Life Span Strategy – A Compiler-Based Approach to Cache Coherence

Hoichi Cheong

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

Abstract

In this paper, a cache coherence strategy with a combined software and hardware approach is proposed for large-scale multiprocessor systems. The new strategy has the scalability advantages of existing software strategies and does not rely on shared hardware resources to maintain coherence. It exploits as much intra-task temporal locality as previously proposed low-cost, compiler–based strategies such as Simple Invalidation and Fast Selective Invalidation. With a small amount of additional hardware and a small set of cache management instructions, the new strategy preserves more inter-task-level temporal locality than these strategies. It is an economical alternative and has potential performance close to that of more elaborate strategies such as Version Control and Time Stamp. Also, the new strategy is easily extendable to include `Doacross` *loops.*

Keywords: **Compiler-based cache coherence, parallel task execution, inter-task-level temporal locality, simple invalidation, fast selective invalidation, version control, time-stamp approach, life span strategy, Doacross loop**.

1 Introduction

Compiler-based cache coherence strategies have been proposed by various researchers [1–8] to maintain cache coherence for multiprocessor computers with either private or partially shared caches or both. Cache coherence is important to preserve correct program execution if nonshared caches are used to hide the long access latency between the processor and the main memory. While compiler-based cache coherence strategies are not restricted by shared hardware or global communication overhead found in conventional hardware-based strategies (directory schemes [9–11] or bus schemes [12–16]) and are thus attractive solutions in large-scale multiprocessor systems, it has also been shown that compiler-based strategies can deliver performance comparable to hardware-based strategies in systems with even a moderate number of processors [17].

Compiler-based coherence strategies such as Simple (Indiscriminate) Invalidation [2] are unable to utilize temporal locality that extends across parallel code boundaries. (This locality will be referred to as "*inter-task-level temporal locality*" and will be discussed in the next section.) The Fast Selective Invalidation (FSI) [6], the Version Control [18], and the Time Stamp strategies [19] were proposed to allow better utilization of such a locality in order to improve performance. The failure to capture this locality and the resultant performance degradation were demonstrated by the cache ping-pong problem of the hardware-based strategies. In order to tackle the cache ping-pong problem, an algorithm [20] has been proposed to schedule a processor to execute tasks that access approximately the same set of variable elements (scalar variables or array elements). Such an algorithm will also benefit the FSI, the Version Control, and the Time Stamp strategies. Among these strategies, Version Control and Time Stamp have higher cost than FSI, but can utilize the most inter-task-level temporal locality preserved by task scheduling. On the other hand, the FSI strategy has a much lower hardware cost. These conditions motivate the search for a solution that can offer increasing utilization of this locality with incremental hardware costs while maintaining the simplicity of the compiler algorithm in these strategies.

In this paper, a new compiler-based strategy is introduced. The minimum implementation of this strategy, with little additional hardware to each private cache, can utilize more *inter-task-level temporal locality* than the FSI strategy. With the incremental addition of hardware when cost allows, it can be shown that the utilization of this locality can also be increased. The paper is organized as follows. In section 2, the assumptions are presented and existing compiler-based strategies are examined. In section 3, the problems of the existing strategies are described, and the objective of the new strategy is introduced. In section 4, the basic ideas of the new strategy are presented with some examples. The compiler algorithm to utilize the hardware for coherence control and preservation of *inter-task-level temporal locality* is discussed in section 5. Section 6 contains a correctness proof. The extended version of the strategy is presented in Section 7. An extension of the strategy to include `Doacross` loops is given in Section 8.

2 Existing Compiler-Based Strategies

2.1 Assumptions and models

In this section, some related existing compiler-based strategies are described briefly, and the assumptions of a parallel execution model are introduced. Then, the problem of coherence maintenance and the solutions provided by the existing strategies are discussed using such a parallel execution model.

For the existing and the new compiler-based strategies, we assume a multiprocessor system with a shared global memory and a weakly ordered access model. The global memory modules and

ICS '92-7/92/D.C., USA
© 1992 ACM 0-89791-485-6/92/0007/0139...$1.50

the processors can either be connected as in a dance-hall organization or can be distributed among the processors. However, a logically shared memory is assumed. For simplicity, in the following discussion only one level of private cache memories with the write-through policy is covered, even though some strategies, such as Version Control, can be applied to multilevel caches with different degrees of sharing [21]. Synchronization variables are assumed to be noncacheable in these strategies. A cache coherence block [22] of one cache word is assumed.

Numerical programs with `Doall`-type loop-level parallelism are assumed to be executed on systems with the above system model. `Doall` loops are parallel loops with no cross iteration dependences [23]. Iterations are scheduled to each processor, and synchronizations are carried out at the beginning and the end of each parallel loop.

To facilitate the following discussion, the execution of a program is modeled as an execution task graph which is a directed graph. Each node in the graph is a task that is either a sequential portion of a program or a group of iterations scheduled to a processor. Each task is executed by one processor. A directed edge from one task to another represents all the dependences between the two tasks. Figure 1 shows a portion of an execution task graph and the code corresponding to each task. A Start task and an End task are used to represent the beginning and the end of the program. Each task also belongs to a task level that denotes the set of tasks at the same distance from the Start task, measured by the length (the number of edges) of the longest directed path. By the definition of the execution task graph, a processor can only execute tasks in a nondecreasing order of level numbers. Multiple tasks on the same level can be executed in parallel. By our model of `Doall` loops, the sets of variable elements written by parallel tasks are mutually exclusive. For convenience, we can draw a line between two adjacent levels as the level boundary. Synchronization operations are required at each level boundary, and therefore, these level boundaries correspond to the beginnings and/or the ends of `Doall` loops and can be detected at compile-time by a compiler.

