Appears in the proceedings of the ACM/IEEE Supercompuing '94

A Compiler-Directed Cache Coherence Scheme with Improved Intertask Locality

Lynn Choi
Center for Supercomputing R & D
University of Illinois
Urbana, IL 61801-1351

Pen-Chung Yew
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455-0519

Abstract

In this paper[1], we introduce a compiler-directed coherence scheme which can exploit most of the temporal and spatial locality across task boundaries. It requires only an extended tag field per cache word, one modified memory access instruction, and a counter called the epoch counter in each processor. By using the epoch counter as a system-wide version number, the scheme simplifies the cache hardware of previous version control [5] or timestamp-based schemes [12], but still exploits most of the temporal and spatial locality across task boundaries. We present a compiler algorithm to generate the appropriate memory access instructions for the proposed scheme. The algorithm is based on a data flow analysis technique. It identifies potential stale references by examining memory reference patterns in a source program.

1 Introduction

Reducing memory latency is critical to the performance of large-scale parallel systems. Due to the temporal and spatial locality of memory reference patterns, private caches can reduce the number of memory accesses, decreasing both the average memory latency and the network traffic. Having multiple cached copies of a shared memory location, however, can lead to erroneous program behavior unless coherence is maintained.

Existing solutions for large-scale multiprocessors include hardware directories [1, 10] and software techniques [2, 3, 5, 6, 11, 12, 14, 15]. By maintaining sharing information at runtime, directory-based schemes can identify stale data accurately, preserving more temporal locality than the compiler-directed schemes. However, the directory storage and its complex directory controller substantially increase the hardware cost.

As an alternative, software techniques can be used to maintain coherence. They usually require both compiler and hardware support to detect possible stale accesses and to invalidate cache entries without the need for interprocessor communication or shared directories. Software coherence schemes that can achieve reasonable performance are attractive because they avoid expensive hardware.

In this paper, we propose a compiler-directed scheme which preserves most of temporal and spatial locality across task boundaries. In this scheme, each epoch is assigned a unique epoch number similar to the version number in previous schemes [2, 5, 6, 12]. The epoch number is maintained at runtime using a counter called the *epoch counter*. Each data in a cache is associated with a *time-tag* that records the epoch number in which the cache copy is created. It requires only one time-tag field per cache word, one modified memory access instruction, and an epoch counter in each processor. The time-tag of each cache word need not be stored back to the main memory when the cache line is replaced. Our simulations also show that using 4 bit time-tag allows reasonably good performance. The scheme supports a model of parallel program execution which includes imperfectly nested parallel loops with or without cross-iteration dependences, including critical sections [7].

Using the epoch counter as a system-wide version number has two advantages. First, it eliminates the storage requirement of keeping the version number of *each variable* as in the version and the timestamp-based schemes [5, 12]. It only needs a *single* system-wide version number during execution. This significantly simplifies the hardware for the cache controller. Second, since time-tags are no longer associated with

[1] This work is support in part by the National Science Foundation under Grant No. MIP 89-20891, MIP 93-07910. ISSN 1063-9535.

each memory reference, but are rather on a per-epoch basis, the time-tags of the variables preloaded on a miss in a multi-word cache line can be computed at runtime by the hardware. This allows the new scheme to preserve spatial locality with multi-word cache lines or in the presence of hardware prefetching techniques. Preserving spatial locality is difficult in version or timestamp-based schemes because on a miss, version information cannot be determined for the preloaded variables in the same cache line (see section 2.2). Similar situations occur in several recent schemes [2, 6, 14]. Our scheme can also avoid flushing the entire cache when the epoch counter overflows by using an overflow reset mechanism.

In the following, we first characterize existing compiler-directed schemes in section 2.1 and discuss our motivation for the proposed scheme in section 2.2. In section 3.1, we present our program execution model, which allows nested doall loops (i.e., loops without cross-iteration dependences) as well as doacross loops (i.e., loops with cross-iteration dependences). The coherence mechanism and its hardware support is presented in section 3.2. In section 4, we present a flow analysis algorithm that automatically identifies potential stale references in a program and generates appropriate memory instructions. Section 5 concludes the paper. A proof for the correctness of the new scheme and the algorithms to handle synchronizations and critical sections within parallel loops are presented in the extended version of this paper [7]. A preliminary performance study of the proposed scheme compared with existing hardware and software coherence schemes can be found in [7].

2 Background

This section describes the concepts used in the previous compiler-directed coherence schemes (see Table 1).

2.1 Stale access detection, prevention and avoidance

Compiler-directed coherence schemes maintain coherence through *detection*, *prevention*, or *avoidance* of stale accesses. Existing software coherence schemes use combinations of these techniques (see Table 1).

