A Timestamp-based Selective Invalidation Scheme for Multiprocessor Cache Coherence*

Xin Yuan Rami Melhem Rajiv Gupta
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract – Among all software cache coherence strategies, the ones that are based on the concept of timestamps show the greatest potential in terms of cache performance. The early timestamp methods suffer from high hardware overhead. Improvements have been proposed to reduce hardware overhead at the expense of either increasing runtime overhead or sacrificing cache performance. In this paper, we discuss the limitations of the previous timestamp-based methods and propose a new software cache coherence scheme. Our scheme exploits the inter-level locality with significantly less hardware support than the early timestamp methods while introducing only constant runtime overhead for each epoch during the execution of a program. Simulation results show that the proposed scheme achieves higher performance than the previous schemes with comparable hardware overhead.

1 Introduction

Private caches are critical components of high performance multiprocessor systems. The use of private caches reduces network traffic and memory access latency. However, it also introduces the cache coherence problem. A mechanism must be implemented to keep the caches coherent in order for a program to run correctly on a multiprocessor system.

Among the existing software controlled cache coherence schemes, the methods based on the concept of timestamps are more effective in preserving cache lines across task boundaries than other software methods [4, 9]. The early timestamp-based methods, such as the version control method [3] and the timestamp method[10], use an explicit timestamp table to store the current version number for each variable. They also use an additional field in each cache line to store the version number of the cache line. Although the cache performance of these methods approaches that of the hardware schemes, the maintenance of the table and the additional timestamp field introduces hardware and runtime overheads. The later methods, such as TS1 with a 1 bit timestamp [8], the generational algorithm [5] and the two-phase invalidation scheme (TPI) [6], try to reduce these overheads. TS1 avoids most of the hardware overhead by eliminating the timestamp table and reducing the additional timestamp field to one bit for each cache line. However, explicit invalidation instructions are needed at the end of each level. Hardware support for an efficient cache invalidation mechanism introduces additional hardware overhead in TS1. The generational algorithm eliminates the timestamp table overhead by having all variables share the same timestamp. However, the author does not address implementation issues that greatly affect the algorithm's performance.

TPI seems to be a promising software cache coherence scheme. It requires reasonable hardware and runtime overhead and exploits data locality in most situations. The limitation of this method is that it does not always exploit data locality in the presence of nested looping structures in which serial loops enclose one or more parallel loops. These patterns are often encountered in scientific programs where loop bounds are commonly parameterized.

In this paper, we propose a software cache coherence scheme which combines the advantage of the TS1 and TPI scheme and overcomes their limitations. In addition, the performance of our method can be improved without incurring extra hardware or runtime penalty when better information is available at compile time. The hardware overhead of our scheme is close to that of TPI and TS1 with parallel invalidation mechanism. Simulation results show that the cache hit ratio of our scheme is higher than that of TPI and almost the same as that of TS1. By reducing the runtime overhead, our scheme achieves better performance than TS1. Thus, the overall performance of our method is superior to both TPI and TS1.

The rest of the paper is organized as follows. In section 2 we describe the parallel computation model used in this work. A survey of the previous timestamp-based software cache coherence methods is given in section 3. In section 4 the timestamp-based selective invalidation software cache coherence scheme is presented. In section 5 a detailed comparison between our method and the previous methods is given. Section 6 reports simulation results. Section 7 summarizes the major contributions of this work.

2 The Computation Model

A parallel program is composed of a series of **levels**. Each level is either a parallel loop with no internal synchronization (e.g., a DOALL loop) or a serial region (e.g., a DOSER loop) between parallel loops. Serial regions can be nested serial loops or those parts of serials loops that are not in parallel loops. The

*This work is supported in part by the NSF awards CCR-9157371 and ASC-9318185 to the University of Pittsburgh. Contact author: xyuan@cs.pitt.edu.

execution of the program is composed of a series of **epochs**. Each epoch consists of one or more tasks which run in parallel. A task is the minimum computation unit that can be scheduled and assigned to a processor for execution at run time. In a serial epoch, a single task is scheduled and executed on a single processor. In a parallel epoch, multiple tasks are created at runtime and executed on multiple processors simultaneously.

```
(1)   DOALL i = 1, n
(2)       u(i) = 0.0
(3)       v(i) = 0.0
(4)   END DOALL
(5)   DOSER i = 1, n
(6)       DOALL j = 1, n
(7)           w(j) = u(j) + v(j)
(8)           x(j) = ...
(9)       END DOALL
(10)      DOSER j= 1, n
(11)          DOALL k = 1, n
(12)              w(k) = w(j) + a
(13)          END DOALL
(14)          a = a* 2.0
(15)      END DOSER
(16)      DOALL j = 1, n
(17)          .. = x(..)
(18)          w(..) = ..
(19)      END DOALL
(20)  END DOSER
```