With the execution graph, the problem of memory incoherence can be easily described. For a processor P_j with a copy of X[j] in its cache, a write to X[j] on a task level L by another processor will turn the existing cache copy of X[j] in P_j *stale* (out-of-date). When processor P_j executes a task on a subsequent level, using the stale copy of X[j] will produce an incorrect result. Therefore, an access on a level and a write to the same variable element on a subsequent level can potentially cause an access to use a stale copy of that element on a later level. This sequence of an access followed by a write (to the same element) on a different task level will be referred to as the *condition for stale accesses* at later levels.

2.2 Existing strategies

The existing compiler-based strategies and their limitations can be described using the execution graph. Within tasks on the same level, cache words can be accessed as in a uniprocessor cache as long as they already contain the up-to-date values at the start of a level. (It is impossible for a task to use a value produced from another task on the same level in our parallel execution model.) The first fetch of a variable element after the beginning of each level will be referred to as the *new fetch* of the variable element on that level. If a new fetch of a variable element detects a stale value in the cache word, a cache miss can be issued and the up-to-date value from the global memory can be loaded into the cache word. Compiler-based strategies rely on the compiler to keep track of which new fetches may access stale cache words. When a new fetch accesses an up-to-date cache word, the hit is credited to inter-task-level temporal locality if the cache word is not loaded in earlier in the same level. All hits by fetches other than the new fetch are the result of intra-task temporal locality (same as conventional temporal locality) or spatial locality. It should be noticed that the inter-task-level temporal locality depends on how the tasks are scheduled to processors, while intra-task-level locality does not.

The Simple Invalidation strategy takes a conservative view in that at every level N, the cache copy of any shared read-write variable element can be potentially stale. Therefore, the compiler inserts an `Invalidate` instruction at code location corresponding to a level boundary, and each processor executes the `Invalidate` instruction to invalidate simultaneously the entire private cache before crossing a level boundary. Then, a new fetch will miss on the invalidated cache word and will load from the global memory copy which is kept up-to-date by the write-through policy. The advantage of this strategy is that the `Invalidate` instruction can be a simple reset operation on the bits, each corresponding to a cache word, and invalidation is thus very fast. The disadvantage of this strategy is that inter-task-level temporal locality cannot be exploited.

The FSI strategy is based on the observation that not all new fetches are preceded by the condition for stale accesses. Therefore, not all new fetches need to miss and to load from global memory. The FSI strategy uses the compiler to identify these fetches. These fetches are marked `Cache Read`. All fetches from read-only variables or private variables are marked `Cache Read`; so are the fetches not preceded by the condition for stale accesses. The compiler backend will generate an appropriate fetch according to the marking. A `Cache Read` fetch is treated as a conventional memory access by the cache controller because it will not access potentially stale cache copies.

The rest of the fetches, which may access stale cache copies, are marked `Memory Read`. Different codes to distinguish these two different types of fetches need to be supported by the instruction set and by the compiler. An extra status bit, called the Change bit, is associated with each cache word. The cache controller checks the Change bit for each `Memory Read`. If the Change bit is set, a miss is issued. If it is not set, the cache copy is guaranteed to be up-to-date. An `Invalidate` instruction is executed by each processor to set all the Change bits when the processor crosses each level boundary. This forces all new fetches that are `Memory Read` to use the up-to-date copy from global memory. The Change bit of a cache word is reset upon a load due to a miss or a write operation. A reset Change bit preserves the intra-task level temporal locality of the cache word.

For shared read-write variables, this strategy can exploit *inter-task-level temporal locality* for `Cache Read` accesses. This is an improvement over the Simple Invalidation strategy. However, once a fetch to a shared read-write variable is preceded by the condition for stale accesses, all fetches in succeeding levels

are marked as `Memory Read`, and their inter-task-level temporal locality cannot be exploited. Limited information about task scheduling forces the compiler to mark the fetches `Memory Read` in these cases. The Version Control and the Time-Stamp strategies will further relax this limitation. (Only the Version Control strategy will be described below, but the Version Control and Time-Stamp strategies are based on the same idea.)

The FSI strategy has better performance than the Simple Invalidation strategy and schemes similar to SI that were proposed by different researchers [3, 5]. However, a recent paper [24] showed that the reference-marking [5] has better simulated performance than the FSI strategy. Verification with the authors confirmed that, in their simulation of the FSI strategy, each processor executes the `Invalidate` operation per iteration instead of per level boundary as proposed. This misunderstanding of the FSI strategy destroyed a large part of the intra-task-level locality which makes both the SI and FSI strategies useful.

The Version Control strategy considers that the writes to a variable on each level create a new version of a variable in the execution. Each processor will keep a current version number (*cvn*) for each shared read-write variable in each program execution (each array is treated as one variable). The version numbers are kept in a local memory accessed in parallel with the cache accesses. Each cache word will have an extra field called the *birth version number* (*bvn*). At the time the cache word is loaded or written, the value of the *cvn* (for read misses) or *cvn + 1* (for writes) will be written into the *bvn*. At the end of each level, the processor increments the *cvn* for each variable that might have been written in that level. Code to increment the *cvn* can be easily generated once the level boundary is detected, and the writes to variables are identified in a level. A cache miss is issued when the *cvn* of a variable accessed is larger than the *bvn* of the cache word accessed. By keeping track of the versions, stale copies are avoided.

