Compiler and Hardware Support for Cache Coherence in Large-Scale Multiprocessors: Design Considerations and Performance Study

Lynn Choi
University of Illinois at Urbana-Champaign
Center for Supercomputing R & D
Urbana, IL 61801-1351
lchoi@csrd.uiuc.edu

Pen-Chung Yew
University of Minnesota
Department of Computer Science
Minneapolis, MN 55455-0519
yew@cs.umn.edu

Abstract

In this paper, we study a hardware-supported, compiler-directed (HSCD) cache coherence scheme, which can be implemented on a large-scale multiprocessor using off-the-shelf microprocessors, such as the Cray T3D. It can be adapted to various cache organizations, including multi-word cache lines and byte-addressable architectures. Several system related issues, including critical sections, inter-thread communication, and task migration have also been addressed. The cost of the required hardware support is small and proportional to the cache size. The necessary compiler algorithms, including intra- and interprocedural array data-flow analysis, have been implemented on the Polaris compiler [17].

From our simulation study using the Perfect Club benchmarks, we found that, in spite of the conservative analysis made by the compiler, the performance of the proposed HSCD scheme can be comparable to that of a full-map hardware directory scheme. With its comparable performance and reduced hardware cost, the scheme can be a viable alternative for large-scale multiprocessors, such as the Cray T3D, that rely on users to maintain data coherence.

1 Introduction

Many commercially available large-scale multiprocessors, such as the Cray T3D and the Intel Paragon, do not provide hardware-coherent caches due to the expensive hardware required for such mechanisms. They instead provide software mechanisms and rely mostly on users to maintain data coherence either through language extensions or message-passing paradigms.

In several early multiprocessor systems, such as the CMU C.mmp, the NYU Ultracomputer, the IBM RP3, and the Illinois Cedar, compiler-directed techniques were used to solve the cache coherence problem. In this approach, cache coherence is maintained locally without the need for interprocessor communication or hardware directories. The C.mmp was the first to allow read-only shared data to be kept in private caches while leaving read-write data uncached. In the Ultracomputer, the caching of the read-write shared data is permitted only in program regions in which the read-write shared data are used exclusively by one processor. The special memory operations *release* and *flush* are inserted into the user code at compile time to allow the caching of the read-write variables during "safe" intervals. The RP3 uses a similar technique but allows a more flexible granularity of data. At compile time, the compiler assigns two attributes, "cacheable" and "volatile", to each data object. Invalidate instructions in various sizes of data, such as a line, a page or a segment unit, are supported. The Cedar uses a shared cache to avoid coherence problems within each cluster. By default, data is placed in cluster memory, which can be cached. But the programmer can place data in global memory by specifying an attribute called "global" for these data. Furthermore, data movement instructions are provided so that the programmer can explicitly move data between the cluster and global memories. By using these software mechanisms, coherence can be maintained for globally shared data.

Several compiler-directed cache coherence schemes [5, 6, 7, 9, 11, 15] have been recently proposed. These schemes give better performance, but demand more hardware and compiler supports than the previous schemes. They require a more precise program analysis to maintain coherence on a reference basis [5, 9] instead of a program region basis compared to the previous schemes. In addition, these schemes require hardware support to maintain local runtime cache states. In this regard, the terminology *software cache coherence* is a misnomer. It is a *hardware approach with strong compiler support*. We call them *hardware-supported compiler-directed (HSCD)* coherence schemes, which is distinctly different from a pure hardware directory scheme and a pure software scheme.

Several studies have compared the performance of directory schemes and some recent HSCD schemes. Min and Baer [15] compared the performance of a directory scheme and a timestamp-based scheme assuming an infinite cache size and single-word cache lines. Lilja [14] compared the performance of the version control scheme [6] with directory schemes, and analyzed the directory overhead of several implementations. Both studies show that the performance of those HSCD schemes can be comparable to that of the directory schemes. Adve et al [1] used an analytic model to compare the performance of compiler-directed and directory-based techniques. They concluded that the performance of compiler-directed schemes depends on the characteristics of the workloads. Chen [4] showed that a simple invalidation scheme can achieve performance comparable to that of a directory scheme and discusses several different write policies. Most of those studies, however, assumed perfect compile-time memory disambiguation and complete control dependence information. They did not provide any indication of how much performance can be obtained when implemented on a real compiler.

Also, most of the HSCD schemes proposed to date have not addressed the real cost of the required hardware support. For example, many of the schemes require expensive hardware support and assume a cache organization with single-word cache lines and a word-addressable architecture. The issues of synchronization, such as lock variables and critical sections, have also rarely been addressed.

In this paper, we address these issues and demonstrate the feasibility and the performance of a HSCD scheme we previously proposed [9]. To study the compiler analysis techniques for the proposed scheme, we have implemented the compiler algorithms on the Polaris compiler. By performing execution-driven simulations on the Perfect Club Benchmarks, we evaluate the performance of our scheme compared to a hardware directory scheme.

2 Background

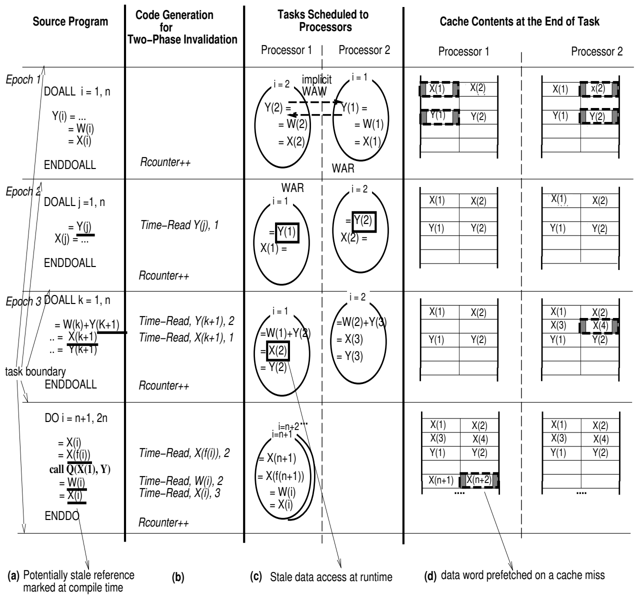

Figure 1: A parallel program and its epochs.

2.1 Stale reference sequence

Parallel execution model The parallel execution model in this study assumes that program execution consists of a sequence of *epochs*. An epoch may contain concurrent light threads (or tasks), or a single thread running a serial code section between parallel epochs. Parallel threads are scheduled only at the beginning of a parallel epoch and are joined at the end of the epoch. For consistency, the main memory should be updated at the end of each epoch. These light threads usually consist of several iterations of a parallel loop and are supported by language extensions, such as DOALL loops (see Figure 1(a)).

Stale reference sequence The following sequence of events creates a stale reference at runtime [21]: (1) Processor P_i reads or writes to memory location x at time T_1, and brings a copy of x in its cache; (2) Another processor P_j $(j \neq i)$ writes to x at time T_2 $(> T_1)$, and creates a new copy in its cache; (3) Processor P_i reads the copy of x in P_i's cache at time T_3 $(> T_2)$, which has become stale. Assuming only a DOALL type of parallelism (no dependences among concurrent threads), the memory events (1) to (3) should occur in different epochs.

However, with multi-word cache lines, there can be implicit dependences due to false sharing. Figure 1 shows a program example (Figure 1(a)), its corresponding memory events at runtime (Figure 1(c)), and the cache contents of each processor (Figure 1(d)). It assumes 2-word cache lines and a write-allocate policy. All caches are empty at the beginning of epoch 1. The read reference to X(2) by processor 1 in epoch 3 is a stale data access since the cache copy is read in epoch 1 but processor 2 creates a new copy in epoch 2.