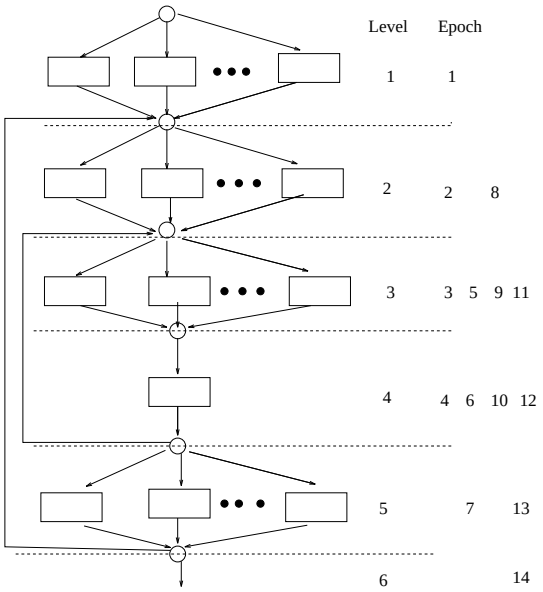

Figure 1: A program and its levels and epochs.

To model dynamic scheduling schemes, we assume that a task may be scheduled on any processor at runtime. We also assume that every processor participates in the execution of every epoch. If there is no useful task to be assigned, the processor will run a task that only performs the coherence operations.

An example parallel program, as well as its levels and epochs during the execution of the program for n = 2, is shown in Fig. 1. This example will also be used in section 4 to describe our cache coherence scheme. The parallel loop in lines (1)-(4) corresponds to Level 1. Level 2 corresponds to lines (6)-(9), level 3 to the parallel loop in lines (11)- (13), level 4 to line (14) and level 5 to lines (16) - (19). In the execution of the program, the epoch number can be much larger than the level number.

In this paper, we initially assume that a cache line is one word long and that the cache uses a write-through policy. Later we also briefly discuss extensions to handle multi-word cache lines. Data variables are classified into private, shared read-only and shared read-write. Only shared read-write variables can cause cache coherence problems. When we refer to a variable, it is assumed to be a shared read-write variable.

3 Timestamp Based Methods

The Version Control Method

In the version control method [3], the *current version number* (CVN) for each variable is kept in the *variable ID table* in each processor. An entire array is treated as a single variable. For the method to be efficient, the variable ID table must be accessed in parallel with the cache access. Each cache line has an extra field called *birth version number* (BVN). At the time the cache line is created (by either a read or a write), the value of CVN (for read misses) or $CVN+1$ (for writes) is written into the BVN. At the end of each level, the processor increments the CVN for each variable that might have been modified in that level. The compiler is responsible for determining the level boundary and generating code to increase the CVNs. The cache line is valid only if its BVN is bigger than or equal to the corresponding variable's CVN. The version control method is effective in preserving the reuse of cache lines. The major limitation of this method is the hardware and runtime overhead. The timestamp method [10] is not discussed here because it is similar to the version control method.

The TS1 Method

In TS1, one additional bit, referred to as the *epoch bit*, is required for each cache line. The compiler determines the levels and the variables modified in each level. The epoch bit is reset at the end of each level and is set when the cache line is referenced. At the end of each level, invalidation instructions are issued to invalidate all the variables modified in that level. An invalidation instruction invalidates a specified cache line only if (1) the address tag matches, and (2) the cache line's epoch bit is not set. Once a variable is modified in an epoch, the variable is invalidated in all the processors except the one that modified it. Thus, the cache will always contain valid cache lines.

The performance and overhead of this scheme depends heavily on the mechanism used to invalidate the stale cache lines. Two implementations of the invalidation mechanism are proposed in [8]. The simple and inexpensive invalidation mechanism uses a low level invalidate instruction which could invalidate either a particular line or a particular page. The high level invalidate would then loop over the proper range of pages and lines. Using this simple *serial* scheme, the invalidation overhead is $O(\sum_{i=1}^{n}(s_i))$, where s_i is the size of the ith section to be invalidated. A faster, but more complex invalidation method was proposed in [8]

and [9]. This *parallel* scheme requires a full associated address tag memory. A bit mask is used to determine which addresses to invalidate. Using this invalidation scheme, the runtime overhead is $O(\sum_{i=1}^{n} log(s_i))$. Besides the invalidation schemes, the runtime overhead of TS1 also depends on the program structure when using precise invalidation. When an epoch contains many discontinuous sections to be invalidated, the overhead for the epoch will be quite large.

In summary, TS1 effectively preserves the reuses of a cache line. Using precise invalidation, this scheme can achieve the best cache hit ratio that any software cache coherence method based on local knowledge can possibly achieve [8]. The limitation is the runtime overhead. The exploitation of higher cache hit ratio by using precise invalidation results in larger runtime overhead. Therefore, it is desirable to develop a mechanism to perform the invalidation more effectively.