Stale access *detection* refers to the use of *compile time* analysis to identify potential references to stale data [4]. By identifying these potential stale references at the compile time, the system can be forced to get up-to-date data directly from the main memory, instead of from the cache.

Stale access *prevention* techniques invalidate or update stale cache entries before stale accesses occur. Simple invalidation [15] maintains cache coherence by flushing the cache at task boundaries. If there is no read-write dependences among processors, e.g., during the parallel execution of a doall loop, no stale accesses will occur in each processor. The fast selective invalidation scheme [3] combines the techniques of *prevention* and *detection*, and extends the simple invalidation scheme to exploit intertask locality for safe (e.g., read-only) accesses. The life span strategy [2], Peir and So's scheme [14] and the generational algorithm [6] allow invalidation on a per-variable basis. In these schemes, a variable is not invalidated until the next write to that variable occurs in a different task. The compiler identifies the task level in which the cache copy needs to be invalidated. This strategy allows more temporal intertask locality to be exploited for shared read-write data. Parallel explicit invalidation [11] uses a more refined selective invalidation technique. The scheme can invalidate the stale cache entries in only the part of an array which is modified. This is done by allocating elements of an array in a structured form and by generating a masking bit pattern for each write reference. Darnell and Kennedy [9] proposed an invalidation scheme that combines the idea of parallel explicit invalidation with a 1-bit timestamp scheme. Most hardware cache coherence schemes use prevention techniques.

Stale access *avoidance* is a more relaxed coherence model than the prevention technique. It allows the existence of stale data but avoids stale accesses to them at run time. Examples of this approach include the timestamp-based scheme [12] and the version control scheme [5]. In such schemes, stale data can exist until an access to the data occurs. At that time, by checking the status of the data in the cache, it is decided whether an up-to-date copy is brought in from main memory or not. For example, in the version control scheme, each cache access compares the birth version number (BVN) of the cached data with the current version number (CVN) of the variable and determines whether the cached copy is stale. An up-to-date copy is brought in if it is stale.

2.2 Limitation on multi-word cache lines

In some existing compiler-directed schemes that exploit temporal locality across task boundaries [2, 5, 12, 14], version information is maintained for each cache copy, which is used to determine whether the copy is

	SI	FSI	VC/Time	Lifespan	Two-Phase Invalidation
Coherence Technique	Prevention	Detection, Prevention	Avoidance	Detection, Prevention	Detection, Avoidance
Intertask Locality	None	Read-only Data	Temporal	Temporal	Temporal, Spatial
Storage Overhead	O(c)	O(c)	O(logN) + Version Storage	O(N)	O(logN)
Runtime Overhead	invalidate	invalidate	version lookup for each access and version update	invalidate (shifting cache columns)	epoch counter increment

SI: Simple Invalidation [15], FSI: Fast Selective Invalidation [3],
VC: Version Control Scheme [5], Time: Timestamp-based scheme [12],
Lifespan: Lifespan strategy [2], Two-Phase Invalidation: Our proposed scheme

Figure 1: Comparisons of existing compiler-assisted coherence schemes. N denotes the number of task levels at which locality is to be preserved while c denotes a constant.

stale or not. However, the version information is lost on the replacement of a cache line unless the version information is stored back to the memory. This is not a problem for a single-word cache line since the version information is provided by hardware [5, 12] or by an operand [2, 14] in every memory access instruction. However, with multi-word cache lines or with hardware prefetching schemes, it is a problem because it is quite difficult for a compiler to know exactly which variables are allocated in the same cache line for every cache line. Furthermore, to provide version information for every word in a cache line, the memory instructions used in [2, 14] should have as many operands as the number of words in a line. For the timestamp-based schemes, they require the cache controller to recompute the version information of each word sequentially by looking up its version storage [5, 12].

This creates a performance problem since the loss of version information may create unnecessary misses when the cache line is reloaded. It can incur a significant performance loss since the intertask locality cannot be preserved for all the preloaded variables in the same cache line. The intratask locality can still be captured if these preloaded variables are invalidated at task boundaries.

As an example, let's look at Figure 2 which shows the situation in the version control scheme. Several other compiler-directed schemes [6, 2, 12, 14] have a similar problem. In this scheme, the birth version number (BVN) is associated with every data item in the cache. In addition, the processor keeps track of the current version number (CVN) for each variable in a separate local memory, called version store. Note that the CVN is maintained on a per-array basis, because maintaining CVN for every data element in the array requires excessive version storage. To decide a cache hit, each memory access compares its CVN for the variable with the BVN of the corresponding cache copy.

