Hardware and Compiler-Directed Cache Coherence in Large-Scale Multiprocessors:

Design Considerations and Performance Study[1]

Lynn Choi
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 West Main Street
Urbana, IL 61801
Email: *lchoi@csrd.uiuc.edu*

Pen-Chung Yew
4-192 EE/CS Building
Department of Computer Science
University of Minnesota
200 Union Street, SE
Minneapolis, MN 55455-0159
Email: *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. The scheme 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 minimal and proportional to the cache size. The necessary compiler algorithms, including intra- and interprocedural array data flow analysis, have been implemented on the Polaris parallelizing compiler [33].

From our simulation study using the Perfect Club benchmarks [5], 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. Given its comparable performance and reduced hardware cost, the proposed scheme can be a viable alternative for large-scale multiprocessors such as the Cray T3D, which rely on users to maintain data coherence.

Keywords : Cache Coherence, Memory Systems, Performance Evaluation, Computer Architecture, Shared-Memory Multiprocessors.

[1] A preliminary version of some of this work appears in [17, 18].

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 [24, 20]. They, instead, provide software mechanisms while relying 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 [38], the NYU Ultracomputer [23], the IBM RP3 [6], and the Illinois Cedar [27], 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 [10, 12, 13, 14, 18, 21, 29, 30] have been recently proposed. These schemes give better performance, but demand more hardware and compiler support than the previous schemes. They require a more precise program analysis to maintain coherence on a reference basis [10, 11, 18] 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 [31] compared the performance of a directory scheme and a timestamp-based scheme assuming infinite cache size and single-word cache lines. Lilja [28] compared the performance of the version control scheme [13] 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 directory schemes. Adve [1] used an analytical 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 [9] showed that a simple invalidation scheme can achieve performance comparable to that of a directory scheme and discussed 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 performance of a HSCD scheme. The proposed scheme can be implemented on a large-scale multiprocessor using off-the-shelf microprocessors, such as the Cray T3D, and 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. To study the compiler analysis techniques for the proposed scheme, we have implemented the compiler algorithms on the Polaris parallelizing compiler [33]. By performing execution-driven simulations on the Perfect Club Benchmarks, we evaluate the performance of our scheme compared to a hardware directory scheme.

In Section 2, we describe an overview of our cache coherence scheme and discuss our compiler analysis techniques and their implementation. In Section 3, we discuss the hardware implementation issues and their performance/cost tradeoffs. The issues discussed include the off-chip secondary cache implementation, partial word access, and write buffer designs. In Section 4, we present our experimental methodology and evaluate the performance of our proposed scheme using the Perfect Club Benchmarks. In Section 5, we discuss synchronization issues that involve locks, critical sections, and task scheduling. Finally, we conclude the paper in Section 6.

2 Overview of our cache coherence scheme

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. A set of perfectly-nested parallel loops is considered as a single epoch. Multiple epochs may occur due to intervening code in non-perfectly-nested parallel loops. Figure 1 shows a parallel program and its corresponding epochs.

Stale reference sequence The following sequence of events creates a stale reference at runtime [37]: (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 later at time T_2 ($> T_1$), and

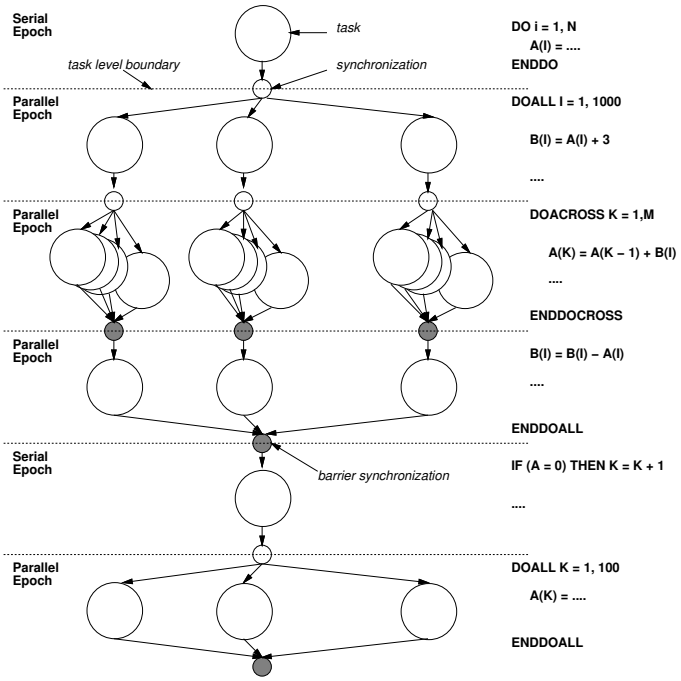

Figure 1: Epochs in a parallel program.

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), 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 2 shows a program example (Figure 2(a)), its corresponding memory events at runtime (Figure 2(c)), and the cache contents of each processor (Figure 2(d)). It assumes two-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 a new copy is created by processor 2 in epoch 2.

However, note that there are two additional stale data references that 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) causes an implicit WAR (write-after-read) 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) a *potentially stale data reference*. This sequence can be detected easily by a compiler using

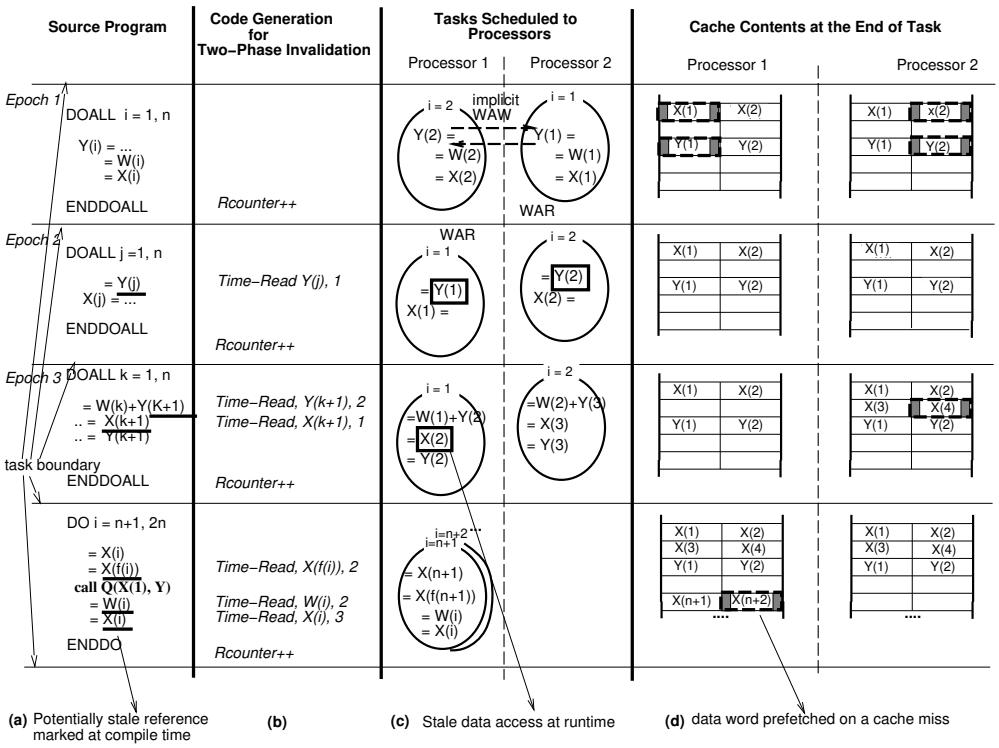

Figure 2: Epochs and stale data references in a parallel program.

a modified def-use chain analysis used in standard data-flow analysis techniques [15, 16].