The Generational Algorithm

The goal of the generational algorithm is to improve over the version control method by making all the variables in the program share one common CVN. In the generational method, the shared CVN called *Current generation number* (CGN) is stored in each processor. The CGN is increased at the end of each epoch. When a cache line is updated, the cache line is provided with a *valid generation number* (VGN) indicating when the cache line will become invalid. The system invalidates a cache line implicitly by causing CGN to become larger than the cache line's VGN. The compiler is responsible for determining the VGN for all the memory references. By using a common CVN, the generational algorithm eliminates the variable ID table and the problems associated with it. However, [5] does not address some important issues. For example, it is not clear how the VGNs of the cache lines are updated for the variables modified in a level. If every cache line's VGN is updated individually at the end of each epoch, this method will incur greater runtime overhead than TS1 does. Since the author does not address these important issues, we are unable to determine the efficiency of the method.

The TPI Scheme

In TPI, the timestamp field associated with the cache line is updated with current epoch number for both read and write instructions. The read instruction is supplied with an additional field to indicate the epoch number of the last write to the variable. The cache copies created after the specified last write epoch are valid. Therefore, the stale cache copies are detected by comparing the timestamp field with the last write epoch number. When the current epoch counter overflows, an explicit invalidation instruction is used to invalidate the cache lines. As shown in [6], the method can preserve most of the localities with reasonable hardware and runtime overhead. Furthermore, interprocedural analysis techniques can be incorporated in this method to preserve the locality across procedure boundaries [7]. The limitation of this method is that it does not always handle nested looping structures shown in Fig. 2 effectively.

The *repeated read pattern* occurs when a shared variable is only read inside a nested loop and is written in some epochs before the loop. An example is shown in Fig. 2 (a). Consider the read reference of x in line (6). For each distinct iteration of the serial loop, the last write epoch for this reference is different. However, since the read instruction can only carry one timestamp, it cannot preserve all this information even if it can be determined by the compiler. For the program to run correctly, the last write epoch number for the reference is two epochs earlier. As a result, the references to x in line (6) result in cache misses since the cache copies created inside the serial loop are separated by three epochs. Note that in TPI, the DOSER is treated as serial epoch.

```
(1) DOALL i = 1, 1000
(2)     x(i) = ...
(3) END DOALL
(4) DOSER i = 1, N
(5)     DOALL j = 1, 1000
(6)         .. = x(j)...
(7)     END DOALL
(8)     ...  ! another level
(9) END DOSER
```

(a) Repeated read pattern.

```
(1) DOALL i = 1, 1000
(2)     x(i) = ...
(3) END DOALL
(4) DOSER i = 1, N
(5)     DOALL j = 1, 1000
(6)         ... = x(j)...
(7)         x(j) = ...
(8)     END DOALL
(9)     ...  ! another level
(10)END DOSER
```

(b) Write interference pattern.

```
(1) DOSER i = 1, 1000
(2)         DOSER j = 1, N
(3)                 DOALL k = 1, 1000
(4)                     ...
(5)                 END DOALL
(6)         END DOSER
(7)         DOALL j = 1, 1000
(8)             ... = x(j)...
(9)             x(j) = ...
(10)        END DOALL
(11)END DOSER
```

(c) Unknown epoch number pattern.

Figure 2: Examples of reference patterns.

The *write interference pattern* occurs when (1) a shared variable is written before a nested loop and is read and written inside the loop, (2) the write to the variable inside the loop takes more epochs to reach the read than the write outside the loop does. Consider the example in Fig. 2 (b). Both the writes to x in line (2) and (7) reach the read of x in line (6). Since the write in line (2) crosses only two epochs to reach the read while the write in line (7) crosses three epochs, the last write epoch number for the read is two epochs earlier. Therefore, except first iteration, the reads in line (6) result in cache misses.

The *unknown epoch number pattern* occurs when the runtime number of epochs inside a nested loop is unknown at compile time. Consider the example in Fig. 2 (c). The epochs between the write in line (9) and the read in line (8) are unknown at compile time due to the value of N being unknown. The compiler must conservatively assume that it takes two epochs for the write in line (9) to reach the read in line (8). Therefore, the last write epoch for the read in line (6) is two epochs earlier. However, in the execution of the program, the level in lines (2) - (6) is usually executed. As a result, the reads in line (8) will cause cache misses.

Some compiler techniques, such as epoch number

adjustment, loop peeling and guard execution, can be used to alleviate some of the problems. However, we believe that the static estimation of the last write epoch number, which is changed dynamically, is the inherent limitation of this method. Therefore, it would be desirable to use a static measurement to decide whether a cache line is valid.