3 Exploiting Inter-task-level Locality

3.1 Inter-task-level temporal locality

The following metric of estimating inter-task-level temporal locality will be used in the discussion. Only variables that are accessed on more than one level in the task execution graph have nonzero inter-task-level temporal locality. Let us suppose that there are M levels in a task execution graph (from level 1 to M), and there are K levels containing accesses to X. The sequence of values l_X^i

$$1 \le \left\{ l_X^1, l_X^2, \ldots l_X^K \right\} \le M$$

represents the level numbers that correspond to these K levels. A useful quantity, *next inter-task-level access distance* of a variable, can be defined as the number of levels from a task level containing accesses to the variable to the next task level containing accesses to the same variable. The next inter-task-level access distance of the variable X at each of the K levels is:

$$d\left(l_X^j\right) = \begin{cases} l_X^{j+1} - l_X^j & ; K > j \ge 1 \\ undefined & ; j = K \end{cases}$$

Then, over those levels that contain accesses to X, we can make use of a quantity $\overline{\ell}_X$ which measures the average inter-task-level temporal locality and is defined as:

$$\overline{\ell}_X = \frac{K-1}{\left(\sum_{j=1}^{K-1} d(l_X^j)\right)} = \frac{K-1}{l_X^K - l_X^1} \quad , for \;\; K > 1$$

A variable that it is accessed on every task level between level l_X^1 and l_X^K is called a variable with *full inter-task-level temporal locality* of accesses over these levels ($\overline{\ell}_X$ is 1). When the distances between levels with accesses to X are large, the variable has lower inter-task-level temporal locality. Notice also that the chance that a cache copy will be replaced increases with such distances. Therefore, for levels l_X^1 through l_X^{K-1}, the reciprocal of the next inter-task-level access distance

$$\ell\left(l_X^j\right) = \frac{1}{d(l_X^j)}$$

is called the immediate inter-task-level temporal locality for the variable X on that level. When a variable X always has immediate inter-task-level temporal locality of value 1, the variable has the best possible immediate inter-task-level temporal locality.

The quantities above can be estimated from flow analysis or measured in program execution through tracing. These metrics are useful for a first-cut evaluation for strategies to exploit inter-task-level temporal locality.

3.2 Inefficient use of the available locality

In the existing compiler-based cache coherence strategies, it is quite costly to take advantage of even the best available inter-task-level temporal locality. For example, let us examine the following scenario. Suppose on level N task i, t_i^N, writes array X[i], and on each of the levels N+1 through N+k, there are tasks t_i^{N+1} and t_i^{N+k} respectively that read X[i], which should have the value written by t_i^N. Further, suppose that in each task from t_i^{N+1} to t_i^{N+k}, the new fetch will load the up-to-date copy from the global memory if the cache copy is determined to be stale. If it can be detected that processor P_j has executed t_i^{N+i}, when P_j is scheduled to execute t_i^{N+j}, where $0 \le i < j \le k$, the cache copy of X[i] in P_j will contain the up-to-date value written on level N. However, in a nonstatic scheduling environment, the above scheduling condition is expensive to guarantee and to detect; therefore, when P_j is scheduled to execute t_i^{N+j}, the cache copy of X[i] in its cache may still contain a value written on a level prior to level N. In this case, if P_j is scheduled to execute any of the t_i^{N+1}, ... , t_i^{N+k} on the respective level, the new fetch in each level is forced to load the up-to-date value from the global memory. This is why the FSI strategy forces each new fetch to a variable element to load from the global memory once the compiler detects the condition for stale accesses.

The Version Control and the Time Stamp strategies provide a solution, but it is not as economical as one would hope. The version number associated with each cache copy will indicate whether the cache copy contains the value by the most recent write. But the version number is expensive to implement when space is precious such as in the case of on-chip caches. Furthermore, for variables

that have very little immediate inter-task-level temporal locality (long next inter-task-level access distance), the Version Control and the Time Stamp strategies are even less cost effective. If a small version number field is implemented, overflows will become more frequent, and the time spent to have all the processors reset the version numbers will be unbearable.

4 The Life Span Strategy

The Life Span strategy targets the highest possible inter-task-level temporal locality. With little additional hardware (one more status bit per cache word than the FSI strategy) in its basic implementation, it can outperform the FSI strategy in utilizing inter-task-level temporal locality. It can be shown that the Life Span strategy can inexpensively exploit the *full inter-task-level temporal locality* of accesses to variable elements. At the same time, it can exploit intra-task-level temporal locality as well as any other strategy. In this section, we describe the basic ideas of the strategy. Let us first discuss the architectural support required for the strategy and illustrate the algorithm to maintain coherence with an example.

4.1 Architectural support

4.1.1 Cache Status bits

Each cache coherence unit is associated with three status bits: Valid, Change, and Stale.

Valid bit (V). This is the same as the valid bit in a conventional cache. If set (logic value 1), the cache item is present; otherwise the cache item is not present. Storing into the cache as a result of a fetch from the memory or a write to the memory will set the valid bit.

Change bit (C). The Change bit of a cache copy is examined by the cache controller upon a `Memory_read` or a `Memory_read_reset_stale` fetch (defined below). A reset Change bit indicates that the associated cache copy is up-to-date. If the Change bit is set, a cache miss is issued to load the global memory copy. The Change bit is reset when a value is stored in the cache copy by a write or by a fetch from the global memory on a miss. The setting of the Change bit is a possible outcome of the `Invalidate` operation discussed in the processor/system support section, and the Change bit is individually addressable along with the associated cache copy and is writable.