Software cache-bypass scheme (SC) Once all 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 using a bypass cache operation that accesses the main memory directly.[2] 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 2) will be marked as potentially stale because Y(2) was updated in epoch 1. Due to dynamic runtime scheduling, The scheme cannot determine that the update in epoch 1 and the read in epoch 3 are issued by the same processor. 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 propose a hardware scheme that keeps track of the local cache states at runtime.

[2]This operation can be implemented in the MIPS R10000 processor with a cache block invalidate (*Index Write Back Invalidate*) followed by a regular load operation [25]. Similarly, the Flush instruction ($DCBF$) can be used for the IBM PowerPC 600 series microprocessor [7].

2.2 Two-phase invalidation scheme (*TPI*)

In our proposed HSCD scheme, called the *Two-Phase Invalidation (TPI) scheme*[3], 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* ($R_{counter}$), and is incremented at the end of each epoch by every processor individually. Every data item in a cache is associated with an n-bit timetag that records the epoch number when the cache copy is created.[4]

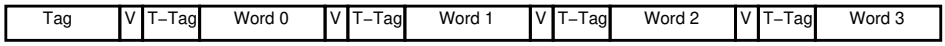

Figure 3: Fields in a 4-word cache line, where T-Tag denotes a timetag.

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 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) the timetag $\geq$ ($R_{counter}$ - offset[5]), which checks whether the cache copy is up-to-date,
- (2) the address tag matches, and
- (3) the valid bit is set.

In addition to memory operations, every read or write instruction is responsible for the update of the timetag. On a read hit including Time-Read, the timetag remains the same since the current timetag is already up-to-date. Therefore, there is no extra delay for the update. On a cache miss or on a write, the timetag of the addressed cache word needs to be updated with the current value of the epoch counter. This update can be performed in parallel with the cache line refill or with the write operation.

Multi-word cache lines Figure 3 shows the format of a 4-word cache line. The valid bit is the same as in a conventional cache. The timetag field is updated by each memory reference with the current value ($R_{counter}$) of the epoch counter. On a cache miss, the timetags of other words in the same cache line are assigned the value ($R_{counter}$ - 1). This is because a multi-word line can cause implicit RAW (read-after-write) or WAR (write-after-read) dependences between two tasks in the same epoch, and may thereby create a stale access in the subsequent epochs, as discussed in section 2.1. Figure 4 illustrates this clearly. On a cache line reload, if we update

[3]It is called two-phase invalidation because of its hardware reset mechanism on epoch counter overflow.

[4]The timetag has the same size as the epoch counter. Our experimental results in section 4 show that a 4-bit or 8-bit timetag will give a good performance. The extra bits needed for each cache word are thus very small.

[5]The maximum value of offset must be less than or equal to $2^{R_{counter}}$ -1, which represents the range of epochs that the epoch counter can represent. Since smaller offsets are always conservative, the compiler needs to reset offsets larger than the maximum back to the maximum value by the compiler.

the timetags of both variables A and B in the same cache line with the current value *(e)* of the epoch counter, and if another task in the same epoch also updates variable B (as shown in Figure 4), two versions of variable B with the same timetag will be created. Then, the access to variable B by the processor *i* in the epoch *e + 1* will become a stale access. This can be prevented by assigning a timetag of *(e - 1)* to the variable B.

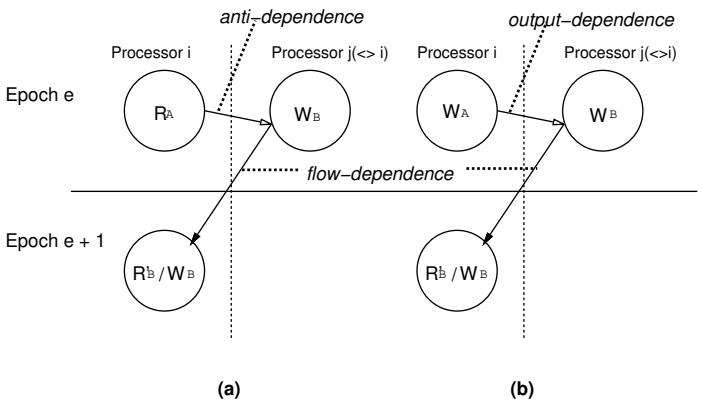

Figure 4: Stale access resulting from implicit dependences with multi-word lines. Variables A and B are assumed to be in the same cache line

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.

Epoch counter overflow and two-phase invalidation 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.

To minimize such unnecessary invalidations, we propose a *two-phase hardware reset mechanism* that can simultaneously invalidate only those cache entries whose timetags are out of phase. The idea is to use the most significant bit (MSB) of the epoch counter as an overflow flag, as shown in Figure 5. The overflow condition is detected whenever the MSB of $R_{counter}$ changes either from 1 to 0 or from 0 to 1. This implies that the largest epoch number is $2^{n-1} - 1$, where n is the number of bits in the $R_{counter}$. When this event occurs, all the cache entries whose MSB of the timetag is the same as the new MSB of the current $R_{counter}$ are identified and invalidated by the reset hardware shown in Figure 5. This is necessary because, otherwise, we cannot distinguish the old entries whose MSB of its timetag is the same as the MSB of current epoch counter from the new entries generated in the current epoch. When the MSB of the epoch counter switches from 1 to 0, the timetag numbers of the old cache entries may become greater than the current epoch counter value. During this phase, to avoid getting negative offsets, the epoch counter should be extended with one extra 1 bit (a borrow bit). This guarantees that the condition, timetag $\geq$ ($R_{counter}$ - offset), will be met for a cache hit for a Time-Read operation. The borrow bit is ignored during the phase in which the MSB of the

epoch counter is 1.

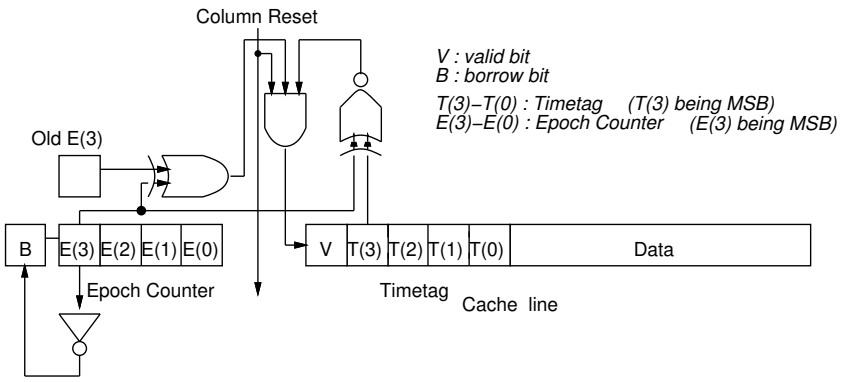

Figure 5: Reset hardware for two-phase invalidation.

Since overflows occur in two phases during which the MSB of $R_{counter}$ is 0 and 1, we call it *two-phase invalidation*. If $R_{counter}$ is large enough, it should take a long time before the epoch counter overflows. In most cases, the old entries whose MSB of the timetag is the same as that of the current epoch should have long been replaced. By keeping most of the recent cache entries, instead of flushing the entire cache, this overflow mechanism can capture more locality on timetag overflow than previous version or timestamp-based schemes. The number of phases can be extended easily to more than two if more significant bits are used to identify each phase.

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** This technique 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 of 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 the demand-driven symbolic analysis using GSA form [4].