4 Timestamp-based Selective Invalidation Scheme

Both TS1 and TPI improve the version control to some extent. Each of the method has its own limitation – the runtime overhead in TS1 and the degradation of performance due to static estimation of dynamic epoch number in TPI. In this paper, we propose a timestamp-based selective invalidation scheme (TBSIS). This scheme combines the ideas of both TS1 and TPI and overcomes their limitations. TBSIS improves over TS1 by having almost the same capability of preserving the cache line reuses and reducing the runtime overhead. TBSIS improves over TPI by having a static measurement, the level number, to decide whether a cache line is valid and avoiding all the problems in TPI when dealing with nested loops. The hardware overhead of TBSIS is close to that of TPI and TS1 with parallel invalidation. Simulation results show that TBSIS exhibits better performance than both TS1 and TPI.

4.1 Hardware Support

A timestamp field called *Invalidation Level Number* (ILN) is associated with each cache line. The ILN is composed of two parts, ILN_m and ILN_r. The ILN_m is the most significant bit in the ILN and the ILN_r represents the remaining bits. We will denote an ILN as (ILN_m, ILN_r).

A special invalidation instruction must be supported by the cache implementation. We will use $INV\ L$ to represent the instruction, where L is a level number. The invalidation instruction operates on all the cache lines in the cache. The operations of the instruction $INV\ L$ are (1) to invalidate all cache lines whose $ILN_r = L$ and $ILN_m = 0$, (2) to reset the ILN_m fields for all cache lines whose $ILN_r = L$ and $ILN_m = 1$. Other cache lines are unaffected.

The ILN_m bit is used to skip one invalidation. As we will see later, this bit is mainly used to deal with the looping structures. The logic for the instruction is shown in Fig. 3. We assume that the cache implementation can support this instruction in O(1) time. The processor also needs to support the memory reference instructions Read_ILN and Write_ILN. Both instructions are augmented with an extra ILN field. Beside the traditional operations, these instructions also update the ILN fields in the cache lines.

4.2 The Scheme

TBSIS is a generalization of TS1 [8]. The main idea is still to explicitly invalidate the stale cache copies if the variable is modified in the current epoch. By using more bits as timestamp for each cache line and augmenting the compiler support, the explicit invalidation in TBSIS can be carried out more efficiently.

In TBSIS, an invalidation instruction $INV\ L$ is issued at the end of each epoch to invalidate all the cache lines that must be invalidated in that epoch. Here L is the level number corresponding to the epoch. In order for the instruction to invalidate all the stale cache lines, all these lines that are still valid before the instruction should have $ILN_r = L$ and $ILN_m = 0$. The compiler determines the levels of the program, and furthermore, it also determines a proper ILN for each memory reference to ensure that the cache does not contain the stale cache copies.

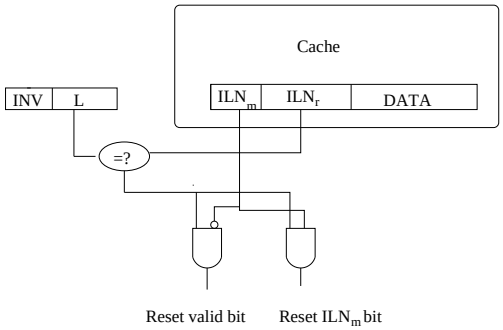

Figure 3: The logic of the invalidation instruction.

It is simple to ensure a coherent cache by using the ILN field and the invalidation instruction. For example, the compiler may provide each memory reference with ILN equal to the current level number. TBSIS is then reduced to the simple invalidation scheme [3] which can only exploit the intra-level locality. To exploit maximum locality, the compiler should always try to set ILN to the next write level number.

4.3 Software Support

The major task of software support is to determine the next write level number for each memory reference. Due to the branches in programs, there may be several next write levels for a memory reference. The compiler must approximate the next write level and/or introduce extra invalidation instructions at selected points to handle such situations. Two approachs can be used to handles the multiple next write level situation. The conservative approach assigns the ILN conservatively and guarantee the program runs correctly through all paths. The aggressive approach assigns the ILN to be the one that saves the cache lines along the most frequently executed path. This approach needs to introduce extra invalidation instructions along other paths to ensure the correctness of the program. TBSIS adopts the second approach and therefore, the compiler must determine the next write level and find the proper place to introduce the extra invalidation instructions.

For a memory reference, if there is a unique next write level, then that level number will be the ILN for the reference. There may be several possible next write levels due to the branches in the program. If the branch is a forward branch, the nearest level at which the variable is modified is used as the next write level. If there is a backward branch involved, the memory reference is inside a loop. The next write level in the innermost loop is considered as the next write level. In this way, TBSIS always captures the locality inside the loop. Using *epoch flow graph* [6], the next write level information can be obtained in two steps. First,

using the algorithm in [1], we determine the looping structures in the flow graph. Then, for each memory reference, the compiler searches forward for all the writes to the same variable that can be reached by the reference in the flow graph. Among the writes, the write that is closest to the current point in the innermost loop is the write of interest.