Stale bit (S). The Stale bit of a cache copy is used to determine the new value of the Change bit before the start of each task level. The Stale bit is reset by the `Memory_read_reset_stale` instruction (defined in the Processor/System Support section) and the `Write` instruction. When a value is stored into the cache copy by a write or by a cache miss due to a `Memory_read_reset_stale` fetch, the Stale bit is reset. It is set by a `Memory_read` (see below) instruction, a `Write_set_stale` instruction, or an `Invalidate` operation inserted before the next task level boundary. The reset Stale bit indicates that the cache copy has acquired an up-to-date value through a write or a miss in the current level. This information is used to guide the fetches in the next task level. A Stale bit is also individually addressable along with the associated cache copy and is writable.

4.1.2 Processor/System support

The processor is required to issue three kinds of memory fetches. The operations of these fetches and the responses from the cache controller are described below. Also, the support to `Write` and `Invalidate` will be discussed.

`Cache_read`. `Cache_read` is the same as the conventional memory fetch operation. The cache controller only tests the valid bit to determine a cache miss or cache hit.

`Memory_read`. Upon receiving a `Memory_read` fetch, the controller examines the value of the Change bit. If the Change bit is set, a miss is issued, the global memory copy is fetched, and the Change bit is reset when the copy comes back from the global memory. If the Change bit is reset, the fetch is serviced as a conventional cache access that is concerned only with the status of the valid bit. Also, the Stale bit of the addressed word is set in both cases. For multiple words prefetched upon a miss, the Change bit of each prefetched word is reset. `Memory_read` is used only in the extension with `Doacross` loops

`Memory_read_reset_stale`. This is almost the same as `Memory_read` except that the Stale bit of the addressed word is reset. For multiple words prefetched upon a miss, only the word addressed by the fetch has its Stale bit reset. When a `Memory_read_reset_stale` hits, the Stale bit of the addressed word is reset. This resets the Stale bit of a prefetched word when it is actually referenced.

`Write`. The `Write` operation resets the Change bit, resets the Stale bit, and sets the Valid bit of the word addressed by the write operation. The write-through policy is assumed even though it is not strictly required.

`Write_set_stale`. Same as `Write` except that the Stale bit is set in the operation (used only in the extension with `Doacross` loops).

System support is also needed to perform the `Invalidate` operation. This support can be in the form of an additional processor instruction or as a write to an output port to trigger the operation. The goal of the `Invalidation` operation is to have each Change bit value reflect the stale status of the associated cache copy in the next task level. An `Invalidate` is carried out once at the end of a task level by each processor. For the cache copies that might contain stale values, their Change bits should be set. Recall that a set Change bit value will cause a miss and a load from the global memory in the next task level. Therefore, the set Change bit effectively invalidates the cache copies to be accessed by a `Memory_read` or a `Memory_read_reset_stale` on the next level. For the cache copies that are written or for those whose up-to-date values are used during the task level, the Change bit should be reset for the next task level. The ability to reset the Change bit for these cache copies is the major distinction between the Life Span strategy and the FSI strategy. To accomplish this, `Invalidate` uses the value of the Stale bit to determine the value of the Change bit for the next task level. The `Invalidate` operation affects both the Stale bit and the Change bit and is carried out in parallel on the entire cache.

`Invalidate` The Stale bit value is stored into the Change bit, and Stale bit is set for all cache words.

4.2 Notation

The following notation is used in the discussions below:

INV - `Invalidate`
R(X[i]) - read X[i]
W(X[i]) - write X[i]
WSS(X[i]) - `write_set_stale` X[i]
CR(X[i]) - `Cache_read` from X[i]
MR(X[i]) - `Memory_read` from X[i]
MRRS(X[i]) - `Memory_read_reset_stale` from X[i]
t_i^N - Task i on task level N.

4.3 Example

Let us assume for the time being that the compiler has correctly generated code for the different kinds of fetches. The basic ideas for generating such code are discussed in the next section. In Figure 1, tasks in four task levels are shown, and the code in each task is listed in a column next to the task graph. The read/write operations to an array variable X are shown in the code. For clarity, let us assume that task i on each level accesses array element X[i]. Also, the corresponding fetches and writes to X[i] as generated by the compiler are listed adjacent to the task code.

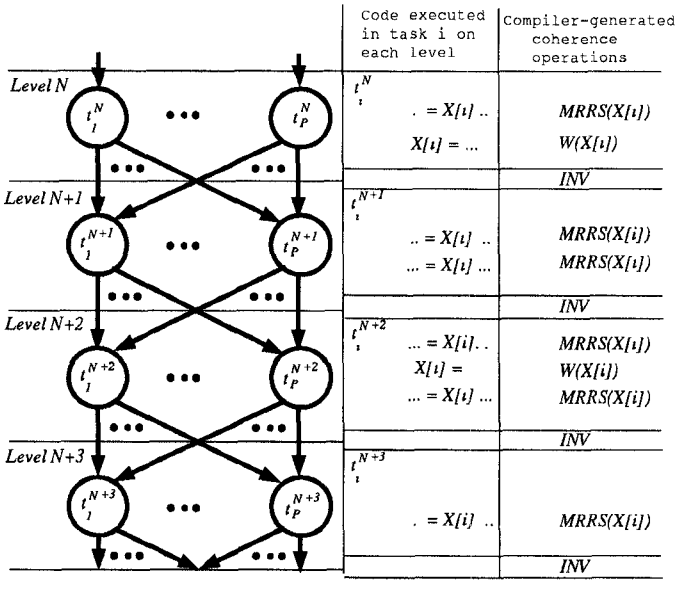

Figure 1. Example of Life Span strategy: task graph and code.