However, note that there are two additional stale data references which do not conform to the above memory reference pattern. The write reference to Y(1) by processor 2 in epoch 1 will cause a cold start miss, and will load the entire cache line from memory, creating a copy of both Y(1) and Y(2). This implicit read of Y(2) on a write miss causes an implicit WAW (write-after-write) dependence to Y(2) when it is written by processor 1 in epoch 1. Therefore, the following read reference to Y(2) by processor 2 in epoch 2 becomes a stale data reference. A similar situation exists for the reference to Y(1) in epoch 2. The read-write dependence among the variables in the same cache line creates a false-sharing effect. These implicit dependences can also occur between tasks across epoch boundaries.

Because of the implicit dependences in the multi-word cache lines, a sequence of events : *(a) a write, (b) one or more epoch boundaries, and (c) a read*, can potentially create a stale reference. We call this sequence of events a *stale reference sequence* and the read reference in (c) as a *potentially stale data reference.*

Software cache-bypass scheme (*SC*) Once all of the potentially stale data references are identified by the compiler, cache coherence can be enforced if we guarantee that all such references access up-to-date data from main memory rather than from the stale cache copies. This can be enforced by using a bypass cache operation, which accesses the main memory directly.[1] It avoids accessing the cached data, which might be stale, and replaces the cached data with the up-to-date copy from the main memory.

However, this pure compiler scheme has several limitations. First, the read reference to Y(2) by processor 1 in epoch 3 (see Figure 1) will be marked as potentially stale because Y(2) was updated in epoch 1. The scheme cannot determine that the update in epoch 1 and the read in epoch 3 are issued by the same processor due to dynamic runtime scheduling. Second, the read reference to X(f(i)) in epoch 4 cannot be analyzed precisely at compile time due to the unknown index value. To overcome these limitations, we proposed a hardware scheme that keeps track of the local cache states at runtime [9].

[1] This operation can be implemented in the MIPS R10000 processor by a cache block invalidate (*Index Write Back Invalidate*) followed by a regular load operation. Similarly, the Flush instruction (*DCBF*) can be used for the IBM PowerPC 600 series microprocessor.

2.2 Two-phase invalidation scheme (*TPI*)

In our proposed HSCD scheme, called the *Two-Phase Invalidation (TPI) scheme*[2], each epoch is assigned a unique epoch number. The epoch number is stored in an n-bit register in each processor, called the *epoch counter* ($R_{counter}$). It is incremented at the end of each epoch by every processor individually. Every data item in a cache is associated with an n-bit time-tag that records the epoch number when the cache copy is created.

In addition to normal read and write operations, a new memory operation, called Time-Read, is used exclusively for a potentially stale reference marked by the compiler. It is similar to the normal read except that it is augmented with an *offset*. The *offset* indicates the number of epoch boundaries between the current epoch and the epoch in which the data was last updated. At runtime, Time-Read determines whether the cached data is stale or not by comparing the runtime timing information (timetag) of each cache copy with the timing information (offset) provided by the compiler. A Time-Read returns a cache hit if (1) timetag $\geq$ ($R_{counter}$ - offset), which checks whether the cache copy is up-to-date, (2) the address tag matches, and (3) the valid bit is set.

Multi-word cache lines Figure 3 shows the format of a 2-word cache line. The valid bit is the same as in a conventional cache. The timetag field is updated with the current value ($R_{counter}$) of the epoch counter by each memory reference. On a cache miss, the timetags of other words in the same cache line are assigned the value ($R_{counter}$ - 1) to handle the implicit RAW (read-after-write) or WAR (write-after-read) dependences between concurrent tasks in the same epoch [9] (see section 2.1). Since the timetags are updated from the epoch counter by hardware at runtime, there is no need to store the timetags in main memory. Hence, the hardware overhead in the *TPI* scheme is proportional to the size of cache memory.

2.3 Program analysis and reference marking

For correct coherence enforcement, the compiler needs to generate appropriate memory and cache management operations. First, the compiler must generate an epoch counter increment operation at the end of each epoch. Second, potentially stale read references need to be identified and issued as Time-Read operations. For such references, the compiler also needs to calculate their offset values.

Program analysis techniques We use the following analysis techniques in our compiler.

- **Stale reference detection** identifies all the data references in a source program that may violate cache coherence at runtime according to the *stale reference sequence* by using a modified def-use chain analysis.
- **Array data-flow analysis** For a more precise analysis for arrays, we identify *the region* of an array that is referenced by an array reference and treat it as a distinct variable.
- **Gated single assignment (GSA) form** To perform effective array flow analysis, the symbolic manipulation of expressions is preferred because the computation of array regions often involves equality and comparison tests between symbolic expressions. We use demand-driven symbolic analysis using the GSA form [2].
- **Interprocedural analysis** Previous HSCD schemes invalidate the entire cache at procedure boundaries to avoid side effects caused by procedure calls. We use a complete interprocedural analysis to avoid such invalidations and to exploit locality across procedure boundaries.

Revisiting the program example Figure 1(b) shows the memory operations marked by a compiler for the program example in Figure 1(a). Time-Reads with offsets 1 and 2 are issued respectively for the read references to Y in epoch 2 and the first read reference to Y in epoch 3 since the variable is written in epoch 1. Note that the second read reference to Y in epoch 3 is not marked as Time-Read because only *the first read reference to a data item in an epoch* can be a potentially stale data reference.[3] After the staleness of the data is resolved by the first reference, all the subsequent references to the same data will access an up-to-date copy in the cache.

The read reference to X(i) in epoch 4 is also not marked as a Time-Read because the region accessed by the write to the same array in epoch 2 is disjoint. Such unnecessary Time-Reads can be avoided by using array data-flow analysis. On the other hand, the read reference to X(f(i)) should be issued as a Time-Read due to the unknown symbolic index value.

Without interprocedural analysis, cache invalidations would be needed before and after each procedure call to avoid side effects. We eliminate such invalidations by using interprocedural analysis. Time-reads with offset 2 and 3 are generated for the last two read references to the variables W and X, assuming they are modified 2 and 3 epochs before the procedure Q returns. The details of the compiler algorithms are beyond the scope of this paper and are described in [8].

Compiler implementation All the compiler algorithms explained earlier have been implemented on the Polaris parallelizing compiler [17]. Figure 2 shows the flowchart of our reference marking algorithm. First, we construct a procedure call graph. Then, based on the bottom-up scan of the call graph, we analyze each procedure and propagate its side effects to its callers.

The per-procedure analysis transforms the source program into a GSA form [2]. Then, we construct a modified flow graph, called the *epoch flow graph* [9] that contains the epoch boundary information and the control flow of the program. Given a source program unit and its epoch flow graph, G, we identify the target references in each epoch to utilize both spatial and temporal reuses inside a task. After the target reference detection, regions of an array that are referenced in each epoch are computed. A stale reference detection algorithm then determines the existence of the stale memory reference sequences. If a stale reference sequence is found for a target reference, the reference is marked as potentially stale.

[2] It is called two-phase invalidation due to its hardware reset mechanism on epoch counter overflow. [9]

[3] These first read references are identified by finding the *first occurrence of upwardly-exposed uses in an epoch* in our algorithm.

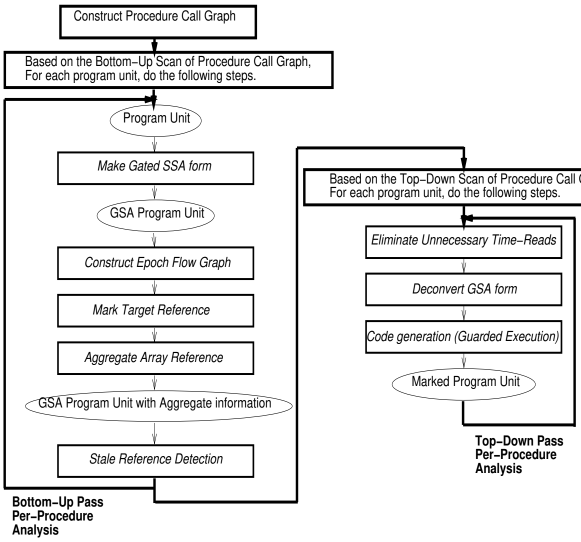

Figure 2: Flowchart for the reference marking.