As for the placement of the extra invalidation instructions, a simple solution is to introduce extra invalidation instructions at the exit of every loop to invalidate all the cache lines whose ILN is equal to any one level inside the loops. The invalidation instruction can be delayed until the first level that modifies the variables that are modified in the loop. The compiler can use similar analysis as in the determination of the next write level for one variable to determine the place for the extra invalidation instruction. In the remainder of this paper, we assume that the simple approach is used.

```
(1)  DOALL i = 1, n              Level   Var    ILN
(2)      u(i) = 0.0                1      u     (0, 6)
(3)      v(i) = 0.0                       v     (0, 6)
(4)  END DOALL
     INV 1
————————————————————
(5)  DO i = 1, n                          u     (0, 6)
(6)      DOALL j = 1, n                   v     (0, 6)
(7)          w(j) = u(j) + ....    2      w     (0, 3)
(8)          x(j) = v(j) + x(j)           x     (1, 2)
(9)      END DOALL
         INV 2
————————————————————-
(10)     DO j= 1, n
(11)         DOALL k = 1, n               a     (0, 4)
(12)             w(k) = w(j) + a   3      w     (1, 3)
(13)         END DOALL
             INV 3
————————————————————-
(14)         a = a* 2.0            4      a     (1, 4)
             INV 4
————————————————————
(15)     END DO
         INV 3, 4          ! extra inv
(16)     DOALL j = 1, n
(17)         .. = x(..)            5      x     (0, 2)
(18)     END DOALL
         INV 5
————————————————————————————
(19) END DO
     INV 2, 3, 4, 5          ! extra inv
```

Figure 4: An example.

4.4 An Example

In this section, we will describe the method through an example. Fig. 4 is the example program in section 2 augmented with the invalidation instructions. The value of ILN is (0, 6) for the write reference to u in line (2), this is because the variable u is not modified after line (2). Since there is no *INV 6* issued in the program, the cache copies will be valid throughout the execution of the program. The value of ILN is (1, 3) for the write reference to w in line (12) since the reference is inside the serial loop and the next write level is the next execution of level 3. Therefore, the cache copies of w will be valid until the end of the next iteration and the read reference to w at line (12) will be a cache hit. The next write level for the write reference to x in line (8) is the next execution of level 2. Therefore, the cache copies should survive one *INV 2* instruction at the end of the level 2 and the ILN should be set to (1, 2). Assuming the variable n is equal to 2, Table 1 depicts the ILN of each variable during the execution of the program.

Table 1: Cache lines and their ILN.

Ep.	L	a	u	v	w	x
1	1	-	(0, 6)	(0, 6)	-	-
2	2	-	(0, 6)	(0, 6)	(0, 3)	(0, 2)
3	3	(0, 4)	(0, 6)	(0, 6)	(0, 3)	(0, 2)
4	4	(0, 4)	(0, 6)	(0, 6)	(0, 3)	(0, 2)
5	3	(0, 4)	(0, 6)	(0, 6)	(0, 3)	(0, 2)
6	4	(0, 4)	(0, 6)	(0, 6)	(0, 3)	(0, 2)
7	5	-	(0, 6)	(0, 6)	-	(0, 2)
8	2	-	(0, 6)	(0, 6)	(0, 3)	(0, 2)
9	3	(0, 4)	(0, 6)	(0, 6)	(0, 3)	(0, 2)
10	4	(0, 4)	(0, 6)	(0, 6)	(0, 3)	(0, 2)
11	3	(0, 4)	(0, 6)	(0, 6)	(0, 3)	(0, 2)
12	4	(0, 4)	(0, 6)	(0, 6)	(0, 3)	(0, 2)
13	5	-	(0, 6)	(0, 6)	-	(0, 2)

4.5 Multi-word Cache Lines

TBSIS can be extended to handle multi-word cache lines. Let us assume that each cache word is associated with an ILN and that arrays are aligned to the cache line boundary. For a cache miss, if the reference is to an array element, then all the ILNs are assigned the ILN value in the instruction. If the reference is to a scalar, only the ILN of the cache word corresponding the scalar is assigned the ILN in the instruction, all other ILNs in the cache line are assigned the current level number. Therefore, those scalar cache lines that are not accessed in the current epoch will be invalidated at the end of the epoch and the cache contains clean data.

TBSIS can also be adapted if a single ILN is associated with an entire cache line. If arrays are aligned to cache line boundaries, then a single ILN per cache line is sufficient to exploit spatial locality. To maintain cache coherence for scalars, a reference to a scalar sets the ILN to be the minimum ILN of all the variables in the corresponding cache line. A frequently accessed shared scalar can be placed in a separate cache line.

5 Some Comparisons

TBSIS vs TS1

Compared to TS1 with serial invalidation, TBSIS requires a higher hardware overhead. However, TBSIS has significantly less runtime overhead. In our simulation study, TBSIS has an average of 11.8% speedup on a 16 processor system and 30.9% speedup on a 64 processor system with regard to the total memory reference time against TS1 with serial invalidation. The speedup is gained by reducing the runtime overhead.