The ability of the Life Span strategy to maintain cache coherence can be illustrated with the following task execution. Let processor P_I be scheduled to tasks t_1^N, t_2^{N+1}, t_2^{N+2}, and t_1^{N+3} on task levels N, N+1, N+2, and N+3 respectively. Table 1 shows the status bits of X[1] and X[2] in P_I's cache and also the cache responses. For clarity of presentation, let us assume a cache coherence block of one word and a prefetched line of one word. The status bits affected by the operations will be represented by two numbers separated by a slash (old value/new value), and the unaffected status bits are represented by a single number, such as the status bits for X[2] on rows 1, 2, and 11, and those for X[1] on rows 4, 5, 7, 8, and 9. The `Invalidate` operation affects the status bits of all cache copies, as shown on rows 3, 6, 10, and 12.

On level N, P_I executes task t_1^N and it misses on X[1]. The change of the Stale bit on row 1 records the result of the MRRS. On the other hand, because P_I does not access its cache copy of X[2], the Stale bit of X[2] does not change. After task t_1^N, the processor executes the `Invalidate` operation (row 3) on its entire cache — the Change bit takes the value of the Stale bit, and the Stale bit becomes 1. The `Invalidate` operation is depicted as a separate task that is executed once by each processor before going on to the next level. The resultant zero Change bit of X[1] indicates its up-to-date status, but the Change bit of X[2] (value 1) indicates a stale copy. In our `Doall` program model, the execution of task t_2^N on level N by another processor will very likely write the up-to-date copy of X[2] in a different cache. Therefore, the Change bits of X[1] and X[2] in P_I's cache correctly signal the status of the cache copies. The resultant Change bits on row 10 can be explained similarly with the roles of task 1 (t_1^{N+2}) and task 2 (t_1^{N+2}) reversed.

When P_I executes task t_2^{N+1} on level N+1, the cache controller generates a cache miss (see row 4) on the access according to the Change bit. Also, since the fetch is an MRRS, the Stale bit is reset. The zero Stale bit indicates that the copy is an up-to-date one from the global memory. And the copy of X[2] will be known to be up-to-date on task level N+2. Therefore, the new fetch by P_I on level N+2 is a hit (row 7).

On level N+1 and N+2, intra-task-level temporal locality is also preserved by the zero Change bit. On rows 5 and 9, the fetches result in a cache hit due to the zero Change bits previously reset in the current level.

To see the advantage of the Life Span strategy, notice that the fetch from X[1] on row 7 is also a new fetch. The FSI strategy will generate a `Memory Read` for this access. Without the Stale

Table 1. Example of Life Span strategy: case 1.

Task no	X[1] cache status bits		X[2] cache status bits		Operations that affect status bits	Cache response	Row
	S	C	S	C			
t_I^N	1/0	1/0	1	1	MRRS(X[1])	miss	1
	0/0	0/0	1	1	W(X[1])		2
INV Task	0/1	0/0	1/1	1/1	INV		3
t_2^{N+1}	1	0	1/0	1/0	MRRS(X[2])	miss	4
	1	0	0/0	0/0	MRRS(X[2])	hit	5
INV Task	1/1	0/1	0/1	0/0	INV		6
t_2^{N+2}	1	1	1/0	0/0	MRRS(X[2])	hit	7
	1	1	0/0	0/0	W(X[2])		8
	1	1	0/0	0/0	MRRS(X[2])	hit	9
INV Task	1/1	1/1	0/1	0/0	INV		10
t_I^{N+3}	1/0	1/0	1	0	MRRS(X[1])	miss	11
INV Task	0/1	0/0	1/1	0/1	INV		12

bit, this fetch will find the Change bit set. (The Change bits are always set to 1 by INV in the FSI strategy.) Therefore, a cache miss will be issued. In the Life Span strategy, this access is a hit.

For the best-case performance of the Life Span strategy, let us suppose that processor P_1 happens to be scheduled to execute tasks t_1^N, t_1^{N+1}, t_1^{N+2}, t_1^{N+3}, on levels N, N+1, N+2, and N+3, respectively. As in Table 2, the new fetches on rows 4, 7, and 11 are all hits due to the reset Change bit. This would otherwise be impossible in the FSI strategy. Only in the Version Control strategy can these new fetches be hits in this particular processor schedule.

Notice that the `Invalidates` for task levels N+1 (row 6) and N+3 (row 12) are not necessary for array X, given that every processor executes the `Invalidate` on level N (row 3) and level N+2 (row 10). However, because an `Invalidate` is carried out on the entire cache instead of only on the cache copies of variables written, it could have been inserted for other variables presumably written on levels N+1 and N+3. For task levels that contain no write to any shared global variable, an `Invalidate` should not be required.

The Life Span strategy with one Stale bit per cache word can keep the up-to-date status of cache copies for at least one extra level. Therefore, such an implementation is called the One-Level Life Span strategy. If a variable element is accessed every level for a number of consecutive levels, the resetting of the Stale bit in each level can sustain the up-to-date status of the cache copy throughout these levels. For variable elements with full inter-task-level temporal locality, this is a remarkable cost-saving advantage over the fixed-length version numbers or time stamps.