After the bottom-up side-effect analysis, we perform a top-down scan of the call graph and propagate side effects from callers to callees, thereby eliminating some unnecessary Time-Reads. We also calculate more precise offsets. We then transform the program in the GSA form back to the original program with the reference marking information, and generate appropriate cache and memory operations. The guarded execution technique can be used to further optimize the code generation [8]. The compiler marking algorithms developed here are general enough to be applicable to other compiler-directed coherence schemes [5, 6].

3 Hardware Implementation Issues

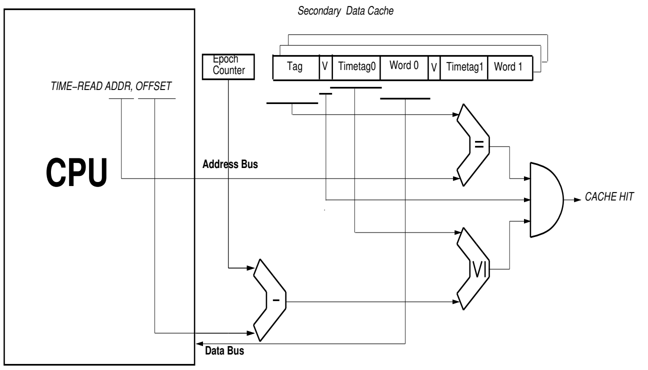

Figure 3: Off-chip implementation for *TPI* Scheme.

Off-chip secondary cache implementation Since most multiprocessors today use off-the-shelf microprocessors, it is more cost effective if the proposed *TPI* scheme can directly be implemented using existing microprocessors. Figure 3 shows a possible off-chip implementation of the *TPI* scheme and the logic circuits necessary to determine a cache hit for a Time-Read operation. In the implementation, the timetag hardware and the epoch counter are placed in the external second-level cache. Note that the timetag needs to be updated only on a read miss and on a write. The following shows the necessary steps required for various on-chip primary cache operations.

- **On a read hit**, the timetag already contains the correct timing information since the cache copy should be up-to-date. Thus, a read hit can be treated as a conventional cache hit.
- **On a read miss**, the timetags of the cache line can be updated off-chip in parallel with the cache line refill since the address is visible off-chip on a cache miss.
- **On a write**, the timetag update can also be made off-chip since, for a write-through on-chip cache, the data address is visible off-chip. Hence, the timetag can be updated in parallel with the memory update. For a write-back on-chip cache, the timetags are updated in parallel with the memory updates when the data is written back at synchronization points, such as at an epoch boundary.
- **On a Time-Read**, the operation is treated as an on-chip cache miss, and a read miss is signaled off-chip. The staleness of data is checked by looking up the off-chip timetag associated with the secondary cache. A cache hit occurs if the condition [$timetag \geq (R_{counter}$ - *offset)*] holds. The fetched data from the off-chip secondary cache then is forwarded to the CPU. Otherwise, a new cache line is fetched from the main memory. This operation requires the offset value to be visible off-chip. The data lines can be used for this purpose since they are not used by a read operation. Similar to a write operation, the Time-Read updates the timetag.

Note that even though the off-chip implementation will incur overhead for Time-Read operations, it will not cause any additional delay for normal reads or writes. The off-chip implementation can also be incorporated into microprocessors that do not have on-chip caches, such as the HP PA-RISC architecture.

Instruction support for Time-Read To avoid the need for a special Time-Read instruction that is not provided in existing instruction set architecture, we can use the following code sequence to implement the Time-Read for the off-chip implementation.

- **store** *offset, special_location* A memory-mapped store operation to write the offset off-chip. This special memory location triggers the Time-Read operation, and the off-chip hardware should interpret the data value as the offset for the following load operation.
- **load** *addr* The load operation for the Time-Read. Together with the previous memory-mapped store operation, this load operation should trigger a cache miss on-chip and the data is fetched from the off-chip secondary cache after the timetag checking is done. Either a bypass cache operation (if supported) or an invalidate followed by regular read operation can be used to force the cache miss intentionally.

Access Size Issues: Byte to Double Word Addressability To this point, we have assumed that memory is accessed in word units and that a timetag is associated with each cache word. However, most microprocessors provide memory operations in various access units, such as bytes, half-words,

words, or double-words. If the data unit with which the timetag is associated does not match the data access unit, a false-sharing effect will occur.[4] For example, with one timetag per cache word, if a byte-store operation modifies a byte in the word, then we have a partial word update. Updating the timetag for the word can lead to a coherence violation because the timing information is no longer accurate for the other bytes of the word. In general, if the unit of data access is smaller than the unit of data with which the timetag is associated, we have a *partial unit access*, and the timetag should not be updated. This can lead to a conservative stale reference detection at runtime since the timetag is no longer updated precisely for a partial word access. A similar situation occurs for a word access if only one timetag is maintained for each multi-word cache line.

This false-sharing issue is common to all cache coherence schemes including hardware directory schemes. Maintaining sharing information per cache word in directory schemes will increase directory storage significantly since the memory requirement is proportional to the total memory size instead of the total cache size. Optimizations to reduce the storage overhead may result in very complicated hardware protocols [12]. However, more fine-grained sharing information can be incorporated in this HSCD scheme more easily because the coherence enforcement is done locally.

A cache memory has two components: data storage and address tag storage. Timetags only increase the data storage and not the tag storage, since the timetags are accessed in the same way as the data. Assuming an 8-bit timetag[5] with four 64-bit word cache lines, the extra storage overhead for the timetags will increase the cache data storage overhead by 12.5%. Since the address tag overhead is much more expensive than the data storage,[6] the extra hardware overhead for the *TPI* scheme is reasonably small.

Write-back vs. write-through policy and write buffer design Due to memory consistency constraints, all global writes need to be committed to the main memory at synchronization points, such as epoch boundaries. Note that this is true for all the shared-memory multiprocessors. This can be done by using write-through caches. However, this will produce more redundant write traffic. By organizing the write buffer as a cache, as in the DEC Alpha 21164 processor [10], such redundant write traffic can be effectively reduced [4]. Note that ordinary write buffers can help hide latencies but cannot eliminate redundant write traffic.

For a write-back cache configuration, the *TPI* scheme should force all global writes to be written back to the main memory at synchronization points. Therefore, the dirty cache copies cannot be kept in the cache across synchronization points. In the hardware directory schemes with an ownership-based protocol, the memory writes can be delayed beyond the synchronization points since the writes can be performed locally if the cached copy is in the exclusive state. The directory can forward the data to the requesting processor from the processor that owns the up-to-date copy even though the memory has not yet been updated.

Cache and system organization	
CPU	single-issue processor
cache size	on-chip 64 KB, direct-mapped
on epoch counter overflow	two-phase reset mechanism
line size	4 32-bit word cache line
timetag size	8-bits
number of processors	16

Cache and system organization	
ALU operations	1 CPU cycle
cache hit	1 CPU cycle
base miss latency	100 CPU cycles
network delay	analytic model [13]
memory bus cycle	3 CPU cycles
off-chip timetag access	2 CPU cycles
off-chip timetag access penalty	8 (3 + 2 + 3) CPU cycles

Table 1: Parameters used for typical simulations.

This difference will create less write traffic in the hardware directory schemes than in the *TPI* scheme.