TS1 with parallel invalidation and TBSIS have comparable hardware overhead. TBSIS requires fully

```
DOALL m = 1, 1000              Level
    x (m) = ...                  i
END DOALL              if (m is odd) ILN = j
INV i                    else ILN = k
————————————————
...
————————————————
DOALLl m = 1, 999, 2
    x(m) = ...                   j        ILN = l
END DOALL
INV j
————————————————
...
————————————————
DOALL m = 2, 1000, 2
    x(m) = ...                   k        ILN = l
END DOALL
INV k
————————————————
...
————————————————
DOALL m = 1, 1000
    x(m) = ...                   l
END DOALL
INV l
————————————————
```

Figure 5: Precise invalidation.

associative ILN memory (6 bits per word) and a regular address tag memory. TS1 requires a fully associative address tag and epoch bit memory (around 20 bits per word) with a bit mask register. TBSIS has a better runtime overhead, especially when exploiting precise invalidation. Consider the example in Fig. 5. To exploit precise invalidation, TS1 requires multiple invalidation instructions to be issued at the end of an epoch. For instance, each processor must issue 499 invalidation instructions in TS1 at the end of level k. In TBSIS, precise invalidation is done by assigning proper ILN for memory reference instruction. Regardless of the program structure, only 1 invalidation instruction is needed in TBSIS. However, conditional assignment statements may be needed to assign proper ILNs as shown in Fig. 5. The disadvantage of TBSIS is the potential over–invalidation when the program exits a nested loop. However, the misses caused by the over–invalidation is almost negligible compared to the total number of references since most of the references are inside the nested loop. In our performance study, TBSIS has an average speedup of 1.8% for 16 processor systems and 14.9% for 64 processor systems against TS1 with parallel invalidation.

TBSIS vs TPI

In comparison to TPI, TBSIS requires a slightly larger overhead. However, TBSIS always exploits data locality inside loops, which accounts for a significant fraction of the data locality in a program, while TPI fails to preserve cache lines in situations where memory references are separated by non-unique number of epochs. Simulation results confirm that our method preserves the reuse of cache lines more effectively. Next, we show how TBSIS handles the patterns for which TPI fails to exploit the locality.

Fig. 6 shows how the repeated read pattern is handled in our method. The read reference to x in line (6) has $ILN = (0, 4)$. Since the instruction $INV\ 4$ is

```
(1) DOALL i = 1, 1000        Level   var   ILN
(2)     x(i) = ...             1      x    (0, 4)
(3) END DOALL
    INV 1
  ————————————————————
(4) DO i = 1, N
(5)     DOALL j = 1, 1000
(6)         .. = x(j)...       2      x    (0, 4)
(7)     END DOALL
        INV 2
  ————————————————————
(8)     ...                    3
        INV 3
  ————————————————————
(9) END DO
```

Figure 6: The repeated read pattern.

not executed until the end of the program, the cache lines created in line (6) will be valid throughout the program under TBSIS. Fig. 7 shows how the write interference pattern is handled. The write reference to x in line (7) has the $ILN = (1, 2)$. Therefore, the cache line created by the statement can survive till the next execution of level 2 and the read references to x in line (6) will result in cache hits. Fig. 8 shows how the unknown epoch number pattern is handled. The write reference to x in line (9) have the ILN = (1, 2), therefore, the cache line created by that statement can survive till the next execution of level 2 and the read references to x in line (8) will result in cache hits no matter how many epochs are executed at level 1.

```
(1) DOALL i = 1, 1000        Level   var   ILN
(2)     x(i) = ...             1      x    (0, 2)
(3) END DOALL
    INV 1
  ————————————————————
(4) DO i = 1, N
(5)     DOALL j = 1, 1000
(6)         ... = x(j)...      2      x    (1, 2)
(7)         x(j) = ...
(8)     END DOALL
        INV 2
  ————————————————————
(9)     ...
        INV 3
  ————————————————————
(10) END DO
```

Figure 7: The write interference pattern.

Counter Overflow and Timestamp Size

Many of the timestamp based methods, such as the version control method and TPI, suffer from the counter overflow problem. Once the timestamp counter is reset, additional actions must be taken to ensure the cache coherence. Our method does not have the counter overflow problem. Using a shorter timestamp only results in the performance penalty for not being able to preserve the cache line for a longer time, while in other methods, using a shorter timestamp results in an additional overhead for dealing with counter overflow. Besides, TBSIS preserves cache lines for 2^{n-1} static levels which is usually much larger than the 2^{n-1} dynamic epochs. Thus, our method can use a shorter timestamp to provide the same cache performance.

```
(1)  DO i = 1, 1000          Level    var    ILN
(2)     DO j = 1, N
(3)       DOALL k = 1, 1000
(4)         ...                 1
(5)       END DOALL
          INV 1
  ————————————————————
(6)     END DO
        INV 1  ! extra INV instruction
(7)     DOALL j = 1, 1000
(8)       ... = x(j)...
(9)       x(j) = ...          2      x      (1, 2)
(10)    END DOALL
        INV 2
  ————————————————————
(11) END DO
```

Figure 8: The unknown epoch number pattern.