Table 2. Example of Life Span strategy: case 2.

Task no.	X[1] cache status bits		Operations that affect status bits	Cache response	Row
	S	C			
t_1^N	1/0	1/0	MRRS(X[1])	miss	1
	0/0	0/0	W(X[1])		2
INV Task	0/1	0/0	INV		3
t_1^{N+1}	1/0	0/0	MRRS(X[1])	hit	4
	0/0	0/0	MRRS(X[1])	hit	5
INV Task	0/1	0/0	INV		6
t_1^{N+2}	1/0	0/0	MRRS(X[1])	hit	7
	0/0	0/0	W(X[1])		8
	0/0	0/0	MRRS(X[1])	hit	9
INV Task	0/1	0/0	INV		10
t_1^{N+3}	1/0	0/0	MRRS(X[1])	hit	11
INV Task	0/1	0/0	INV		12

5 Compiler Support

The compiler algorithm to identify and generate code for different kinds of fetches is based on the one introduced in the FSI strategy. A flow algorithm is used in the FSI strategy to identify `Cache Reads` from `Memory Reads`. The mechanism of the original algorithm will be briefly described, and the extension to further divide `Memory Read` fetches into `Memory_read` and `Memory_read_reset_stale` fetches will be discussed. (For a more detailed description of the original algorithm, refer to [7].) After the different types of fetches are identified and marked, it should be straightforward for the compiler backend to generate appropriate code for them.

5.1 `Cache Read` identification

The compiler algorithm requires that the program has already been parallelized by a restructurer. The flow algorithm tries to detect, for each variable X (scalar or array), the sequential occurrence of {read(X) or write(X), ... , level boundary, ... , write(X), ... , level boundary} before any read(X). This sequence is the condition for stale accesses; the level boundary is either the beginning or the end of a `Doall` loop. Notice that this test does not require checking of index expressions. To a shared variable, any read(X) that is preceded by such a sequence is identified as `Memory Read` and marked for later code generation; any read(X) not preceded by such a sequence is identified and marked as `Cache Read`.

5.2 `Invalidate` insertion

Next, `Invalidates` must be inserted for the proper task level. Recall that the purpose of the `Invalidate` operation is to make potentially stale cache copies known to the cache controllers on the next task level. The only condition that can cause cache copies of a variable stale is to have a write to the variable in a task level. Therefore, the algorithm needs to identify a write to a variable in order to insert an `Invalidate` before the next level boundary. However, because an `Invalidate` affects all cache copies (this is more efficient than seeking out all cache copies of variables written during run-time), a write to any shared variable on a task level will cause the compiler to insert an `Invalidate` before the next level boundary.

A boundary level is identified whenever the beginning or the end of a `Doall` loop is found. A back-to-back `Doall-end` and `Doall-begin` pair of two adjacent `Doall` loops is considered to form only one boundary. Also, multiple `Doall-ends` (or `Doall-begins`) of perfectly nested loops constitute one boundary. At run time, when a processor attempts to take a task from the task pool, it should check whether a task level boundary has been crossed. If it has, the processor will execute the inserted `Invalidate` operation.

5.3 `MRRS` identification

In our parallel execution model (i.e., no data dependence exists across `Doall` loop iterations) only `MRRS` is needed. The function of `MRRS` implies that cache copies written or used by a processor will stay up-to-date in the next level. In this case, all

fetches marked Memory Reads in the FSI compiler algorithm can be marked MRRS. In parallel program execution models that include data dependence across iteration, Memory Reads identified by the FSI algorithm will be further divided into MRRS and Memory_reads and this will be discussed in an extension in a later section.

6 Correctness Proof

For the One-Level Life Span strategy to function correctly, a fetch should deliver only up-to-date copies of variable elements to the processor. In the following sketch of a proof, the caches are assumed to start out clean, and the Stale bits start out all set.

Cache misses. If a fetch misses, the copy from the global memory must be up-to-date due to the write-through policy and the weakly ordered consistency model.

Cache hits. A Cache_read fetch does not have preceding accesses that match the pattern of the condition for stale accesses; that is, either no writes to the same variables occur in a previous task level, or previous writes only modified the variable element that had no existing cache copy. Therefore, the cache copies accessed by a Cache_read fetch must be up-to-date.

For MRRS fetches, a hit occurs when the Change bit is found reset. It is sufficient to show that when a cache copy with a reset Change bit is accessed, the cache copy must be up-to-date. There are two cases in which the Change bit is reset in a task level. Either the Change bit is reset due to a write or a miss in the current task level or has already been reset at the beginning of the current task level. In the first case, a write or a load from global memory due to a read miss was the cause, and the resultant cache copy must be up-to-date.

Let us call the reset Change bit in the second case "*initially reset Change bit*" and present the following proof. The program model of Doall loops is assumed.

Hypothesis: A cache copy with an initially reset Change bit, is always up-to-date.

Proof: Let a task level M contain a fetch that accesses a cache copy with an initially reset Change bit. There exists a task level L, prior to level M, with the following properties. To task level M, level L is the closest level that contains a write to the variable of the cache copy accessed in M. Level L exists because fetches other than Cache_reads must be preceded by a Write due to the condition for stale accesses. Because there is no Write between level L and level M, the cache copies written on level L must still be up-to-date.