4 Performance Evaluation

4.1 Experimental Methodology

In this section, we describe our simulation study to evaluate our proposed scheme. We use six programs from the Perfect Club benchmark suite as our target benchmarks. They are first parallelized by the Polaris compiler. In the parallelized code, the parallelism is expressed in terms of DOALL loops. We then mark the Time-Read operations in the parallelized source codes using our compiler algorithms [8], which are also implemented in the Polaris compiler. After the compiler marking, we instrument the benchmarks to generate memory events, which are then used for the simulation.

Simulation Execution-driven simulations [18] are used to verify the compiler algorithms and to evaluate the performance of our proposed coherence scheme. All the simulations assume a 16-processor, physically distributed shared-memory multiprocessor. Each processor has an on-chip 64-KB direct-mapped lock-up free data cache with a 4-word cache line and an infinite size write buffer.[7] The default values of the parameters used for typical simulations are given in Table 1. A write-through write-allocate policy is used for both the *TPI* and the *SC* schemes, while a write-back cache is used for the hardware directory protocol. These write policies are chosen to deliver the best performance for each type of coherence scheme [4]. A weak consistency model is used for all the coherence schemes.

It is assumed that each processor can handle basic arithmetic and logical operations in one cycle and has synchronization operations to support parallel language constructs. The memory system provides a one-cycle cache hit latency and a 100-cycle miss latency assuming no network load. The network delays are simulated using an analytical delay model for indirect multistage networks [13].

[4] Note that the stale data reference marking and offset computation can be performed precisely at compile time regardless of the size of access units since each access unit (such as character, integer, or floating point data) is a distinct variable analyzed by the compiler.

[5] Our experimental results in section 4 show that a 4-bit or 8-bit timetag is large enough to achieve very good performance.

[6] As noted by [16, 22], the tag overhead in the design of the on-chip caches are significant, occupying area comparable to the data portion, especially for set-associative caches with 64-bit addresses.

[7] We use ordinary write buffers for simulations. Therefore, redundant writes are not merged. The assumption of the infinite write buffer will decrease the CPU stall times during the simulation compared to a fixed size write buffer. Chen [4] studied the issue of write buffer design for the compiler-directed schemes and found that 8 words of write merging write buffers will reduce the traffic significantly, and that the write through with the write merging write buffer is a better choice than a write back cache implementation for compiler-directed schemes.

Program	Size (lines)	Total Refs.	% Shared	% Private	% Time-Read (Potential Stale Access)	Number of epochs created
SPEC77	5346	$9.0*10^6$	55.3%	44.7%	39.0%	11074
OCEAN	3101	$9.2*10^6$	34.9%	65.1%	2.1%	1481
TRFD	572	$9.7*10^6$	61.1%	38.9%	29.0%	5299
FLO52	3182	$7.1*10^6$	39.4%	60.6%	12.2%	1222
MDG	1523	$4.9*10^6$	36.5%	63.5%	5.94%	80016
QCD	3505	$6.7*10^6$	18.3%	81.7%	1.96%	45687

Table 2: Dynamic Memory reference statistics for the Perfect Club benchmarks.

The execution-driven simulator instruments the application codes to generate events that reflect the behavior of the codes executing on the target architecture. Simulated events include global and local memory accesses, parallel loop setup and scheduling operations, and synchronization operations. Static cyclic scheduling is used for distributing loop iterations to the processors. The memory events are generated from a source code-level instrumentation. Thus, we do not consider back-end compiler code generation and optimization in our simulations. These back-end compiler issues mostly affect only private references.[8] Since we are interested in comparing different coherence schemes for shared data references, these are not considered in our simulations.

Dynamic reference statistics Table 2 shows the dynamic reference counts of the benchmarks. Due to the automatic array privatization performed by Polaris, the percentage of shared memory references are modest, ranging from 18.3% (QCD) to 61.1% (TRFD). We use a fully-inlined version of the TRFD code to avoid aliasing due to parameter passing. The number of Time-Read operations shows the potentially stale data references detected through our interprocedural analysis. Depending on the application, the range of these stale data references varies substantially, ranging from 1.96% (QCD) to 39.0% (SPEC77). The figure also shows the number of epochs created at runtime for each benchmark, which can be used to estimate the task grain size for each benchmark. OCEAN has the largest grain size and MDG has the smallest.

4.2 Impact of Hardware Support

In this section, we investigate how the cache coherence support impacts the overall system performance. Four different hardware and compiler-directed coherence schemes are studied. They differ in the caching strategy for shared memory accesses, and in the amount of hardware and compiler support required. All schemes assume the same underlying multiprocessor building blocks, i.e. the same processor, memory, and network modules.

Cache coherence schemes In addition to the *TPI* scheme, we simulate the software-bypass (*SC*) scheme introduced in section 2 to study the performance of our compiler algorithms without hardware support. We also study the following two schemes for comparison.

[8] Code optimizations usually decrease the frequency of private references through register allocation and other optimizations. On the other hand, code generation, such as address calculation, will increase the frequency of private references due to spill code. Shared data consistency prevents optimization of the references to the shared data [19].

1. **Base Scheme** (*BASE*) The scheme does not cache shared data to avoid the cache coherence problem. All shared memory references are treated as cache misses. They will access main memory directly. This coherence mechanism is similar to that of the Cray T3D, where the compiler generates noncacheable loads for shared memory references and cacheable loads for private references.

2. **Full-Map Directory Scheme** (*HW*) This scheme uses a simple, three-state (*invalid, read-shared, write-exclusive*) invalidation-based protocol with a full-map directory [3] but without broadcasting. It gives a performance comparison to the hardware directory protocols.

Table 3 summarizes the characteristics of the four schemes according to the hardware and compiler support required, the caching strategy for shared data, and their performance limitations.

Miss rates Figure 4 shows the miss rates of each scheme for the six benchmarks with a 64KB direct-mapped cache and 4-word cache lines. The impact of different cache line sizes are discussed in section 4.3. An interprocedural analysis is performed for both the *TPI* and *SC* schemes.

The miss rates are classified into *sharing misses* and *nonsharing misses*. A sharing miss occurs when the address tag matches but the cached data has been either invalidated or a timetag mismatch occurs. All other misses are nonsharing misses. The sharing misses are further classified into true and false sharing misses. The *true sharing misses* are necessary misses to satisfy cache coherence, while the false sharing misses are the unnecessary misses due to the lack of compile-time information or the false-sharing effect of the hardware protocols. The *false sharing misses* are identified during simulations using the method in [20]. If an invalidation is caused by an access to a word that the local processor had not used since getting the block into its cache, then it is a false sharing invalidation. Any subsequent invalidation miss on this block will also be counted as a false sharing miss.

With the compile-time analysis alone (*SC*), a significant number of unnecessary misses in the base scheme are eliminated, decreasing the miss rates up to 29.8% (TRFD). This demonstrates how much compile-time program analysis can improve cache utilization by caching remote shared data. First, read-only shared variables can be cached safely. Second, writes to shared variables can always utilize cached data since only a read can access a stale cache copy. Third, some of the redundant memory accesses in a task can safely access the cached data by analyzing temporal and spatial reuses except for the first one. In particular, *SC* eliminates most of the sharing misses for OCEAN and QCD. However,

Coherence Scheme	Hardware support	Software support	Sharing information	Caching for shared data access	Limitations
BASE	No	No	No	no caching for shared data	exclusively remote memory access
SC	No	stale reference detection	static local/global	writes partial reuses within a task	no inter-epoch locality
TPI	timetag storage	stale reference detection/ offset computation	static local/global dynamic local	caching for read-write variables	branches unknown symbolic constants
HW	directory storage cache/directory controller design	optional	dynamic local/global	caching for read-write variables	false sharing coherence traffic increased miss penalty

Table 3: Characteristics of different hardware and software coherence schemes.