Assume that at the time of LOAD B, a conflict miss occurs, replacing the cache line that contains A[0], A[1], A[2] and A[3] with the new cache line that contains B. Since LOAD B only updates the BVN of the variable B, the BVN's for the variable C, D and E in the same cache line are unknown. In addition, when A[0] is accessed again, the BVN's of A[1], A[2] and A[3] in the same cache line are lost. Although only 2 read accesses (LOAD B and LOAD A[0]) are the real misses due to the conflict, four additional misses occur due to the incomplete version information. Storing the version information in main memory will substantially increase the storage overhead, and is not a cost-effective solution. Even if the version number is stored into main memory, another problem arises because on a timestamp or version overflow, all the version information in main memory needs to be modified, which is very time-consuming. We can also use a compiler to pack the data in such a way that different variables will not be in the same cache line. However, the code will no longer be portable to a machine with a different cache organization. Parallel explicit invalidation [11] and the scheme in [9] also rely on compile time memory allocation to support multi-word cache lines.

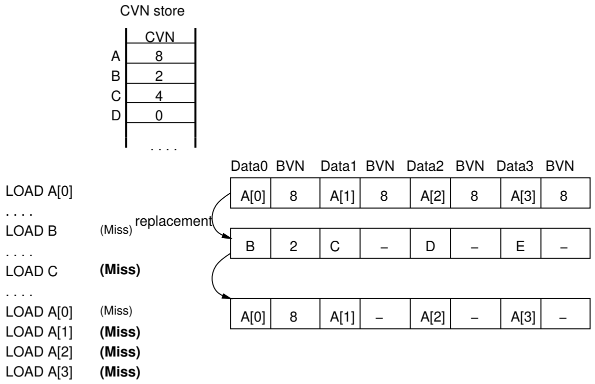

Figure 2: Loss of cache state information for the version control scheme. A bold-typed miss represents an unnecessary miss.

3 Two-phase invalidation scheme

3.1 Parallel execution model

The execution of a parallel program is viewed as a sequence of *epochs*. An epoch is either a parallel loop (*parallel epoch*) or a serial section of code (*serial epoch*) between the parallel loops (see Figure 3).

A *task* is a unit of computation that gets scheduled and assigned to a processor at runtime. In a serial epoch, a single task is scheduled and executed on a single processor. In a parallel epoch, multiple tasks are created at runtime and executed on multiple processors concurrently. Therefore, each task boundary corresponds to an epoch boundary where processor scheduling occurs. 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 (See Figure 3.) To simplify our discussion, synchronizations are assumed at each epoch boundary. Barrier synchronizations are used at the end of parallel epochs. For consistency, the memory should be up-to-date at synchronization points.

3.2 Coherence mechanism and its hardware support

In our cache coherence scheme, each epoch is assigned a unique epoch number which is similar to the version number in previous schemes [2, 5, 6, 12]. The epoch number is stored in a n-bit register, called the *epoch counter* ($R_{counter}$), and incremented at the end of every epoch.

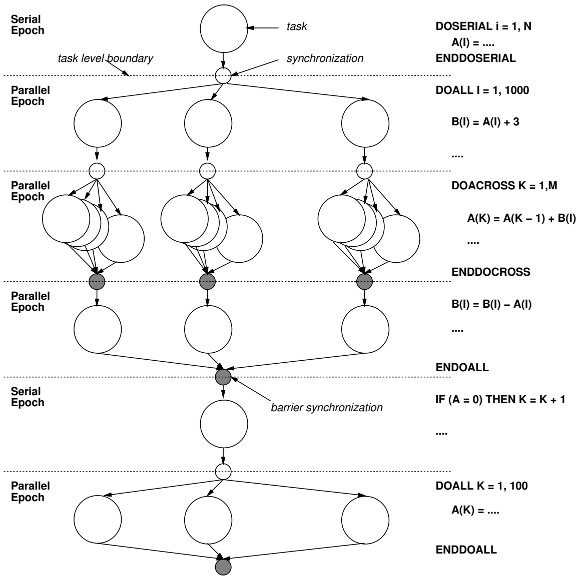

Figure 3: Epochs in a parallel program.

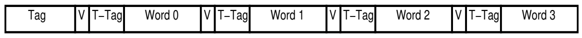

Figure 4: Fields in a 4-word cache line entry. T-Tag denotes a time-tag.

Figure 4 shows the format of a cache line with a 4-word line size. Every word in a cache is associated with an n-bit time-tag that records the epoch number when the cache copy is created.[2] The valid bit is the same as the valid bit in a conventional cache. *All* of the data that are accessed in the same epoch are assigned the *same* time-tag number at runtime. Those data accesses can refer to *different* variables.

Three memory access instructions are required to support our software scheme.

- **Read:** Read is the same as a conventional read operation. If the address tag matches and the valid bit is set, then a cache hit occurs. Otherwise, it is a miss, and the data need to be fetched from memory.