- **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 2(b) shows the memory operations marked by a compiler for the program example shown in Figure 2(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.[6] 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 because of the unknown symbolic index value.

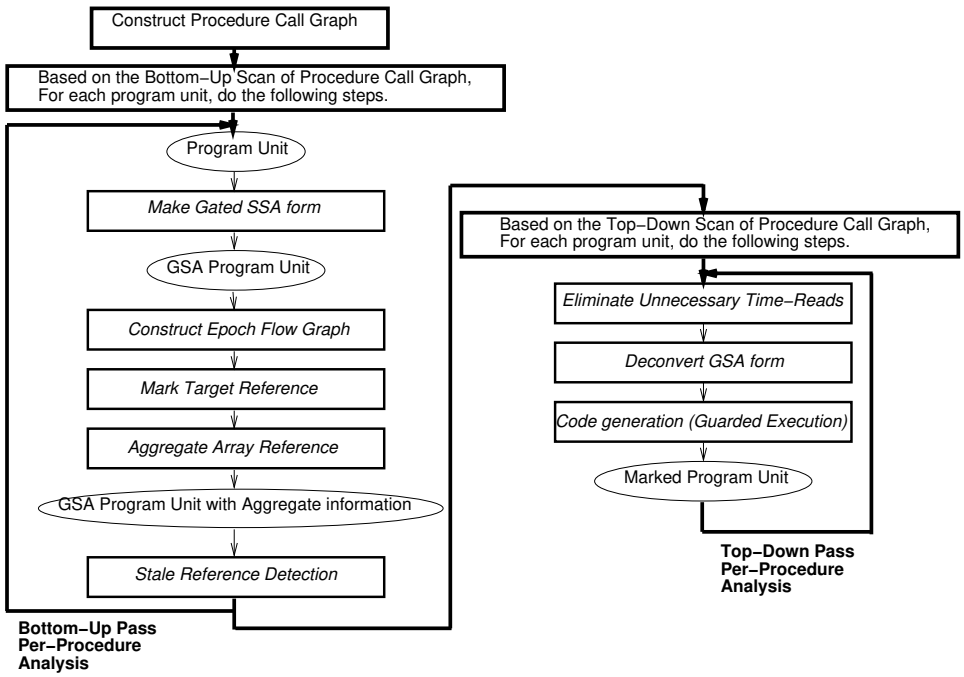

Figure 6: Flowchart for the reference marking algorithm.

Without interprocedural analysis, cache invalidations would be needed before and after each procedure call to avoid side effects. We eliminate such invalidations by using an interprocedural analysis. Time-reads with offsets 2 and 3 are generated for the last two read references to

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

variables W and X, assuming they are modified 2 and 3 epochs before the procedure Q returns. The details of compiler algorithms are beyond the scope of this paper and are described in [15, 16].

Compiler implementation All the compiler algorithms explained earlier have been implemented on the Polaris parallelizing compiler [33]. Figure 6 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 is based on the following steps. First, we transform the source program into GSA form [4]. Then, we construct a modified flow graph, called the *epoch flow graph* [18]. It contains the epoch boundary information as well as the control flows 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 target reference detection, the referenced regions of an array 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.

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 code generation [15, 16].

The compiler marking algorithms developed here are general enough to be applicable to other compiler-directed coherence schemes [10, 12].

3 Hardware implementation issues

Off-chip secondary cache implementation Since most of today's multiprocessors use off-the-shelf microprocessors, it would be more cost effective if the proposed *TPI* scheme can directly be implemented using existing microprocessors.

Figure 7 shows a possible off-chip implementation of *TPI* scheme and its logic circuits necessary to determine a cache hit for a Time-Read operation. In the implementation, the timetag hardware and 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.

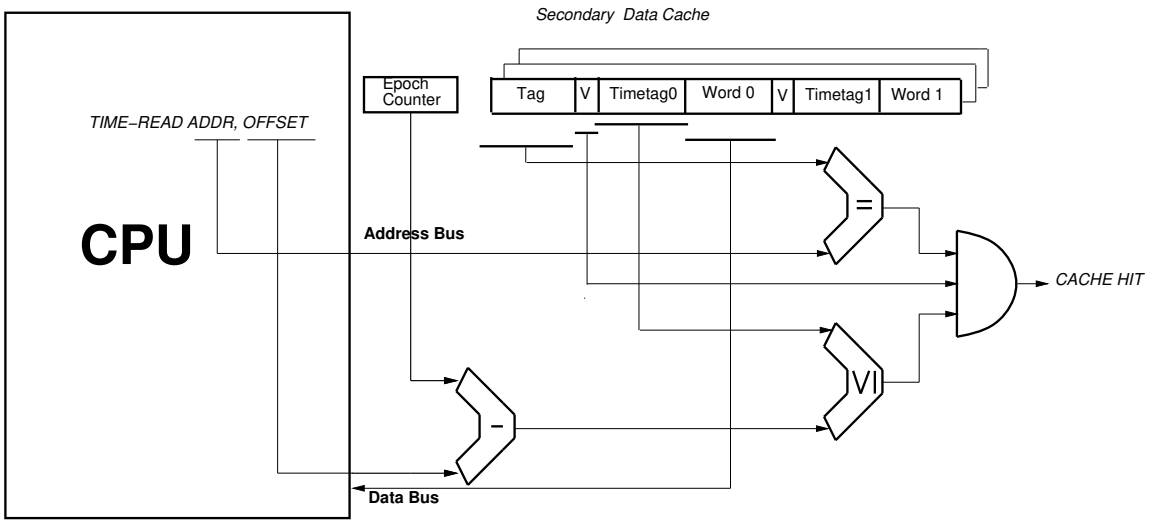

Figure 7: Off-chip implementation for *TPI* scheme.

- **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 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 a regular read operation can be used to cause 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.[7] 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 [22]. 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 increase only 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[8] 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,[9] the extra hardware overhead for *TPI* scheme is reasonably small. Table 1 compares the storage overhead of our *TPI* scheme with a full-map directory scheme [8] as well as the LimitLess directory scheme [2]. It shows both the cache (SRAM) and memory overhead (DRAM) in terms of the cache line size (L), node cache size (C), node memory size (M) and number of processors (P). The total storage overhead was computed assuming a 4 64-bit word cache line (L = 4), 256KB cache (C = $8*10^3$), 64 MB node memory (M = $2*10^6$) and 1024 processors (P = 1024).

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

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

[9] As noted by [32, 39], 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.

	Full-map Directory[8] (bits)	LimitLess Directory (DIR_i^{NB})[2] (bits)	Two-phase Invalidation (bits)
Cache Overhead (SRAM)	2*C*P	2*C*P	8*L*C*P
Memory Overhead (DRAM)	(P+2)*M*P	(i+2)*M*P	None
Total (P = 1024, i = 10)	4MB SRAM / 64.5GB DRAM	4MB SRAM / 3GB DRAM	64MB SRAM only

Table 1: Cache and memory overhead of different hardware cache coherence schemes using the following parameters: P : number of processors, L : number of words in a memory block, C : cache size in blocks, M : memory size in blocks. Total storage overhead for each coherence scheme is divided into cache overhead (SRAM) and memory overhead (DRAM) respectively.

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 accomplished 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 [19], such redundant write traffic can be reduced effectively [9]. Note that ordinary write buffers can help hide latencies but cannot eliminate redundant write traffic.