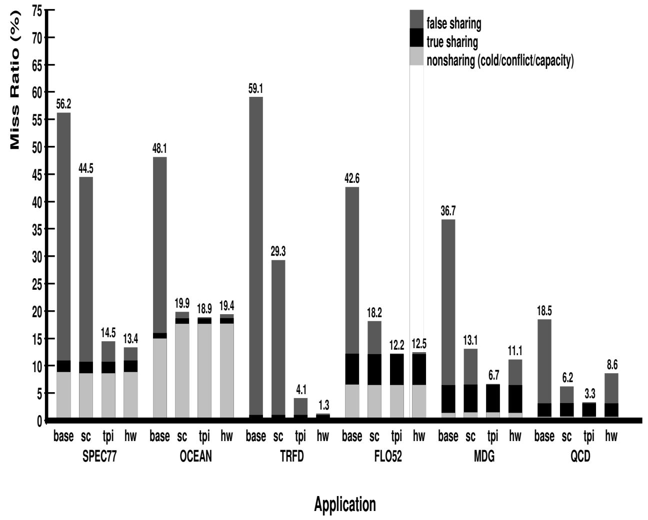

Figure 4: Distribution of the misses.

for the other benchmarks there are still a significant number of false sharing misses due to the conservative compiler decision on branches, memory disambiguation, and scheduling. In addition, no locality is preserved across epoch boundaries for all shared read references.

In *TPI*, the timetag provides timing information for runtime checking which enhances the cache-ability of shared data. First, *TPI* can determine at runtime whether the most recent cache update and the current Time-Read is performed on the same processor. It allows locality to be exploited across task boundaries for read-write shared variables. Second, the intra-task locality is fully captured in *TPI*. *SC* cannot preserve full intra-task locality because of its conservative analysis of temporal and spatial reuses. The additional reduction in miss rates by *TPI* over *SC* ranges from 6.4% (MDG) to 30.0% (SPEC77). Interestingly, *TPI* shows comparable miss rates to the hardware directory scheme (*HW*) for all the benchmarks tested. For SPEC77 and TRFD, *TPI* increases the miss rates slightly compared to *HW*. On the other hand, for FLO52, MDG, and QCD, *TPI* decreases the miss rates as much as 4.4% (MDG) compared to *HW*.

The increase in miss rates is due to the conservative compiler analysis. On the other hand, the decrease in miss rates is due to the elimination of the false-sharing effect of multi-word cache lines in the directory protocol. On a write to a shared memory location, the directory protocol invalidates all the cache lines that are shared by other processors due to the invalidation traffic. In *TPI*, there is no invalidation traffic. In addition, only affected cache words are invalidated in *TPI* rather than the entire cache line.

Network traffic and miss penalty One criticism against compiler-directed schemes has been the increased write traffic caused by their write-through policy.[9] Figure 5 classifies the network traffic into write traffic, read traffic, and coherence traffic.

The read traffic is directly proportional to the miss rates, and *TPI* and *HW* incur comparable read traffic. The write traffic reflects the write-policy used. Since both *TPI* and *SC* use write-through, they incur approximately the same amount of write traffic. Compared to the *HW* scheme that uses a write-back policy, the additional increase in the write traffic in both compiler-directed schemes is small, except for TRFD. In TRFD, there is a significant number of redundant writes which increases the overall network traffic of *TPI* substantially compared to *HW*. This additional write traffic can be effectively eliminated by organizing a write buffer as a cache [4]. Similar technique can also be employed to remove redundant write traffic for update-based coherence protocols. The third type of network traffic is for coherence transactions in the directory protocol. This extra traffic is relatively small compared to the read and write traffic for the benchmarks considered.

A more interesting issue regarding the network traffic is its impact of on the miss penalty. Usually, higher network loads imply a higher miss penalty. Hardware directory schemes require a higher miss penalty due to coherence-related transactions. Since we use a weak consistency model with an infinite write buffer, write latency can be completely hidden by buffering. On a read miss to a data item that is exclusively-owned by another processor, however, the requesting processor should block until the owner processor supplies the data, which requires several network transactions.[10] This problem would be much more significant with a sequential consistency model since both reads and writes are affected by the coherence transactions.

Table 4 shows the average miss penalty for both *TPI* and *HW*. For SPEC77, OCEAN, and FLO52, both schemes show a comparable penalty because the shared data is usually in a read-shared mode. On the other hand, for MDG, QCD, and TRFD, *HW* has a much higher miss penalty (up to 31.2%

[9] Although the compiler-directed schemes can employ write-back at task boundaries, it increases the latency of the invalidation, and results in more bursty traffic [4].

[10] This is the network latency vs. traffic tradeoff. Using write-through cache will increase the traffic while the write back cache will will increase the latency for hardware directory schemes.

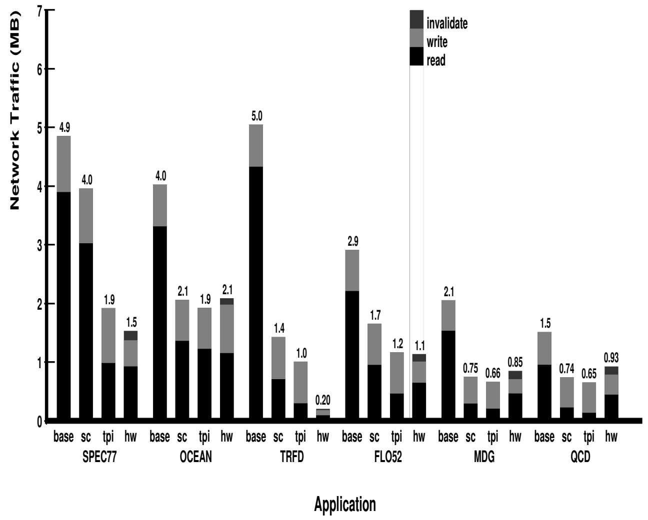

Figure 5: Distribution of the network traffic.

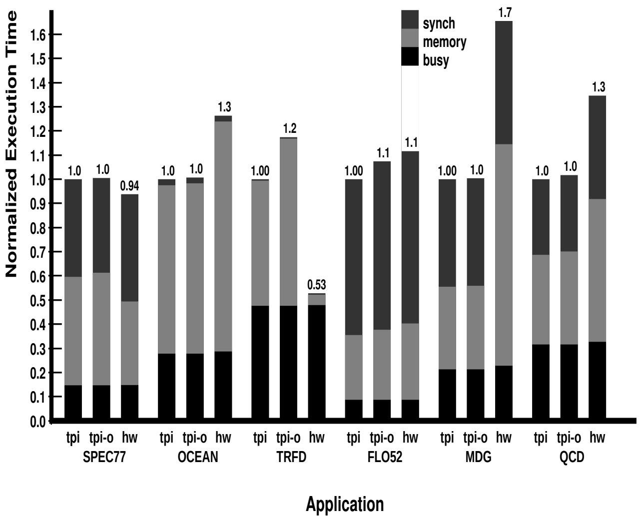

Figure 6: Distribution of the execution time normalized to the *TPI* scheme.

Program	Average Miss Penalty			
	Two-Phase Invalidation		Hardware Directory	
	16 bytes	64 bytes	16 bytes	64 bytes
SPEC77	136.2	356.3	136.4	355.5
OCEAN	136.2	354.3	136.4	353.6
FLO52	136.2	355.1	136.6	361.2
MDG	136.0	353.4	178.7	420.1
QCD	136.0	354.7	145.5	405.4
TRFD	136.0	352.4	149.1	418.6

Table 4: Average miss penalty of *TPI* and *HW*.

for MDG) due to the migratory sharing patterns in these benchmarks. Note that for TRFD, even though *TPI* shows substantially higher write traffic than *HW*, the impact on the miss penalty from the write traffic is negligible since the overall traffic for the application is still low and writes can be buffered. However, the migratory sharing pattern of the application increases the miss penalty of *HW* by 9.63% compared to *TPI*.