- **Time-Read:** Time-Read is a special cache read operation to check for a potential stale access.[3] A Time-Read is similar to a Read except that it is augmented with an *offset*. The *offset* indicates the

[2] The time-tag has the same size as the epoch counter. Our experimental results [7] show that n = 4, i.e. a 4-bit timetag with the two-phase invalidation mechanism will give good performance. The extra bits needed (only 4 bits) for each cache word are thus very small.

[3] Note that only a read can result in a stale access.

number of epoch boundaries between the current epoch and the epoch which contains the most recent write. On a Time-Read, the time-tag of the cache word is tested, in addition to the address tag and the valid bit, to determine a cache hit. If (1) time-tag $\geq R_{counter}$ - offset[4], (2) the address tag matches, and (3) the valid bit is set, then we have a cache hit. [5]

- **Write:** Write is the same as a conventional write operation.

In addition to normal read and write operations, every read or write instruction will update the time-tag of the accessed cache word with the current value of the epoch counter.

With multi-word cache lines, the values of $R_{counter}$ - 1 are assigned to the time-tags of other words in the same cache line when the cache line is loaded on a miss. 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 create a stale access in the following epoch as is the case for the variable B shown in Figure 5. On a cache line reload, if we update the time-tags of both variable 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 the variable B (as shown in Figure 5), two versions of variable B with the same time-tag will be created. Then, the access to the variable B by the processor i in the epoch $e + 1$ will be a stale access. This can be prevented by assigning a time-tag of *(e - 1)* to the variable B.

In summary, using the epoch counter as a system-wide version number has two advantages. First, it eliminates the storage requirement of keeping the version number of *each variable* as in the version and the timestamp-based schemes. It only needs a *single* system-wide version number during execution. Second, since time-tags are no longer associated with each memory reference, but are rather on a per-epoch basis, the time-tags of the variables preloaded on a miss in a multi-word cache line can be computed at runtime by hardware. For these variables, the previous epoch number is assigned as the time-tag. This allows the exploitation of spatial locality without storing time-tags in memory.

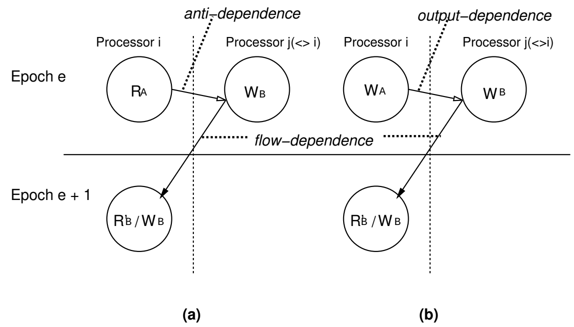

Figure 5: Stale access resulting from implicit dependences with multi-word lines. Variables A and B are assumed in the same cache line

Epoch counter overflow and two-phase invalidation The most significant bit (MSB) of the epoch counter is used 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 $2^{n-1} - 1$, where n is the number of bits in the $R_{counter}$. When this event occurs, all the cache entries whose MSB of the time-tag is the same as the new MSB of the current $R_{counter}$ are identified and invalidated by the reset hardware shown in Figure 6. This is necessary because, otherwise, we cannot distinguish the old entries whose MSB of its time-tag is the same as the MSB of current epoch counter from the new entries in the current epoch. When the MSB of the epoch counter switches from 1 to 0, the time-tag numbers of the old cache entries may be greater than the current epoch counter value. During this phase, to avoid getting negative offsets, the epoch counter is extended with an extra 1 bit (borrow bit). This will guarantee that the condition, Time-Tag $\geq$ ($R_{counter}$ - offset), be met for a cache hit for a Time-Read operation. The borrow bit is ignored during the phase when the MSB of the epoch counter is 1.

Since overflows occur in two phases with the MSB of $R_{counter}$ being 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 time-tag 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,

[4]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, offsets larger than the maximum need to be reset to the maximum by a compiler.

[5]Time-Read requires an offset in addition to a memory operand. To avoid adding an extra field, Time-Read operation can be implemented with a register operation followed by a regular memory operation. The first register operation stores the offset into a register and subtracts it from the epoch counter. The result is compared with the time-tag when the memory operation is performed.

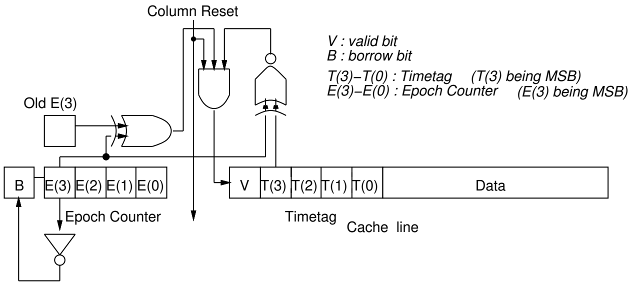

Figure 6: Reset hardware for two-phase invalidation.

this overflow mechanism can capture more locality on time-tag overflow than previous version or timestamp-based schemes. The number of phases can be easily extended to more than two if more significant bits are used to identify each phase.