6 Performance Study

To compare TBSIS with other schemes, simulations were conducted on three parallel programs for three software cache coherence protocols: TS1, TPI and TBSIS. We study both cache hit ratio and total number of cycles for memory reference, which includes the memory reference cycles and overhead cycles.

The benchmarks are written in CRAFT FORTRAN and are executed on a CRAY-T3D. The simulation is carried out by adding code in the source program that simulates the cache in each processor. For each memory reference that refers to the shared data, a routine is called to simulate the effect of the cache. We assume that no interprocedural analysis is performed by the compiler and thus, all the cache lines are invalidated at the subroutine boundaries.

Table 2: General characteristics.

	lp	tscf	ep
total references($\times 10^6$)	70.8	114.4	33.6
% of data read	73.1	65. 0	62.3
% of data write	26.9	35.0	37.7
number of epoch	5869	3041	4148
number of level	11	18	8

The first benchmark program, *lp*, uses Gauss-Siedel iterations to solve Laplace equations on a 64×64 discretized unit square with Dirichlet boundary conditions. The second program *tscf*, is a program that simulates the evolution of a self-gravitating system using a self consistent field approach. The third program *ep* is the NAS's embarrassingly parallel benchmark [2]. The outermost loop iteration number of the *ep* program is reduced by a factor of 256 to shorten the simulation time. Table 2 shows the general characteristics of the three benchmarks used in our experiment. The epoch number is obtained from the execution of the program on 32 processors. The level numbers are the maximum level number among all subroutines in each program.

The architecture simulated consists of direct-mapped, write through caches. The block size is one word. Only the references to the shared data are simulated. We use 6–bit timestamp for TPI and TBSIS. Table 3 shows the simulation results for different cache sizes using 16 processors. Table 4 shows the result for 64K word cache using different number of processors.

Table 3: Hit ratio for different cache size (16 PEs).

scheme	prog	16KW	32KW	64KW	128KW
TS1	ep	74.33	81.97	85.28	93.71
	lp	85.64	97.86	97.86	97.86
	tscf	70.27	70.62	70.62	70.62
TPI	ep	73.50	74.74	75.89	83.60
	lp	82.82	88.94	88.95	88.95
	tscf	70.27	70.62	70.62	70.62
TBSIS	ep	74.33	81.97	85.28	93.71
	lp	85.64	97.86	97.86	97.86
	tscf	70.27	70.62	70.62	70.62

Table 4: Hit ratio for different PE numbers (64KW cache).

scheme	prog	16 PEs	32 PEs	64 PEs
TS1	ep	85.28	92.84	96.51
	lp	97.86	95.39	90.98
	tscf	70.62	70.61	70.59
TPI	ep	75.89	82.72	85.65
	lp	88.95	86.20	82.18
	tscf	70.62	70.61	70.59
TBSIS	ep	85.28	92.84	96.51
	lp	97.86	95.39	90.98
	tscf	70.62	70.61	70.59

For the *ep* program, 64k word cache size is not large enough to exploit all the data locality in 16 processor systems. Therefore, the hit ratio increases with the number of processors, since the total cache size increases with the number of processors. In a 16 processor system, TS1 and TBSIS have better hit ratio than TPI. TS1 actually has a slightly higher cache hit ratio. But the difference is so small that it doesn't show in the table where values are rounded off. For the *lp* program, 64K word cache size is large enough to exploit all the possible data locality. Therefore, when the number of processors is increased, the hit ratio decreases significantly. TS1 and TBSIS have better hit ratio than TPI. TS1 actually has a slightly higher cache hit ratio when the cache size is large enough. But the difference is so small that it doesn't show in the table. For *tscf* program, the cache hit ratio is almost the same when the number of processors increases. All the three cache coherence methods have exactly the same cache hit ratio. The reason is the extensive usage of subroutine in this program. Without interprocedural analysis, the software cache coherence methods fail to exploit most of inter-level locality.

Table 5: Total memory reference cycles($\times 10^6$).

prog.	PE	TPI	TS1(ser.)	TS1(par.)	TBSIS
ep	16	21.88	14.55	14.70	14.19
	32	8.14	4.36	4.52	4.00
	64	3.47	1.62	1.77	1.26
lp	16	23.49	10.42	8.26	8.13
	32	14.12	8.49	6.33	6.20
	64	8.80	7.30	5.14	5.01
tscf	16	89.09	93.17	89.29	89.10
	32	44.56	48.64	44.76	44.57
	64	22.30	26.37	22.50	22.30

Table 5 compares the total cycles for memory ref-

Table 6: Runtime overhead percentage.