Execution time Figure 6 displays the distribution of execution times normalized to the *TPI* scheme. The execution time consists of busy cycles (CPU time), memory cycles (waiting time on memory accesses), and synchronization cycles (waiting time due to synchronizations) that are averaged over 16 processors. For a more focused study, we show only the performance of *TPI* and *HW* in the remaining part of the paper since the performance improvement by caching shared data in both *TPI* and *HW* shows a speed up of at least 3 to 4 times over the *BASE* scheme from our simulations. In the figure, we also show the execution time of the off-chip implementation of the *TPI* scheme. This off-chip implementation pays an extra 8 cycles of penalty (3 CPU cycles of address bus + 2 CPU cycles of off-chip timetag access + 3 CPU cycles of data bus) for Time-Read operations. To avoid the side effects caused by secondary cache implementation (see section 3), we assume off-chip timetag hardware without secondary caches.

The memory stall times constitute the largest portion of the execution time in most of these benchmarks. In addition, except for OCEAN and TRFD, processors spend a significant amount of time on synchronization. For OCEAN, MDG, and QCD, *TPI* improves performance from 30.0% (FLO52) up to 70.0% (MDG). For MDG, in particular, *TPI* can eliminate more than half of the memory stall times in *HW* due to both the small miss penalty (136 cycles instead of 178.7 cycles, see Table 4) and the smaller miss rates (6.7% instead of 11.1%, see Figure 4) of *TPI*. However, for TRFD, *TPI* shows an execution time that is longer than that of *HW* by 88.7%. This increase is primarily due to its higher miss rates (4.1% instead of 1.3%), which are caused by conservative reference marking. Overall, *TPI* shows a performance that is competitive to that of *HW*, which suggests that the compiler-directed coherence scheme can be a viable alternative for large-scale loop-parallel scientific codes like the ones used in our experimentation.

Surprisingly, the off-chip *TPI* scheme performs as well as the integrated on-chip *TPI* scheme for benchmarks SPEC77, OCEAN, MDG, and QCD. This is because with the non-blocking caches we used, the extra off-chip access penalty can be hidden most of the time for these benchmarks. For benchmarks TRFD and FLO52, the off-chip *TPI* scheme increases the execution time up to 20%. Overall, the performance of the off-chip implementation is comparable to both the on-chip *TPI* scheme and the *HW* scheme. This suggests the off-chip *TPI* scheme using the off-chip microprocessors would be a good practical solution.

4.3 Impact of cache line size

The choice of the cache line size can have a critical impact on cache performance. Figure 7 shows the cache miss rates for both the *TPI* and the *HW* using small (16 bytes) and large (64 bytes) cache lines. For benchmarks SPEC77, OCEAN, TRFD and FLO52, large cache lines are effective in reducing cache misses. By an implicit prefetching effect of a large cache line, a significant portion of nonsharing misses, such as cold start misses, are removed. Note that a large cache line reduces the amount of sharing, decreasing the true sharing misses while increasing nonsharing misses. The MDG and QCD benchmarks show similar trends. This is due to increased conflict misses resulting from a smaller number of cache lines available. Overall, we can see that the spatial

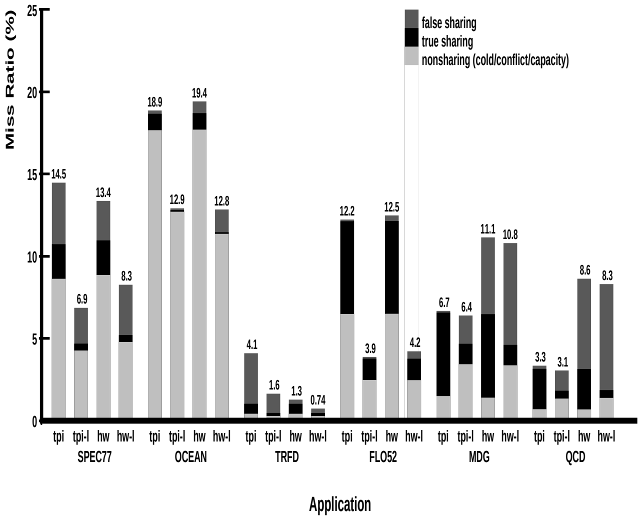

Figure 7: Miss rates for small (16 bytes) and large (64 bytes) cache line sizes (represented by tpi-l and hw-l for *TPI* and *HW* schemes, respectively).

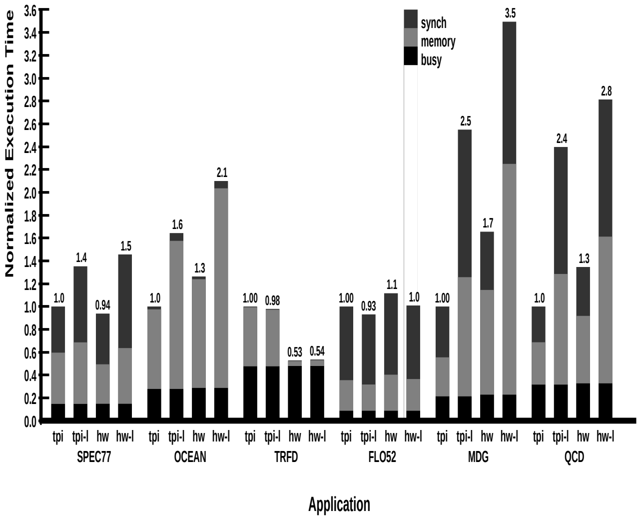

Figure 8: Normalized execution time for small (16 bytes) and large (64 bytes) cache line sizes.

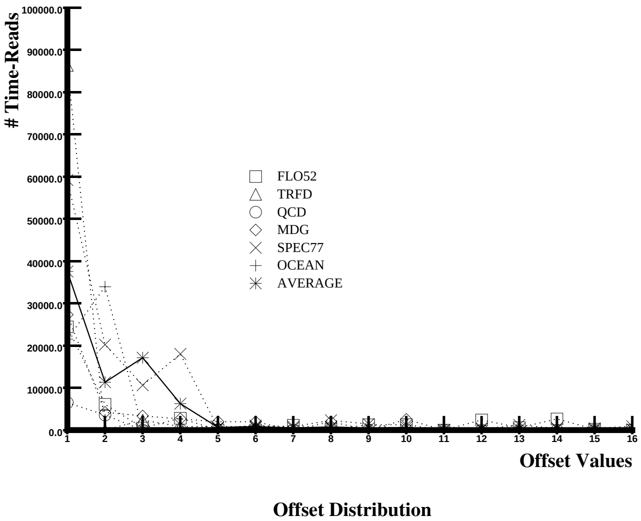

Figure 9: Distribution of offsets for Time-Read.

locality exploited by a large cache line reduces cache misses for most benchmarks.

The next question is on the negative side effect of large cache lines, i.e., increased miss penalty and network loads. Table 4 in section 4.2 shows the average miss penalty for both 16 byte and 64 byte cache lines. The figure shows that large cache lines increase the miss penalty substantially (more than a factor of two) for all the benchmarks. This has a significant impact on the performance as we can see in Figure 8, which shows the execution times with both small and large cache lines normalized to the execution time of the *TPI* with 16 byte lines. Except for TRFD and FLO52, the execution times of most benchmarks increase with 64 byte lines. This demonstrates that the negative impact (increased network load and miss penalty) of large cache lines is more significant for these benchmarks than the positive impact (higher cache hit rates). This increased miss penalty results in both longer memory stall times and longer synchronization delays in spite of higher cache hit rates.