3.3 Hardware overhead

Using the epoch counter as a system-wide version number can reduce the hardware complexity of the new scheme significantly compared to previous techniques. It eliminates the required version storage in the version control [5] and the timestamp-based scheme [12]. Our proposed scheme can also avoid flushing the entire cache when the epoch counter overflows by using its two-phase invalidation mechanism. This can be significant because it avoids using a large time tag to reduce the frequency of the time tag overflows. The hardware cost can thus be reduced substantially. Our simulations show that it can eliminate most of the unnecessary invalidations caused by the overflows using a small time tag.

Recently, several new schemes have been proposed to address the above problems [2, 14]. In these schemes, every memory instruction stores the version information as part of the cache state. To exploit n levels of task locality, those schemes require n bits of state information per cache word. This state information is used to invalidate the cache entries before a new version is created on a per-variable basis.

Compared to the those schemes, the scheme can capture more intertask locality with the same amount of cache state information. The scheme only requires $log_2 n$ bits to exploit n levels of intertask locality as opposed to n bits for the bit vector information used in the lifespan strategy [2] or Peir and So's scheme [14]. From our simulation results, 16 levels of intertask locality is usually enough to keep most of temporal locality. The intertask locality can be captured with only a 4-bit timetag in this scheme, while the previous schemes will require 16 bits of state information per word.

4 Compiler algorithms for reference marking

For correct coherence enforcement, the compiler needs to generate appropriate memory and cache management instructions. Potential stale read references need to be identified and issued as Time-Read operations. For such references, the compiler also must calculate the offset value, which is used to determine a cache hit at runtime. In this section, we present a compiler algorithm for automatically finding the potential stale accesses by examining the memory reference patterns in the source code.

4.1 Problem specification

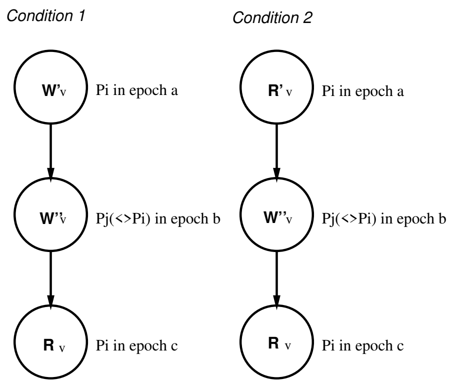

Figure 7: Stale access conditions.

For simplicity, let's consider a source program consisting of only doall loops. As shown in Figure 7, the following sequence of events creates a stale access: (1) a read/write to data V by the processor i in the epoch a (2) a write to V by another processor j ($\neq i$) in the epoch b (3) a read of V by the processor i in the epoch c. For doall loops, the following condition always holds: $a < b < c$. Since the processor assignment is unknown at compile time, we must assume that the references in each epoch can be issued by different processors. Therefore, we must find a sequence of events in a source program that consists of (1) a read or a write, (2) one or more epoch boundaries, (3) a write, (4) one or more epoch boundaries, and ends with (5) a read. The first read or write reference will create an initial cache copy of the data V in processor i, and the second write reference will create a new cache copy in processor j and makes the copy in processor i's cache stale. The sequence of events from (1) to (4) is called collectively an RDS (relaxed determining sequence) [4]. When an RDS of the variable v precedes a read of v with no reference to v in between, the read of v is considered a potential stale reference, and should be emitted as a Time-Read instruction. The number of epoch boundaries crossed between the epoch a and the epoch c is the offset value for the Time-Read. To simplify the RDS detection at compile time, we only look for write accesses in an RDS, which is (1) a write (2) one or more epoch boundaries (3) another write (4) one or more epoch boundaries. However, it may not cover the case when the first access to a variable is not preceded by any write to the variable in the same epoch. Such a read should be detected and treated as a write in our analysis.

4.2 Epoch flow graph

We consider the problem of identifying potential stale references in a data flow framework. Data flow analysis examines the flow of values through a program. It solves data flow problems by propagating information along the paths of a control flow graph. Because the epoch boundary information is essential in identifying potential stale references, we extend the flow graph used in a sequential program to include parallel constructs.