prog.	PE	TPI	TS1(ser.)	TS1(par.)	TBSIS
ep	16	0.0	2.6	3.6	0.1
	32	0.0	8.6	11.7	0.2
	64	0.1	23.1	29.9	0.3
lp	16	0.0	22.1	1.8	0.1
	32	0.0	27.1	2.3	0.2
	64	0.1	31.5	2.9	0.2
tscf	16	0.0	4.4	0.2	0.0
	32	0.0	8.4	0.4	0.0
	64	0.0	15.5	0.9	0.0

erences. The total cycles consist of memory reference cycles and protocol overhead cycles. We assume that a cache hit takes 1 cycle, a parallel invalidation in TS1 and TBSIS takes 2 cycles, a serial invalidation to invalidate a page of 512 words or a certain cache word in TS1 takes 1 cycle, a cache miss takes 40 cycles. Table 6 shows the runtime overhead percentage of all the protocols. TPI incurs the least runtime overhead. However, it has lower cache hit ratio than both TS1 and TBSIS. Thus the performance of TPI is worse than that of TS1 and TBSIS. The overhead incurred by TS1 with serial invalidation ranges from 2.6% to 22.1% in 16 processor system and 15.5% to 31.5% in 64 processor system. With parallel invalidation scheme, TS1 greatly reduces the runtime overhead. However, we still observe 29.9% overhead in *ep* program with 64 processors. Since TBSIS incurs constant overhead in each epoch and achieves almost the same cache hit ratio as TS1, its performance is better. TBSIS is faster than other methods in the *ep* and *lp* programs. For the *tscf* program, TPI has slightly higher performance than TBSIS. The speedup of TBSIS is shown in Table 7.

Table 7: Percentage speedup of TBSIS.

prog	PE	TS1(ser.)	TS1(para.)	TPI
ep	16	2.5	3.6	54.2
	32	9.2	13.1	103.8
	64	28.7	41.1	176.5
lp	16	28.2	1.7	186.4
	32	36.9	2.1	127.8
	64	45.7	2.7	75.7
tscf	16	4.6	0.2	0.0
	32	9.1	0.4	0.0
	64	18.3	0.9	0.0

Table 8: Effect of invalidation cycle.

cycles	TS1(para. inv.)		TBSIS	
	overhead	percent	overhead	percent
2	0.147	2.3%	0.012	0.2%
4	0.294	4.5%	0.024	0.4%
8	0.588	8.8%	0.048	0.8%
16	1.176	16.0%	0.096	1.5%

Table 8 shows the effect of invalidation cycles on the total performance for both TS1 and TBSIS. We study the case of the *lp* program in 32 processor system. As shown in the table, the performance of TS1 is greatly affected by the efficiency of invalidation scheme while TBSIS sustains a graceful degradation and achieves reasonably good performance with slow invalidation.

7 Conclusion

In this paper, we propose a new software cache coherence protocol based on the timestamp concept. Our method combines the ideas in TS1 and TPI and overcomes their limitations. Furthermore, with improved compiler techniques, the performance of our protocol can be further improved without extra hardware and runtime overheads. In our protocol, the timestamp is used for the invalidation of cache lines. By using the timestamp, selective invalidation of O(1) time is possible. This scheme has the characteristic of high cache performance and low hardware and runtime overhead.

References

[1] Aho, A.V., Sethi, R. and Ullman, J.D. "Compilers: Principles, Techniques and Tools." Addison-Wesley, Reading, Massachusetts, 1986.

[2] Bailey, D. et al. "The NAS Parallel Benchmarks." RNR Technical report, RNR-94-007, March, 1994.

[3] H. Cheong and A. Veidenbaum "Compiler-directed Cache Management for Multiprocessor." *IEEE Computer*, 23(6):39-47, June 1990.

[4] H. Cheong, "Life-span strategy - A Compiler-based Approach to Cache Coherence." In *Proceedings of 1992 International Conference on Supercomputing*, 1992.

[5] T. Chiueh, "A Generational Algorithm to Multiprocessor Cache Coherence." In *Proceedings of the 1993 International Conference on Parallel Processing*, pages I20-I24, 1993.

[6] L. Choi and P. Yew "A Compiler-Directed Cache Coherence Scheme with Improved Intertask Locality." In *Supercomputing'94*, pages 773-782, 1994.

[7] L. Choi and P. Yew "Interprocedural Array Data–Flow Analysis for Cache Coherence." In *8th Intl. Workshop on Languages and Compilers for Parallel Comp.*, pages 6.1-6.15, Aug. 1995.

[8] E.Darnell and K. Kennedy "Cache Coherence Using Local Knowledge." In *Supercomputing'93*, pages 720-729, 1993.

[9] A. Louri and H. Sung. "A Compiler Directed Cache Coherence Scheme With Fast and Parallel Explicit Invalidation." In *Proc. of the 1992 International Conference on Parallel Processing*, pages 2-9, Aug 1992.

[10] S. Min and J. Baer. "Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps." *IEEE Trans. on Parallel and Dist. Systems*, 3(1):25-44, Jan. 1992.