4.4 Impact of timetag size on inter-epoch locality

The size of the timetag determines how far *TPI* can exploit locality across epoch boundaries. The inter-epoch locality can be identified by studying the offset values of the Time-Read operations. Figure 9 shows the distribution of offsets for the six benchmarks. Most offsets are small, implying the distance between the inter-epoch reuses is small. In fact, more than 90% of offsets are within a distance of 8 epochs. The largest offset value is 37 that occurred in SPEC77. This implies a timetag size as small as 3 or 4-bits might be sufficient to capture most of the inter-epoch locality. Besides, the larger the offset, the more likely it is to be replaced before it is used in later epochs.

Since we use a timetag of a fixed size, the epoch counter will eventually overflow when it reaches the maximum value. Therefore, there must be a strategy to recycle the timetag values for a correct operation. A simple strategy is to invalidate the entire cache and reset the epoch counter when it overflows. However, a small timetag may lead to frequent invalidations. In [9], we propose a *two-phase hardware reset mechanism* that can simultaneously invalidate only those cache data whose timetags are out of phase. The idea is to use the most significant bit (MSB) of the epoch counter as an overflow flag. The overflow condition is detected whenever the MSB of $R_{counter}$ changes either from 1 to 0 or from 0 to 1. This implies the largest epoch number is $2^{n-1} - 1$ when using an n-bit $R_{counter}$. On this event, all the cache entries whose MSBs of their time-tags are the same as the new MSB of the current $R_{counter}$ are out of phase and should be invalidated.

To compare the effectiveness of the proposed reset mechanism, we simulate both the two-phase reset mechanism (TPR) and the entire cache invalidation (INV) on the epoch counter overflow. Figure 10 shows the miss rates of the *TPI* scheme for various timetag sizes using the two reset mechanisms. As we can see, with an 8-bit timetag, most of the locality can be captured for all the benchmarks regardless of the reset mechanism chosen. However, with smaller 2-bit or 4-bit timetags, the two-phase reset mechanism can avoid

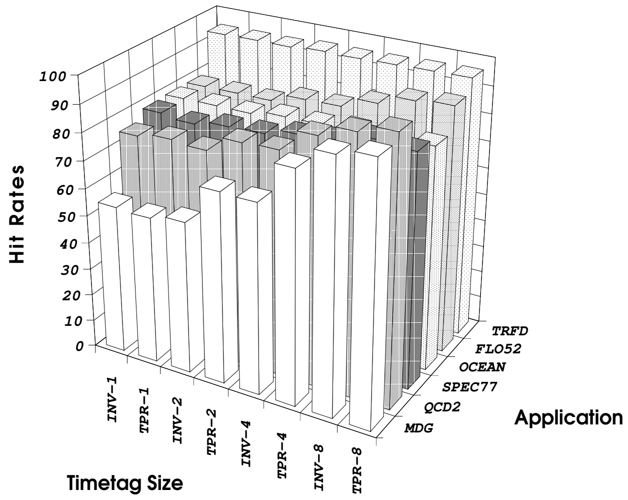

Figure 10: Impact of timetag size and reset mechanism on epoch counter overflow.

```
      DOACROSS  i = 1, n
S1:   A(i-1) =         /* Update T-tag */
S2:          = A(i)    /* Time-Read, 0, No T-tag update */
S3:   A(i+1) =         /* No T-tag update */
S4:          = A(i)
      ENDDO
```

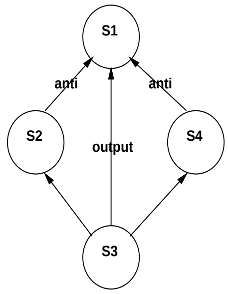

Figure 11: Example of compiler marking in a doacross loop.

most of unnecessary invalidations compared the entire cache invalidation. With a 1-bit timetag, both reset mechanisms do little since much of inter-epoch locality distance is larger than 2 epochs. The impact of timetag size also depends on the benchmarks characteristics. For example, the performance impact is more significant in benchmarks MDG, QCD and FLO52 than the other benchmarks. Generally, we found that the performance impact of timetag size is larger than that of the reset mechanism chosen. And, relatively small timetags, such as 4-bit or 8-bit, are enough to capture most of the locality for the benchmarks considered.

5 Discussion

We will now address some language and operating system issues which arise in the implementation of the *TPI* scheme.

Handling DOACROSS For parallel threads in an epoch which requires synchronizations due to data dependences, there will be reads and writes to the shared data in the same epoch from different threads. One important example is the DOACROSS loop. In a DOACROSS loop, there are cross-iteration data dependences. If each iteration is scheduled on a distinct processor, proper synchronization is required to enforce the data access order. For a Write-after-Read dependence (an anti-dependence), the write and the read operation are performed in different processors in the same epoch. In this case, the processor that performs the write will make the cache copy brought in by the processor that performed the read become stale. A similar situation occurs for a Write-after-Write dependence (an output-dependence). A Read-after-Write dependence (a flow dependence) will not have such a problem because the source of the dependence is a write, and the sink of the dependence will read the updated copy.

Consider the example in Figure 11. Assume that each iteration of the DOACROSS loop is scheduled on a distinct processor. Using our compiler algorithm, since statement S2 contains the first read reference to variable A, thus the read access is marked as a Time-Read operation. Since S2 reads the value produced by statement S3 in the previous iteration in the same epoch, the offset for the Time-Read should be zero. According to the timetag update strategy used in DOALL loops, both write references in S1 and S3, and the read reference in S2, would update the timetag field using the same epoch counter value. This will result in stale copies with an up-to-date timetag. For example, the value written by S3 in the first iteration should be overwritten by S1 in the third iteration because they reference the same array element, A(2). However, both copies of A(2) will have the same timetag value because they occur in the same epoch. As a result, a read access to A(2) in later epochs in the first processor cannot detect A(2) as a stale copy since it has an up-to-date timetag. A similar situation can occur for the read reference in S2 and the write reference in S1. The read reference of S2 in the first iteration will create a cache copy which has the same timetag as the cache copy produced by S1 in the second iteration.

In summary, the *source* of a cross-iteration anti- or output-dependence can create a stale cache copy which has an up-to-date timetag. To avoid such a problem, if an access is the source of a cross-iteration anti- or output-dependence, the access should not update its timetag even if it is the first access in the epoch. This requires an extra flag, called the *timetag update* flag, to determine whether or not the timetag should be updated. If the flag is set, the associated reference will update the timetag of the accessed cache word with the current value of the epoch counter. Otherwise, the timetag will not be updated. In our example in Figure 11, the Time-Read in S2 and the write access in S3 should not have the timetag update flag set. On the other hand, the write reference in S1 should update the timetag since it is the sink of a dependence.

Handling locks and critical sections We have focused on maintaining cache coherence with parallel constructs, such as DOALL or DOACROSS loops. In this section, we show how the locks and critical sections can be handled with the *TPI* scheme.

A critical section is a region of code where concurrent tasks may access the same data, which produces memory dependences across processors. The critical section is usually protected by lock variables or other synchronization primitives to enforce mutual exclusion. The difference between the critical sections and the synchronizations used in DOACROSS loops is that at compile time, the memory access order is nondeterministic in critical sections while the event order is deterministic in DOACROSS loops. Because of this characteristic, efficient cache management for critical sections is inherently more difficult.

