of *registerable* reads	3982428	0	2383201	0	701843	0
number of *registerable* writes	863435	0	19856	0	190750	0
number of epochs	641	641	201	201	73	73
number of serial epochs	161	161	102	102	13	13
number of parallel epochs	480	480	99	99	60	60
number of parallel instances	26080	26080	5049	5049	2148	2148
number of shared values	158351	158351	667009	667009	199898	199898
number of reads to shared values	518462	251569	2354402	1339154	704584	365195

Table 6: Characteristics of the traces used in the experiment