For a write-back cache configuration, *TPI* scheme should force all global writes to be written back to the main memory at synchronization points. Therefore, dirty cache copies cannot be kept in the cache across synchronization points. In the hardware directory schemes with an ownership-based protocol, memory writes can be delayed beyond the synchronization points since writes can be performed locally if the cached copy is in the exclusive state. Even when the memory is not updated, the directory can forward the data from the processor owning the up-to-date copy to the requesting processor. This difference will create less write traffic in the hardware directory schemes than in *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 [5] 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 [15], which are also implemented in the Polaris compiler. After compiler marking, we instrument the benchmarks to generate memory events, which are used for the simulation. Figure 8 shows the experimentation tools used in our simulations.

Simulation Execution-driven simulations [34] 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

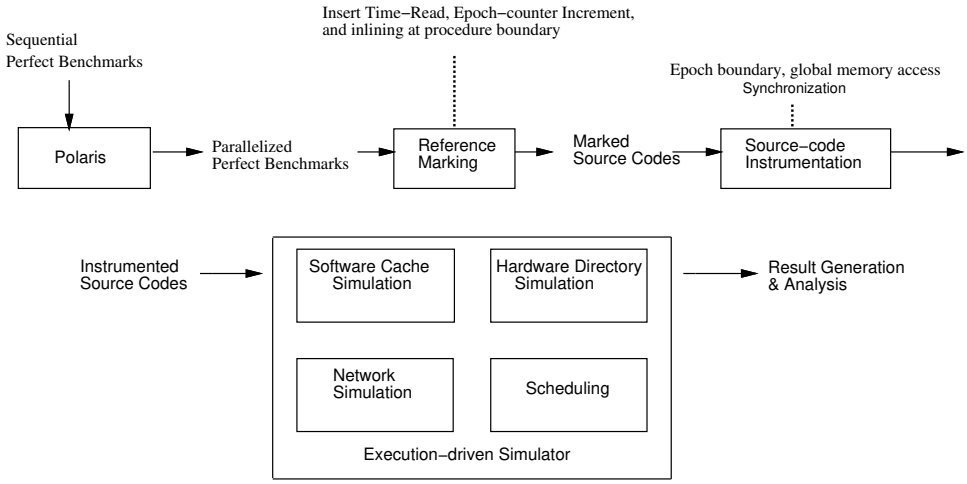

Figure 8: Experimental tools and their functions.

on-chip 64-KB direct-mapped lock-up free data cache with a 4-word cache line and an infinite size write buffer.[10] The default values of the parameters used for typical simulations are given in Table 2. A write-through write-allocate policy is used for both *TPI* and *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 [9]. 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 that it 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 [26].

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.[11] Since we are

[10] We use ordinary write buffers for simulations. Therefore, redundant writes are not merged. The use of write buffers will decrease only the stall times of the CPU, and not the network traffic. The assumption of the infinite write buffer will decrease the CPU stall times during the simulation compared to a fixed size write buffer. Chen [9] 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.

[11] 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 [35].

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 [26]
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 2: Parameters used for typical simulations.

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 3: Dynamic memory reference statistics for the Perfect Club benchmarks.

interested in comparing different coherence schemes for shared data references, these are not considered in our simulations.

Dynamic reference statistics Table 3 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 that results from parameter passing. The number of Time-Read operations shows the potentially stale data references detected through our interprocedural analysis. Depending on the application used, 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 the smallest.

4.2 Impact of hardware support

In this section, we investigate how cache coherence support impacts the overall system performance. Four different hardware and compiler-directed coherence schemes are studied. They

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 intertask 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 4: Characteristics of different hardware and software coherence schemes.

differ in 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 *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.

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 [8, 3] but without broadcasting. It gives a performance comparison to the hardware directory protocols.

Table 4 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.[12]

Miss rates Figure 9 shows the miss rates of each scheme for the six benchmarks with a 64KB direct-mapped cache. The impact of different cache organizations are discussed in section 4.4. An interprocedural analysis is performed for both *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 either the cached data has been invalidated or a

[12] *HW* also scheme can employ compiler techniques, such as migratory sharing optimizations, but this compiler support is optional since coherence can be maintained regardless of the compiler support. However, the compiler algorithms are essential for coherence enforcement in both *SC* and *TPI*.

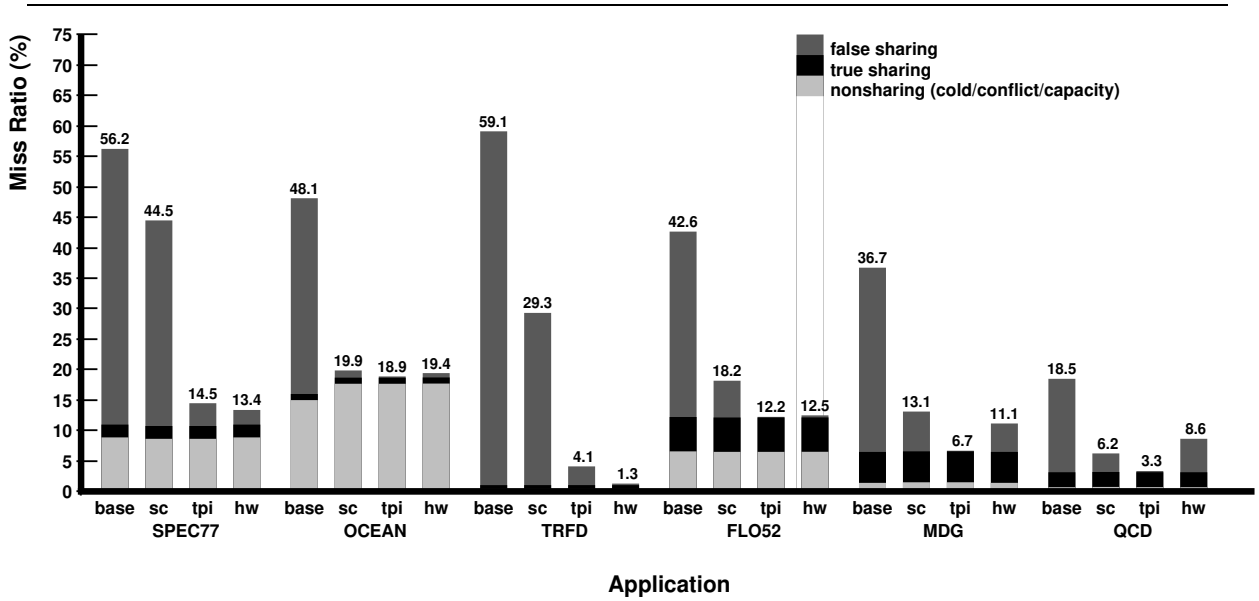

Figure 9: Distribution of misses for each coherence scheme.

timetag mismatch has occurred. All other misses are nonsharing misses. The sharing misses are further classified into true and false sharing misses. The *true sharing misses* are necessary to satisfy cache coherence, while false sharing misses are unnecessary misses resulting from lack of compile time information or the false-sharing effect of hardware protocols. *False sharing misses* are identified during simulations using the method in [36]. 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 be counted also as a false sharing miss.

With compile time analysis alone (*SC*), a significant number of unnecessary sharing 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, for the other benchmarks there remain a significant number of false sharing misses, resulting from the conservative compiler decisions on branches, memory disambiguation, and scheduling. In addition, locality is not preserved for any shared read references across epoch boundaries.