Consider the program example in Figure 12(a). All par-

```
B(j) = ....
DOALL  i = 1, k
   lock (sync1)          /* Begin of Critical Section */
     ....
     A = B(i) ...
       = SUM ...
     SUM = ...  B(i - 1)
     ....
   unlock (sync1)        /* End of Critical Section */
   ....
ENDOALL
DOALL j = 1, m
     ....
       = A + B(j)
```

(a)

```
/* Rcounter ++ */

/* Time-Read B(1),1, T-tag Update */
/* Write A, No T-tag Update */
/* Time-Read SUM, 0, No T-tag Update */
/* Time-Read B(i-1),1, T-tag Update */
/* Write SUM, No T-tag Update */

/* Rcounter ++ */

/* Time-Read A,1, T-tag Update */
/* Time-Read B(j),2, T-tag Update */
```

(b)

Figure 12: Program example with critical section.

ticipating processors can concurrently write to variables A and SUM, forcing local cache copies to become stale at anytime. The simplest solution is to "bypass" the cache within critical sections and access data directly from main memory. A Time-Read operation with an offset of 0 can be used for this purpose, but this simple strategy is obviously too conservative.

For those variables that are not involved in any dependence in an epoch, we still can preserve some locality inside critical sections. For example in Figure 12(a), the read references to B(i) and B(i-1) are safe because array B is read-only in the epoch. Figure 12(b) shows the memory operations generated by the compiler considering the above observation.

The strategy will not preserve locality across critical sections for the variables that are modified inside the critical section. However, the locality across critical sections is likely to be small anyway due to contention among many participating processors.

6 Conclusions

Private caches are critical components in the design of multiprocessor systems. However, maintaining cache coherence for large-scale systems is a complicated issue. Pure-hardware directory schemes can be very effective, but are too expensive for large-scale multiprocessors. Thus, many commercial systems today, such as the Cray T3D and the Intel Paragon, do not use such schemes.

In this paper, we investigated a hardware-supported, compiler-directed (HSCD) cache coherence scheme, called the two-phase invalidation (*TPI*) scheme which relies mostly on compiler analysis and provides a reasonable amount of hardware support. This approach has a long history of predecessors, including C.mmp, IBM's RP3, Illinois Cedar, and several recently proposed schemes.

The *TPI* scheme can be implemented on a large-scale multiprocessor using off-the-shelf microprocessors, and can be adapted to various cache organizations, including multi-word cache lines and byte-addressable architectures. Several system related issues, including coherence enforcement for critical sections, inter-thread communication are also addressed. The cost of the required hardware support is small and proportional to the cache size. The cost is less than the Scalable Coherence Interface (SCI) because the tag size required is much smaller than that required in SCI. The necessary compiler algorithms, including intra- and inter-procedural array data-flow analysis, have been implemented on the Polaris paralleling compiler [17].

The result of our simulation study using the Perfect Club Benchmarks shows that both hardware directory schemes and the *TPI* scheme have comparable amount of unnecessary cache misses. In hardware schemes, these misses result from the false-sharing effect while in our proposed scheme, they come from the conservative assumptions made by the compiler. In addition, the hardware directory scheme incurs a higher miss penalty for applications with many migratory sharing patterns. Our results show that in large-scale loop-parallel scientific applications, such as the benchmarks considered in our experimentation, the performance degradation due to the conservative decisions made by the compiler is comparable to the degradation caused by the false sharing and the increased miss penalty in the hardware directory scheme.

With its comparable performance and reduced hardware costs compared to hardware directory schemes, the proposed *TPI* scheme can be a viable alternative for machines, such as the Cray T3D, that do not have hardware-coherent caches. We also demonstrated that the required compiler analysis is feasible using existing compiler technologies.

Acknowledgments The research described in this paper was supported in part by the NSF Grant No. MIP 89-20891 and MIP 93-07910. We wish to thank several friends and colleagues at the University of Illinois who participate in the review of this paper, especially Vijay Karamcheti for his thoughtful comments. We also thank Prof. Sang Lyul Min at Seoul National University and Prof. David Lilja at University of Minnesota for proofreading the paper and their valuable comments. Special thanks go to Tae Hyung Kim at University of Maryland for his early comments and encouragement. Lastly, we'd like to thank Steve Scott at Cray for his help on the revision of this paper.

References

[1] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon. Comparison of Hardware and Software Cache Coherence Schemes. *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 298–308, May 1991.

[2] R. Ballance, A. Maccabe, and K. Ottenstein. The Program Dependence Web: a Representation Supporting Control- Data- and Demand-Driven Interpretation of Imperative Languages. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271, June 1990.

[3] 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.

[4] Yung-Chin Chen. Cache Design and Performance in a Large-Scale Shared-Memory Multiprocessor System. Technical report, Univ. of Illinois, Dept. of Elec. Eng ., 1993. Ph.D. Thesis.

[5] H. Cheong. Life Span Strategy - A Compiler-Based Approach to Cache Coherence. *Proceedings of the 1992 International Conference on Supercomputing*, July 1992.

[6] H. Cheong and A. Veidenbaum. Compiler-Directed Cache Management In Multiprocessors. *IEEE Computer*, 23(6):39–47, June 1990.

[7] T. Chiueh. A Generational Approach to Software-Controlled Multiprocessor Cache Coherence. *Proceedings of the 1993 International Conference on Parallel Processing*, August 1993.

[8] Lynn Choi. Hardware and Compiler Support for Cache Coherence in Large-Scale Multiprocessors. *Ph.D. thesis, Computer Science Dept., University of Illinois*, Mar. 1996.

[9] Lynn Choi and Pen-Chung Yew. A Compiler-Directed Cache Coherence Scheme with Improved Intertask Locality. *Proceedings of the ACM/IEEE Supercomputing'94*, pages 773–782, Nov. 1994.

[10] Digital Equipment Corp. Alpha 21164 Microprocessor: Hardware Reference Manual. 1994.

[11] E. Darnell and K. Kennedy. Cache Coherence Using Local Knowledge. *Proceedings of the Supercomputing '93*, pages 720–729, November 1993.

[12] C. Dubnicki and T. LeBlanc. Adjustable Block Size Coherent Caches. *Proceedings the 19th Annual International Symposium on Computer Archtecture*, pages 170–180, May 1992.

[13] C. P. Kruskal and M. Snir. The Performance of Multistage Interconnection Networks for Multiprocessors. *IEEE Transactions on Computers*, C-32(12):1091–1098, Sept., 1987.

[14] D. J. Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Computing Surveys*, 25(3):303–338, Sep. 1993.

[15] S. L. Min and J.-L. Baer. Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):25–44, January 1992.

[16] J. M. Mulder, N. T. Quach, and M. J. Flynn. An area model for on-chip memories and its application. *Journal of Solid State Circuits*, 26:98–106, Feb. 1991.

[17] D. A. Padua, R. Eigenmann, J. Hoeflinger, P. Peterson, P. Tu, S. Weatherford, and K. Faign. Polaris: A New-Generation Parallelizing Compiler for MPPs. In *CSRD Rept. No. 1306*. Univ. of Illinois at Urbana-Champaign., June, 1993.

[18] D. K. Poulsen and P.-C. Yew. Execution-Driven Tools for Parallel Simulation of Parallel Architectures and Applications. *Proceedings of the Supercomputing 93*, pages 860–869, Nov. 1993.

[19] J. Torrellas, M. S. Lam, and J. L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, C-43(6):651–663, June 1994.

[20] D. M. Tullsen and S. J. Eggers. Limitations of Cache Prefetching on a Bus-Based Multiprocessors. *Proceedings of The 15th Annual International Symposium on Computer Architecture*, pages 278–288, May 1993.

[21] A. V. Veidenbaum. A Compiler-Assisted Cache Coherence Solution for Multiprocessors. *Proceedings of the 1986 International Conference on Parallel Processing*, pages 1029–1035, August 1986.

[22] H. Wang, T. Sun, and Q. Yang. CAT - Caching Address Tags: A Technique for Reducing Area Cost of On-Chip Caches. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 381–390, June 1995.