From the compiler and system supports, Stale bits reset during a task level (by an MRRS or a Write) are always set again before the next level by an Invalidate; existing cache copies of the variable produced prior to level L must have their Stale bits set before the start of level L. Therefore, the Invalidate between level L and level L+1 must have set the Change bits (using the Stale bits) of the variable's cache copies not written by the processors on level L. A subsequent access to any potentially stale copy will result in a miss. A cache copy with a set Change bit after level L could only have its Change bit reset by a miss. All subsequent hits are either hits on the copies produced on level L or on copies loaded from global memory between level L and level M. Therefore, the cache copy with an initially reset Change bit is an up-to-date copy. □

A cache copy with an initially reset Change bit or a Change bit reset in a task level has been shown always to contain an up-to-date copy when accessed; therefore, the proof is complete.

7 Extension — N-Level Life Span Strategy

The One-Level Life Span strategy works best for variable elements whose next inter-task-level access distance is one, especially for variable elements that are accessed in every one of a sequence of adjacent levels. Variable elements with next inter-task-level access distance longer than one can also be exploited provided an Invalidate is not needed for every task level. However, we assume the worst case, in which an Invalidate is needed for each task level. In the worst case, the strategy cannot preserve inter-task-level temporal locality for variables with inter-task-level access distance larger than one because the basic implementation uses only one Stale bit per cache word. If the variable element is accessed every J task levels (J > 1), at the level in which the variable element is not accessed, the Stale bit will not be reset during the task level. As a result, the Change bit will be set by an Invalidate operation possibly inserted at the end of the level. Therefore, in the next task level, even though the cache copy may be perfectly up-to-date, a Memory_read_reset_stale will cause a miss because of the set Change bit. Of course, for a large J, the opportunity that the cache copy of the variable element will be replaced is also large. However, for a smaller J, it may be worthwhile to keep the up-to-date status of a cache copy for J levels.

It should be pointed out that the Version Control and the Time Stamp strategies keep the up-to-date status until the variable is modified again. By doing that, the cost of the version control is paid even for variable elements that have very low intermediate inter-task-level temporal locality. Also, with the Stale bits in the Life Span strategy, there is no execution time overhead in overflowing a finite version number or time stamp.

To exploit the high intermediate inter-task-level temporal locality of variable elements, additional architectural and compiler support are needed. The basic idea is to increase the number of Stale bits from 1 to N. Each cache copy is associated with $\{S_{N-1} \ldots S_0\}$ Stale bits which are addressable and writable as a unit. There exist different ways to support the manipulation of the Stale bits. In the following list of requirements in the N-Level Life Span strategy, only one way is shown.

Processor/System Support. Only the additional functionalities required on top of the One-Level Life Span strategy are discussed.

Memory_read. The Memory_read type fetch will set all the Stale bits of the addressed cache copy.

Memory_read_reset_stale (MRRS). The MRRS requires an extra field that specifies the number of low-order Stale bits to reset. The compiler will generate the field value as the minimum number of Invalidates in subsequent levels through which the cache copy needs to keep its up-to-date status. Instead of resetting only one Stale bit associated with the addressed cache word, the specified number of Stale bits in the field starting from

S_0 will be reset in the N-Level implementation. The notation used is `MRRS(X,f)`, where f is the extra field.

`Write`. The same extra field in the `MRRS` fetch is required, and the Stale bits of the addressed cache word are reset in the same manner. The notation `W(X,f)` is used. `WSS` sets the Stale bits of a cache copy.

`Invalidate`. An `Invalidate` affects the status bits of every cache word as follows:

$\{C = S0;\ S_j = S_{j+1}$ for $0 \le j < N\text{-}1,\ S_{N-1} = 1\}$.

The shift decreases by one the number of reset stale bits which are needed to keep the remaining `Invalidates` from rendering the cache copy stale.

Extended Compiler Support. In addition to the functions required in the basic implementation, the compiler needs to identify the minimum number of `Invalidates` from a `Write` or an `MRRS` of a variable to its next `Write` access on a different task level on a flow path. This number is used by the `Write` or the `MRRS` to reset the number of Stale bits.

For each `Write` or `MRRS`, flow analysis is used to find the `Writes`, to the same variable, reachable from the `Write` or the `MRRS`. As reachability is established, the minimum number `Invalidate` inserted on each path is determined. Then, the minimum of these numbers is selected among all the possible flow paths. If no such reachable `Writes` are found, the maximum number of bits allowed in the field is used instead.

The chosen number will be used for the extra field in `MRRS(X,f)` and `W(X,f)` to reset the appropriate number of Stale bits of a cache copy. The reset Stale bits ensure that a cache copy will maintain its up-to-date status beyond such a number of `Invalidates`. The minimum is used because any additional reset Stale bit can erroneously extend the up-to-date status of a cache copy even though it might have indeed become stale.

8 Extension to Include Doacross

In parallel execution models that include data dependence across parallel loop iterations such as those in `Doacross` loops, the combination of proper synchronization, `Memory_read`, and `Write_set_stale`, will ensure cache coherence. In such cases, the reference-marking algorithm further divides the fetches into `Memory_read` and `MRRS` fetches. This extension focuses locally on the iteration body of a `Doacross` loop which constitutes a task in our execution model. In the following discussion, we assume the One-Level Life Span strategy first and that *the* `Doacross` *loops are automatically restructured from sequential loops.*[1]

The fetches and writes that are either the source or the sink of a dependence arc across iterations [23] are of concern. For clarity, let's call them sink-fetches, source-fetches, sink-writes, and source-writes. *Proper synchronization is assumed to guarantee that the dependence is satisfied. Furthermore, a source-write in a cross-iteration flow-dependence (read-after-write) is required to deposit the written value into the global memory before the necessary synchronization is carried out to satisfy the dependence.*

[1] This assumption excludes loops with backward dependences from a later iteration to a previous one.