In *TPI*, the timetag provides local timing information for runtime checking, thus enhancing the cacheability 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

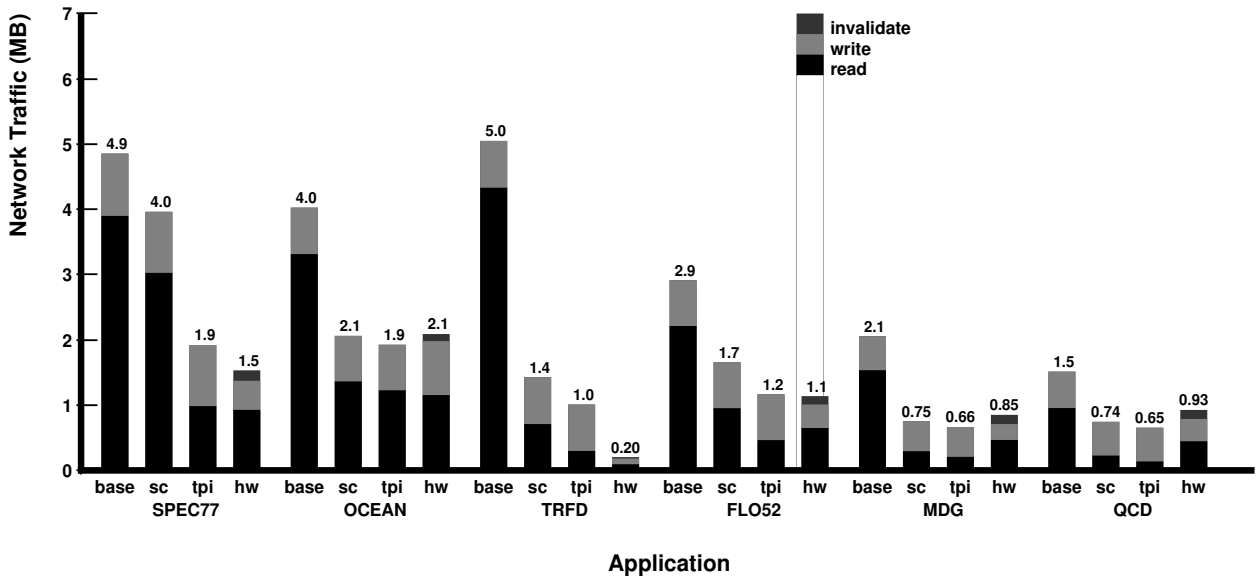

Figure 10: Distribution of network traffic for each coherence scheme.

conservative compile time 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, for all the benchmarks tested *TPI* shows miss rates comparable to the hardware directory scheme (*HW*). 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 because of 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 about compiler-directed schemes has been the increased write traffic caused by their write-through policy.[13] Figure 10 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 write traffic in both compiler-directed schemes is small, except for TRFD. In TRFD, there is a significant number of redundant writes that increases the overall network traffic of *TPI* substantially compared to *HW*. This

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

Program	Average Miss Latency			
	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 5: Average miss penalty of *TPI* and *HW* schemes.

additional write traffic can be eliminated effectively by organizing a write buffer as a cache [9]. A 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 network traffic is its impact of on miss penalty. Usually, higher network loads imply a higher miss penalty. Hardware directory schemes require a higher miss penalty because of coherence-related transactions. Since we use a weak consistency model with an infinite write buffer, write latency can be hidden completely 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. This problem would be much more significant in a sequential consistency model since both reads and writes are affected by the coherence transactions.

Table 5 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% 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 write traffic is negligible since writes can be buffered and the overall traffic rate for the application is still low. However, the migratory sharing patterns of the application increases the miss penalty of *HW* by 9.63% compared to *TPI*.

Execution time Figure 11 displays the distribution of execution times partitioned into busy cycles (CPU time), memory cycles (waiting time spent on memory accesses), and synchronization cycles (waiting time due to barrier synchronizations) that are averaged over 16 processors. The remainder of this paper focuses on the performance of only *TPI* and *HW* schemes since they demonstrate a significant performance improvement over the other schemes by caching shared data: speedups of at least 3 to 4 times over *BASE* scheme from our simulations. In the figure, we also show the execution time of the off-chip implementation of *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 as discussed in section 3,

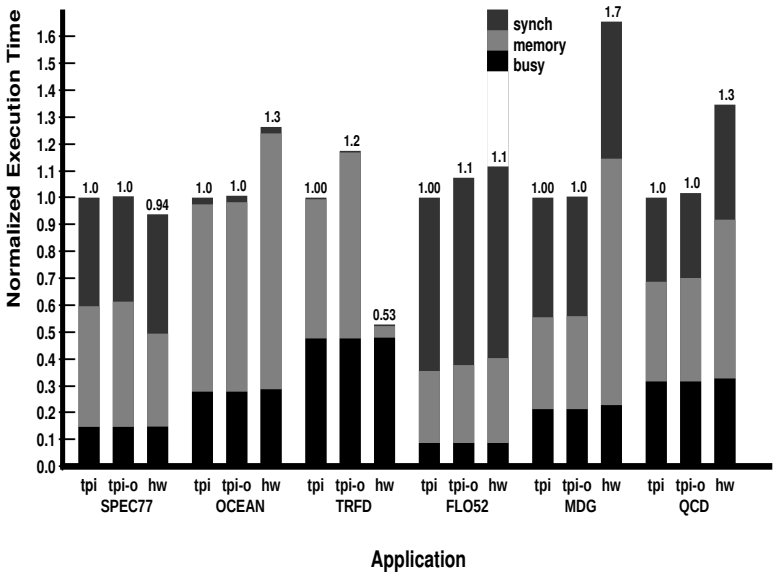

Figure 11: Distribution of execution time for each coherence scheme normalized to *TPI* scheme.

here we assume only an off-chip timetag hardware implementation without secondary caches.

The memory stall times constitute the largest portion of 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* resulting from both small miss penalty (136 cycles instead of 178.7 cycles, see Table 5) and smaller miss rates (6.7% instead of 11.1%, see Figure 9) of *TPI*. However, for TRFD, *TPI* shows an execution time that is 88.7% longer than that of *HW*. This increase is primarily a consequence of 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 with 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 nonblocking caches we used, the extra off-chip access penalty can be hidden most of 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 off-chip implementation is comparable to both the on-chip *TPI* scheme and *HW* scheme. This suggests that the off-chip *TPI* scheme using the off-chip microprocessors would be a good practical solution.

4.3 Impact of timetag size on intertask locality

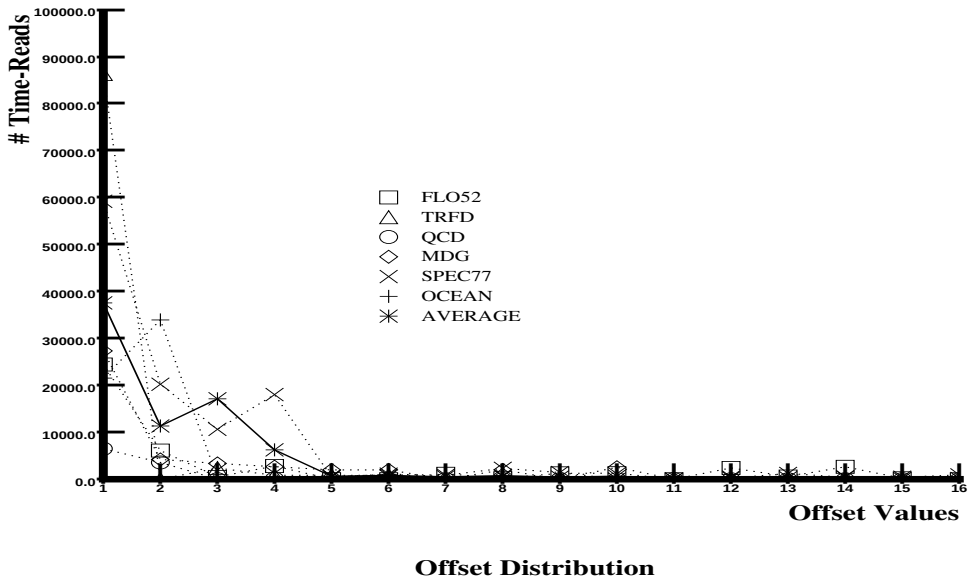

Figure 12: Distribution of offsets for Time-Read.