Definition 1: epoch flow graph Let a directed graph G = (V, E) be a control flow graph where V is a set of basic blocks, and E is a set of directed edges, representing the control flow between nodes in V. We define *epoch flow graph* G' = (V $\cup$ S, E') where E' = E - {e: e is the back edge from the end of a parallel loop to the beginning of the loop} - {e: e is the edge from the beginning of a parallel loop to the outside of the loop} + {e: e is the edge from the end of a parallel loop to the outside of the loop}. S is called the *start node* and is inserted at the beginning of the epoch flow graph. We divide the edges into two types. The edges into the beginning of a parallel loop and the edges out of a parallel loop are called *scheduling edges*. The scheduling edges convey the epoch boundary information. The remaining edges are called *control flow edges*.

In addition, the following definitions are used in this section.

- *Head Node*: A basic block which has an incoming *scheduling* edge. The start node is a special head node which does not have any incoming *scheduling* edge.
- *Tail Node*: A basic block which has an outgoing *scheduling* edge.
- *Epoch Level*: A subset L of an epoch flow graph G' that includes only a single head node B, and all the nodes and edges that have a directed path from B without crossing a scheduling edge. The set L is called the *epoch level from B*. There is an one-to-one relationship between head nodes and epoch levels. Multiple tail nodes can exist in L. A directed path from B to each tail node T is called an *epoch from B to T*.

Figure 8 shows a program example and its corresponding epoch flow graph. Note that a basic block can belong to more than one epoch. For example, in Figure 8, the node B1 can belong to the epoch from S to B1 (The epoch consists of nodes S, B0 and B1) and the epoch from B3 to B1 (The epoch consists of nodes B3, B1). This implies that the basic block B1 can belong to different epochs at runtime depending on the control flow path taken.

Given a source program and its epoch flow graph G, first, we mark the first read reference to a data item in each epoch as a target for potential stale reference. Second, using a flow analysis algorithm, the existence of an RDS for such references is determined along with the computation of offsets. If an RDS is found for the marked references, those references are emitted as Time-Read instructions.

4.3 Detection of upwardly-exposed references

Definition2: upwardly-exposed reference An *upwardly-exposed reference* is a read reference in an epoch which is not preceded by any other reference to the same data item in the epoch. Note that the definition is different from the term used in the definition-use chains of the standard flow analysis because a read following a read will not be marked as an upwardly-exposed reference in our definition.

Note that *only the upwardly-exposed reference* to a variable in an epoch can be a potential stale access. The following accesses in the epoch will not be stale since the cache copy should have been made up-to-date by the first read reference. If the cache copy has been replaced due to a cache conflict, a miss will occur and an up-to-date copy will again be fetched from memory. The rest of the memory references can thus be marked as regular reads or writes.

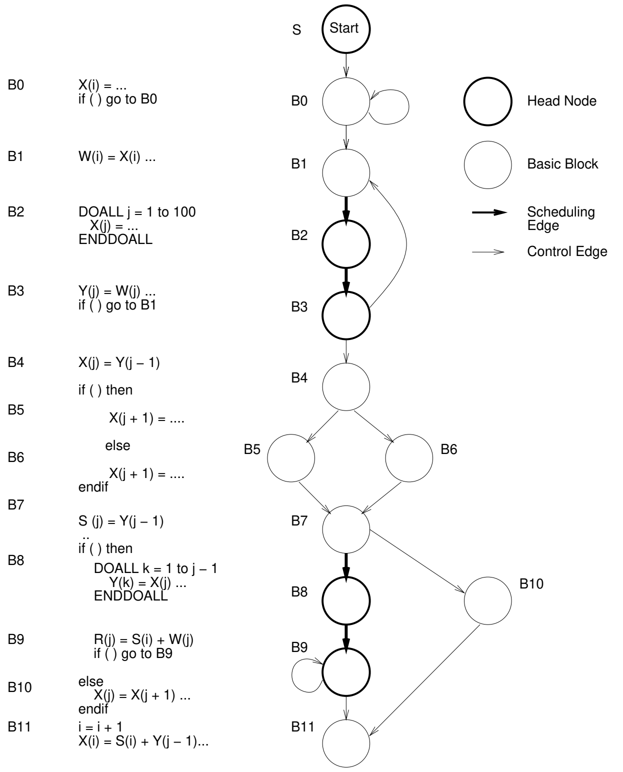

Figure 8: A program example and its epoch flow graph.

The detection of upwardly-exposed reference for a scalar variable can be done using variable name analysis. However, for array variables, treating an entire array as a single variable will mark all the read references to the array in an epoch as potential stale references. To refine reference marking for array references, we use the static single assignment form (SSA form) [8]. SSA is a representation of a program where each use of a variable is reached by exactly one single definition of the variable. It allows us to track the value of a variable by its name. By transforming a source program into its SSA form, we can treat different array reference expressions to an array as different symbolic variables.