The size of the timetag determines how far *TPI* can exploit locality across epoch boundaries. The intertask locality can be identified by studying the offset values of the Time-Read operations. Figure 12 shows the distribution of offsets for the six benchmarks. Most offsets are small, implying that the distance between 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, which occurred in SPEC77. This implies that a timetag size as small as 3 or 4-bits might be sufficient to capture most of the inter-epoch locality. Moreover, the larger the offset, the more likely it is to be replaced before it is used in later epochs.

However, small timetags can lead to frequent invalidations from timetag overflows. To compare the impact of this invalidation on the epoch counter overflow, we simulate both the two-phase reset mechanism and the entire cache invalidation on the epoch counter overflow. Figure 13 shows the miss rates of *TPI* 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 most of unnecessary invalidations as compared the entire cache invalidation. With a 1-bit timetag, both reset mechanisms do little since most inter-epoch locality distances are 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.

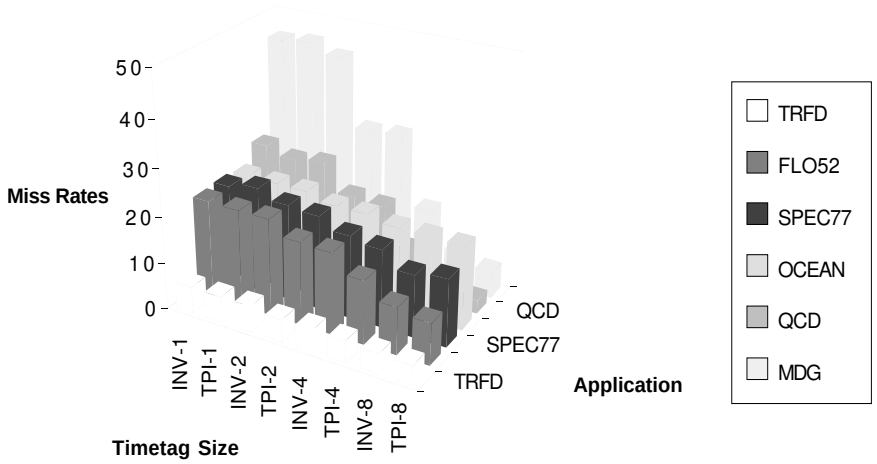

Figure 13: Impact of timetag size and reset mechanism on epoch counter overflow.

4.4 Impact of cache organizations

So far, we assumed a 64KB direct-mapped cache with 4-word cache lines. In this section, we investigate the performance impact of different cache design parameters.

Cache line size The choice of cache line size can have a critical impact on cache performance. Figure 14 shows cache miss rates for both *TPI* and *HW* schemes 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, thereby decreasing 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 locality exploited by a large cache line reduces cache misses for most benchmarks.

The next question is regarding the negative side effect of large cache lines (i.e., increased miss penalty and network loads). Table 5 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 15 which shows the execution times with both small and large cache lines normalized to the execution time of *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.

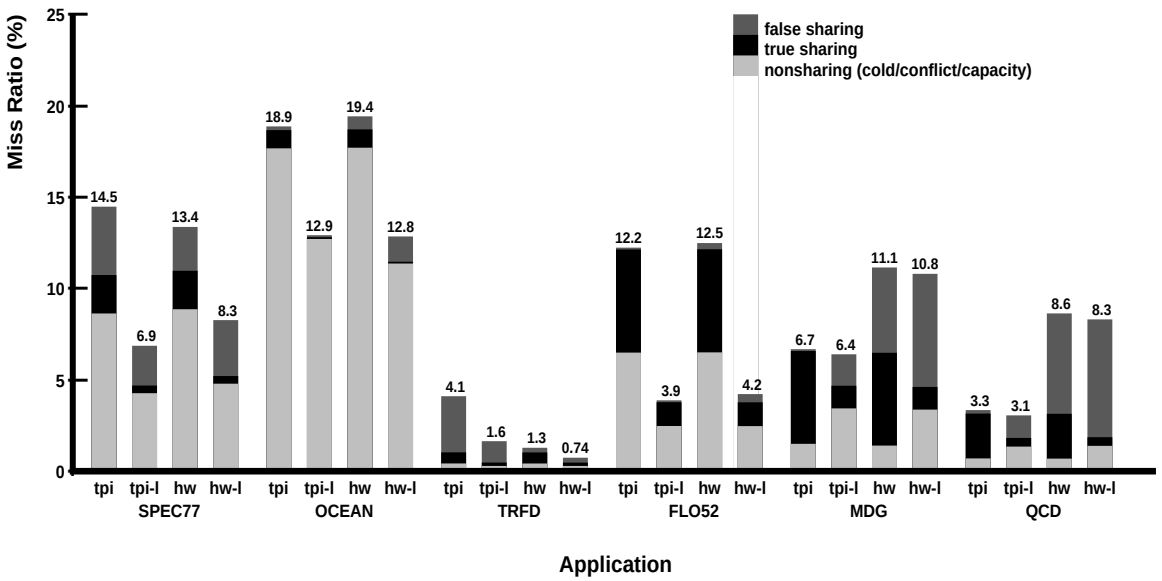

Figure 14: Miss rates for small (16 bytes) and large (64 bytes) cache line sizes (represented by tpi-l and hw-l for *TPI* and *HW* schemes, respectively).

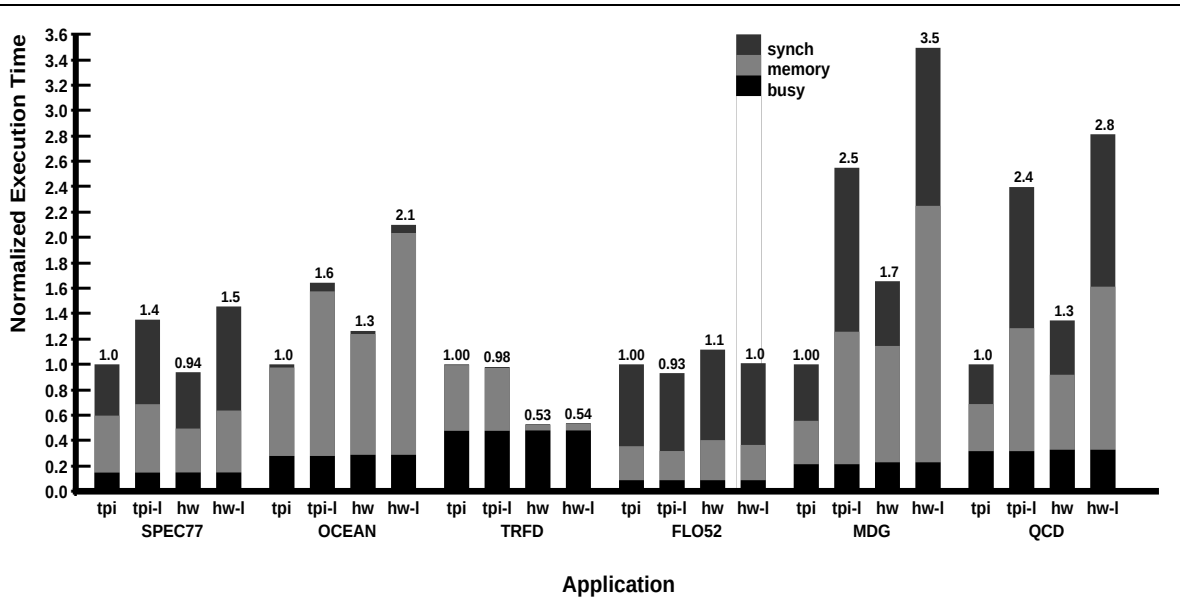

Figure 15: Normalized execution time for small (16 bytes) and large (64 bytes) cache line sizes.

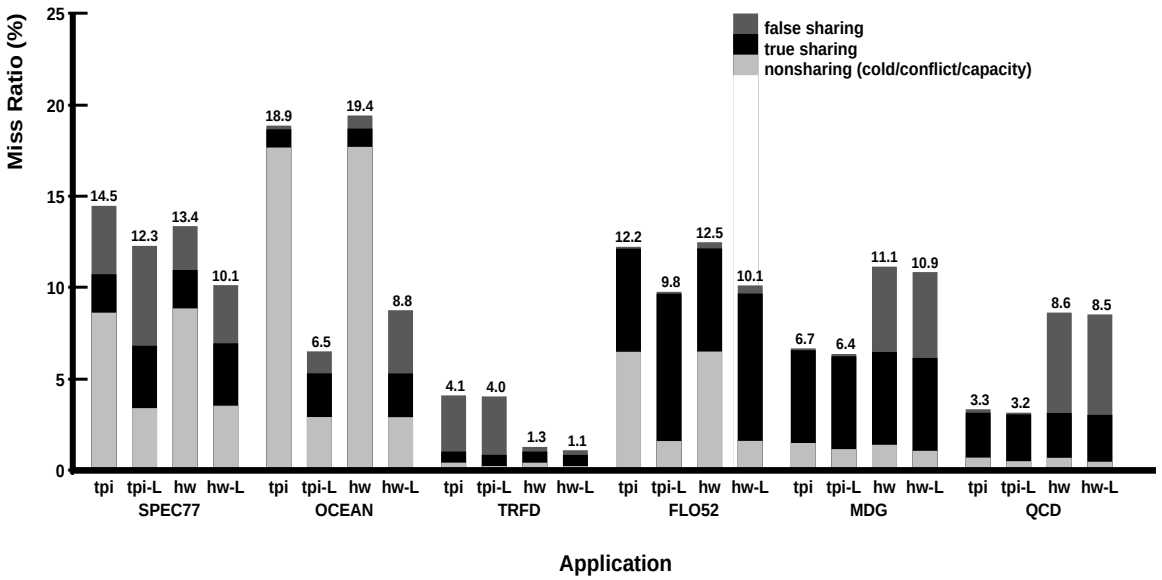

Figure 16: Miss rates for small (64 Kbytes) and large (1 Mbytes) cache sizes (represented by tpi-L and hw-L for *TPI* and *HW* schemes, respectively).

Cache size As VLSI technology advances, large on-chip primary and off-chip secondary caches are becoming popular. To investigate the performance impact of the cache size, we performed the simulations using 1MB caches and compared its miss rates to those of the 64KB cache for both *TPI* and *HW* (see Figure 16). Overall, large caches reduce nonsharing misses by eliminating more capacity and conflict misses. In particular, for OCEAN, the large caches reduce misses rates substantially (12.4% and 10.6% for *TPI* and *HW* respectively). A similar situation occurs for SPEC77 and FLO52. However, note that large caches also increase the sharing misses by retaining cached copies for an extended period of time. This leads to a notable increase in the sharing misses for SPEC77, OCEAN, and FLO52. For MDG and QCD, the miss rates remain unchanged. In these benchmarks, both nonsharing and sharing misses are unaffected by large caches. This implies that 64KB caches are already large enough for these benchmarks to capture most of the actively shared data.

Set-associativity Given more chip area, another way of increasing cache hit rates is to use set-associative caches. Set-associative caches decrease conflict misses but may increase the cache cycle time due to their associative tag searching. Figure 17 shows the miss rates for both direct-mapped and 4-way set-associative caches. Overall, the decrease in the nonsharing misses is only modest for all the benchmarks. In fact, for *TPI* scheme, the set-associative cache increases the miss rates slightly due to increased sharing misses for SPEC77, TRFD, and MDG. This indicates that large direct-mapped caches are a better alternative than set-associative caches.

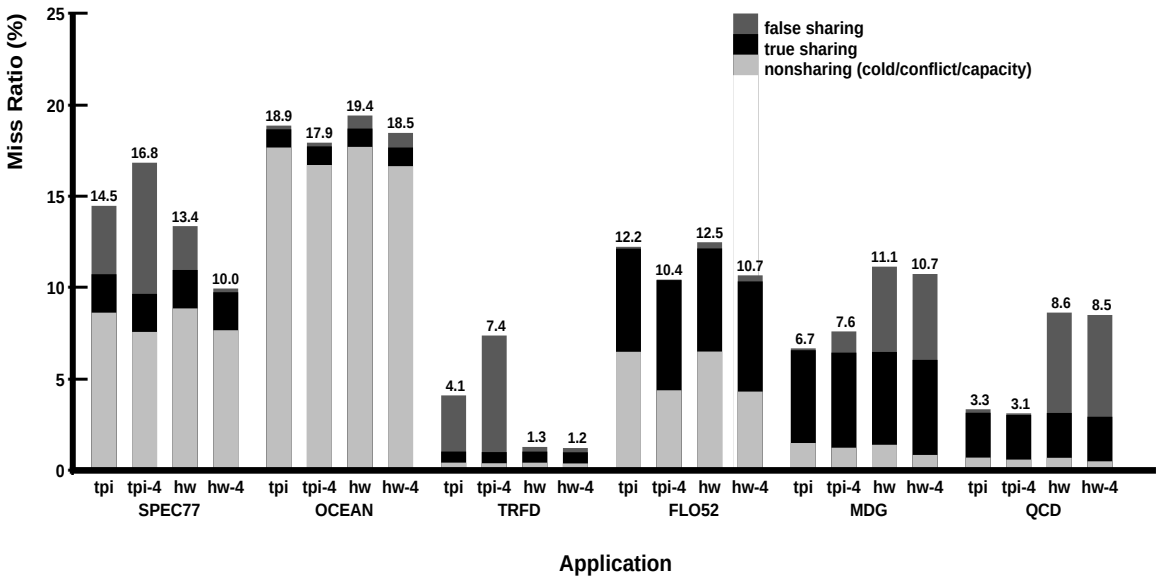

Figure 17: Miss rates for direct-mapped and 4-way set-associative caches (represented by tpi-4 and hw-4 for *TPI* and *HW* schemes, respectively).