The algorithm to find upwardly-exposed references in an epoch works as follows. First, it translates the original program to SSA form. Second, based on the preorder search of basic blocks in the epoch flow graphs, it marks the first read reference to each data item in each epoch level. For the program example in Figure 8, all read references except the read reference to Y(j-1) in B7 are marked as upwardly-exposed references.

```
INPUT :    Epoch Flow Graph and GEN(B) for each basic block B
           offset of definitions in GEN(B) is denoted collectively as offset(B)

OUTPUT:    VERSION(B) and STALE(B) for each basic block B

for each basic block B do
    VERSION(B) = {};
    STALE(B) = {};
    offset(B) = 0;
    IN(B) = {};
    OUT(B) = GEN(B);
end
while changes to STALE(B) occur for any basic block B do
    for each basic block B do
        for each incoming edge i and its corresponding predecessor P of B do
            if i is a scheduling edge then begin
               increment offsets of all the definitions in OUT(P)
            IN(B) = IN(B) U OUT(P);
            OUT(B) = IN(B) U GEN(B);
            VERSION(B) = {d: d is in OUT(B) and offset(d) > 0};
            if VERSION(B) includes at least two definitions for a variable x then
               STALE(B) = STALE(B) U {definition of x with minimum offset};
            STALE(B) = STALE(B) U STALE(P)
        end /* inner for */
    end /* outer for */
end /* while */
```

Figure 9: Simplified flow analysis algorithm for stale set computation.

4.4 Data flow analysis for stale access detection and offset calculation

In this section, we present a simplified version of the flow analysis algorithm [7]. The algorithm finds all the potentially stale references but it is conservative and generates superset of necessary RDS. The full discussion of the detail algorithm is beyond the scope of this paper and is described in another paper [7].

Definition3: version set and stale set A write to a variable v is called a *definition* of the variable v. Each definition, denoted as v_{offset}, is associated with an *offset*. Initially, an offset of 0 is assigned to a definition, and incremented when the definition crosses the scheduling edge. The offset represents the number of epoch boundaries crossed for the definition. Multiple definitions of a variable with the same offset are considered as a single definition. Therefore, multiple writes in an epoch is considered as a single definition of the variable. Because of the processor reassignments at each epoch boundary, a definition in an epoch can reach any following epoch without being *killed*. Therefore, the notion of *kill* used in the traditional reaching definition algorithm is unnecessary. A definition that has crossed at least an epoch boundary is called a *version*. The version denotes a distinct cache copy of the variable.

For each basic block B, we define the following sets.

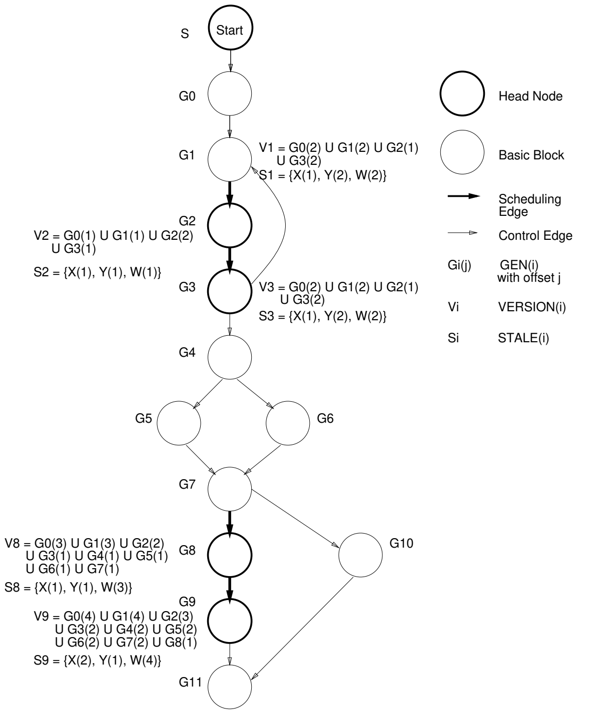

Figure 10: Result of the flow analysis algorithm. Gi(j) denotes $GEN(B)_j$. X(i) denotes a definition of variable X with an offset i.

- $GEN(B)_{offset}$ is a set of definitions created in B. Since all the definitions created in B are assigned the same offset, the offset can be denoted collectively for B as $GEN(B)_{offset}$.
- *IN(B)* is a set of definitions reaching the beginning of B, IN(B) $= \bigcup_P$ OUT(P) where P is a predecessor of B.
- *OUT(B)* a set of definitions reaching the end of B, OUT(B) = IN(B) $\cup$ $GEN(B)_{offset}$.
- *VERSION(B)* is a set of definitions reaching the beginning of B across at least one epoch boundary, VERSION(B) $= \{v_{offset}$: $v_{offset} \in$ IN(B) and offset $> 0\}$, VERSION(B) $\subseteq$ IN(B).
- *STALE(B)* is a set of definitions with the minimum offset when at least two versions reach the beginning of B, STALE(B) $= \{v_{offset}$: there exists at least two definitions of v in VERSION(B) and offset is the minimum of the offsets among the definitions$\}$.

Given an epoch flow graph G and $GEN(B)_{offset}$ for every basic block B in G, we want to compute the STALE set and the VERSION set. STALE(B) denotes a set of definitions when a possible RDS is detected at the entry of B. Figure 9 shows the iterative flow analysis algorithm to compute the STALE set and the VERSION set. If there are two versions reaching the same basic block B, the access to the variable can be potentially stale and should be included in the STALE set. Note that a single definition alone can also result in an RDS if there is a cycle including the definition and at least one epoch boundary. The flow analysis algorithm can find such situations since all the definitions are passed along the back edge of the cycle and the definitions with different offsets are treated as different definitions. In such a case, all the variables in the VERSION(B) belong to a stale set. Since the STALE set contains the version with the minimum offset, it guarantees the termination of the iterative algorithm.

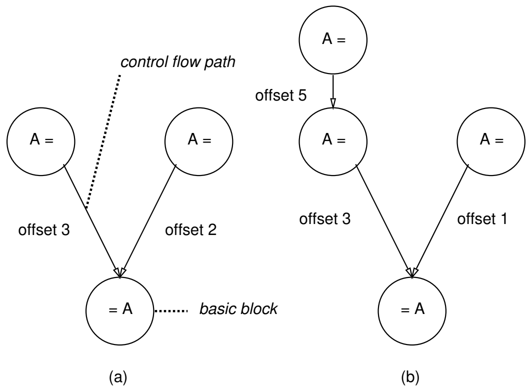

Figure 11: Conservative decision of flow information.

Figure 10 shows the result of our computation for the STALE set and the VERSION set from the program example in Figure 8.

The solutions to unknowns IN, OUT, VERSION and STALE are not necessarily unique and we want the smallest solution. However, the algorithm in Figure 9 is conservative in two aspects. First, if two versions reach a basic block from separate flow paths, then it will not result in an RDS because, at runtime, only a single version can be reached (see Figure 11a). However, the algorithm will detect such a case as an RDS. Second, in the algorithm in Figure 9, the minimum of all the offsets in VERSION set is selected as the offset for the RDS. However, when the version with the minimum offset does not cause the RDS, then the offset value is too conservative. For example in Figure 11b, although the offset for a read reference to A should be 3, the algorithm will find 1, which does not constitute an RDS, as an offset.

5 Conclusion

In this paper, we present a compiler-directed cache coherence algorithm with minimal hardware support. It can exploit both temporal and spatial locality across task boundaries. The scheme requires a small amount of hardware support and can be implemented efficiently on a microprocessor with minimal modifications. The uniform version numbering on a per-epoch basis eliminates the performance and implementation problems of the timestamp [12] or version control [5] schemes. It also allows us to support multi-word lines without losing spatial locality. We describe the hardware support, memory and cache management operations and the compiler marking algorithms needed to perform correct memory and cache operations. We also present a flow analysis algorithm which can identify potential stale references and generate appropriate memory instructions for the proposed scheme. Compared to the algorithm introduced in [4], our algorithm eliminates the graph construction needed for each variable for the RDS detection step [4], and integrates the RDS detection in the flow analysis. We are currently implementing these compiler algorithms in the Polaris parallelizing compiler [13].

References

[1] 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.

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

[3] Hoichi Cheong and Alex Veidenbaum. A Cache Coherence Scheme with Fast Selective Invalidation. *Proceedings of The 15th Annual International Symposium on Computer Architecture*, page 299, June 1988.

[4] Hoichi Cheong and Alexander V. 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.

[5] Hoichi Cheong and Alexander V. Veidenbaum. A Version Control Approach To Cache Coherence. *Proceedings of 1989 ACM/SIGARCH International Conference on Supercomputing*, June 1989.

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

[7] Lynn Choi and Pen-Chung Yew. A Compiler-Directed Cache Coherence Scheme with Improved Intertask Locality. *CSRD Technical Report No. 1325*, July, 1994.

[8] Ron Cytron, Jeanne Ferrante, and Barry K. Rosen. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[9] Ervan Darnell and Ken Kennedy. Cache Coherence Using Local Knowledge. *Proceedings Supercomputing'93*, pages 720–729, November 1993.

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

[11] 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:I–2–I–9, August 1992.

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

[13] 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.

[14] J. K. Peir, K. So, and J. H. Tang. Techniques to Enhance Cache Performance Across Parallel Program Sections. *Proceedings of the 1993 International Conference on Parallel Processing*, I, Architecture:I–12–I–19, August 1993.

[15] 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.