5 Discussion

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

Handling DOACROSS For parallel threads in an epoch that 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 read operations are performed in different processors in the same epoch. In this case, the processor that performs the write will make stale the cache copy that was brought in by the processor performing the read. A similar situation occurs for a Write-after-Write dependence (an output-dependence). A Read-after-Write dependence (a flow dependence), however, 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 18. 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, 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 as well as the read reference in S2 would

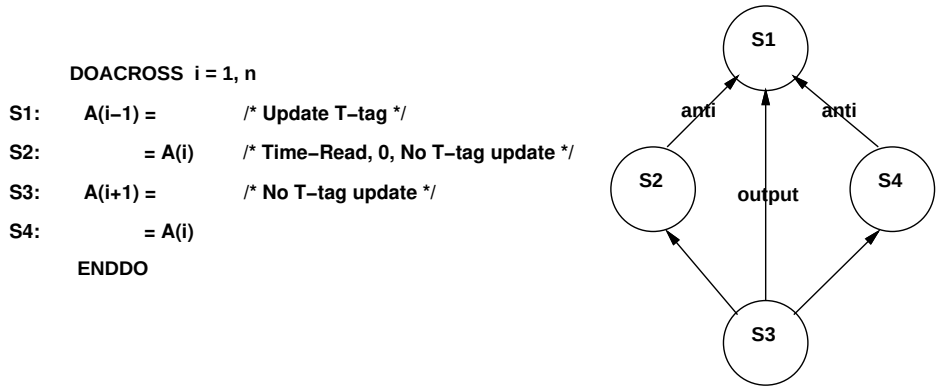

Figure 18: Example of compiler marking in a doacross loop.

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 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 18, 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, thus producing 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.

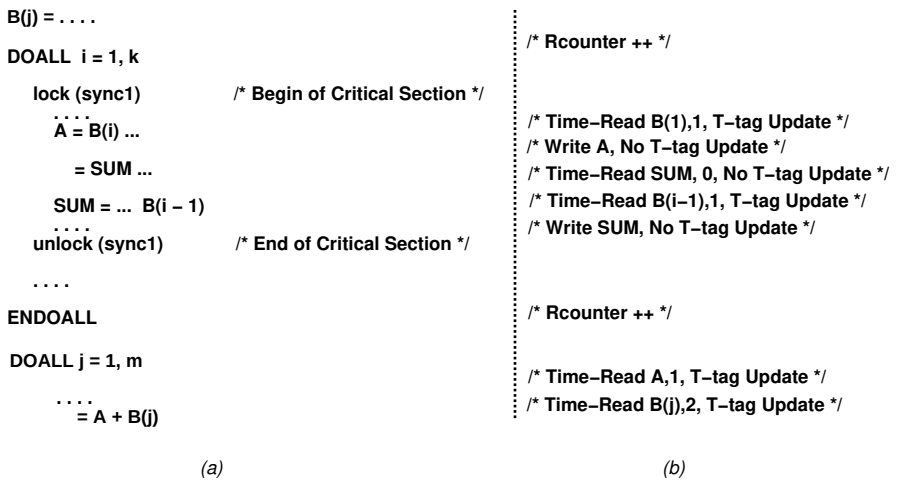

Figure 19: Program Example Including Critical Section.

Consider the program example in Figure 19(a). All participating 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 19(a), the read references to B(i) and B(i-1) are safe because array B is read-only in the epoch. Figure 19(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 because of 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, yet also provides a reasonable amount of hardware support. This approach has a long history of predecessors, including C.mmp [38], IBM's RP3 [6], Illinois Cedar [27], and several recently proposed schemes [10, 14, 12, 13, 18, 21, 29, 30].

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, and inter-thread communication, are also addressed. The cost of the required hardware support is minimal and proportional to the cache size. The necessary compiler algorithms, including intra- and interprocedural array data flow analysis, have been implemented on the Polaris paralleling compiler [33].

The results of our simulation study using the Perfect Club Benchmarks [5] show that both hardware directory schemes and the *TPI* scheme have comparable number 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 conservative decisions made by the compiler is comparable to the degradation caused by false sharing and increased miss penalty in the hardware directory scheme.

Given its comparable performance and lower hardware costs compared to the hardware directory schemes, the proposed *TPI* scheme can be a viable alternative for machines without hardware-coherent caches, such as the Cray T3D. 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 of the University of Illinois who have participated in the review of an early version of this paper, particularly Vijay Karamcheti for his thoughtful comments and Tae Hyung Kim of the University of Maryland for his early comments and encouragement. We express thanks to Prof. Sang Lyul Min of Seoul National University and Prof. David Lilja of the University of Minnesota for proofreading the paper and their valuable comments. Special thanks go to David Poulsen of Kuck and Associates, Inc. for his numerous help with the development of execution-driven simulations, and to Lawrence Rauchwerger and IBM for providing RS6000 clusters for simulations.

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] A. Agarwal et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, 1991.

[3] J. Archibald and J. Baer. An Economical Solution to the Cache Coherence Problem. *Proceedings of The 11th Annual International Symposium on Computer Architectur*, pages 355–362, June 1984.

[4] 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.

[5] M. Berry and others. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall, 1989.

[6] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 processor-memory element. *Proceedings of the 1985 International Conference on Parallel Processing*, pages 782–789, August 1985.

[7] IBM Inc. Edited by C. May, E. Silha, R. Simpson, and H. Warren. The PowerPC Architecture: A Specification for a New Family of RISC Processors. March 1995.

[8] 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.

[9] 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.

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

[11] H. Cheong and A. Veidenbaum. Stale Data Detection and Coherence Enforcement Using Flow Analysis. *Proceedings of the 1988 International Conference on Parallel Processing*, I, Architecture:138–145, August 1988.

[12] H. Cheong and A. Veidenbaum. A Cache Coherence Scheme with Fast Selective Invalidation. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988.

[13] H. Cheong and A. Veidenbaum. A Version Control Approach To Cache Coherence. *Proceedings of the 1989 ACM/SIGARCH International Conference on Supercomputing*, June 1989.

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

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

[16] Lynn Choi and Pen-Chung Yew. Compiler Analysis for Cache Coherence: Interprocedural Array Data-Flow Analysis and Its Impacts on Cache Performance. *submitted to IEEE Transactions on Parallel and Distributed Systems*, Feb. 1996.

[17] Lynn Choi and Pen-Chung Yew. Compiler and Hardware Support for Cache Coherence in Large-Sca le Multiprocessors: Design Considerations and Performance Study. *To appear in the 23rd Annual ACM International Symposium on Computer Architecture*, May 1996.

[18] 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.

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

[20] Intel Corporation. Paragon XP/S Product Overview. 1991.

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

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

[23] J. Edler, A. Gottlieb, C. P. Kruskal, K. P. McAuliffe, et al. Issues related to MIMD shared-memory computers: the NYU Ultracomputer approach. *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 126–135, June 1985.

[24] Cray Research Inc. Cray T3D System Architecture Overview. Mar. 1993.

[25] MIPS Technology Inc. R10000 User's Manual, Alpha Revision 2.0. March 1995.

[26] 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.

[27] D. Kuck, E. Davidson, et al. The Cedar System and an Initial Performance Study. *Proceedings of the 20th Annual International Symposium on Com puter Architecture*, pages 213–223, May 1993.

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

[29] A. Louri and H. Sung. A Compiler Directed Cache Coherence Scheme with Fast and Parallel Explicit Invalidation. *Proceedings of the 1992 International Conference on Parallel Processing*, I, Architecture:2–9, August 1992.

[30] S. L. Min and J.-L. Baer. A Timestamp-based Cache Coherence Scheme. *Proceedings of the 1989 International Conference on Parallel Processing*, I:23–32, 1989.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] C. G. Bell W. A. Wulf. C.mmp - a multi-mini processor. *Proceedings of the Fall Joint Computer Conference*, pages 765–777, December 1972.

[39] 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.