8.1 Source-fetches

For a source-fetch, the dependence is an anti-dependence (write-after-read) with a sink-write in a later iteration in the original sequential loop. The variable element of the cache copy used will be written later by another processor in the same task level. In such a case, the cache copy fetched is made stale by this later write and should not be used beyond the current task level. Therefore, the Stale bit should be set instead of reset when such a copy is accessed. For this reason, such fetches must be marked `Memory_read`. The compiler support to identify them is easy provided the `Doacross` loops and the cross-iteration dependence are identified by the restructuring front end. Even though most anti-dependences can be eliminated by restructuring compilers [25], those involving complicated index functions will likely remain. The simple index expressions in the example below are used for the purpose of clarity.

The example in Figure 2 shows a simplified view of such a `Doacross` loop. Notice, by the Change bits, that X[2] in P_1's cache will be treated as stale after the task level whereas the up-to-date copy in P_2's cache will be recognized as a fresh copy.

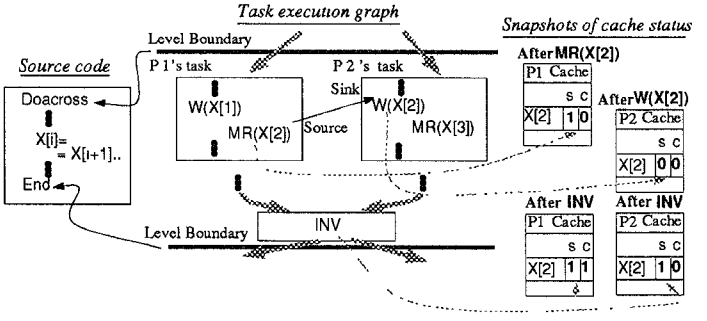

Figure 2. `Doacross` example: Source-fetches

8.2 Sink-fetches

For a sink-fetch, the dependence is a flow-dependence (read-after-write) with a source-write in a previous iteration in the original sequential loop. The fetch must use a new value produced by another processor in the same task level, and the copy used can also stay up-to-date beyond the current task level boundary. However, because the copy used must be produced by another processor on the same task level, the existing cache copy at the beginning of the current task level must be avoided. This requires that all existing copies of the same variable element must have set Change bits in the current level.

Therefore, neither `Writes` nor fetches to the same variable on the last task level should reset the Stale bits of the cache copies. With write accesses marked `Write_set_stale` and fetches `Memory_read` in the last task level, a sink-fetch to the same variable element is guaranteed to access the up-to-date value produced on the current level by another processor.

In order to mark correctly an access immediately before a test level corresponding to a `Doacross` loop, compile-time analysis is needed to look into the `Doacross` loop body to find sink-fetches to the same variable. If found, write and fetches in the last task level will be marked as above.

It remains unanswered how to mark the sink-fetch in question. Because the sink-fetch uses the most up-to-date copy produced on the same task level by another processor, the copy loaded in the cache by the sink-fetch is up-to-date. If the fetches in the next task level will use these up-to-date copies, their up-to-date status should be maintained by the zero Change bit in the next task level. Hence, the sink-fetch in question should be marked MRRS. However, in case sink-fetches to the same variable exist in the next task level, the copies produced on the current level must not be allowed to stay up-to-date into the next task level. Therefore, the sink-fetch in question should be marked `Memory_read` which will result in the stale status for such copies in the next task level.

For a fetch that is both a sink and a source, it is marked `Memory_read`.

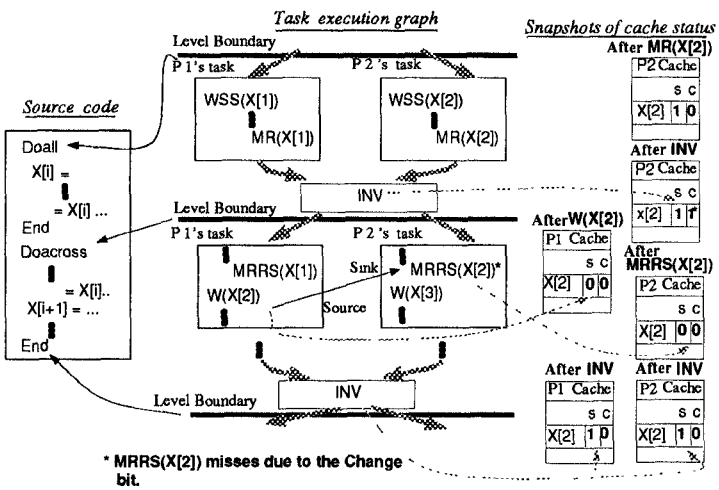

Figure 3. `Doacross` example: Sink-fetches

Figure 3 illustrates an example of `Doacross` with sink fetches. The snapshot of cache status shows the Stale and the Change bit corresponding to cache copy X[2]. Due to the sink-fetch on X[i], the write and the fetch to X in the last task level (corresponding to the `Doall` loop) are marked MR and `Write_set_stale` (WSS) which result in the set Change bit of X[2] in P_2's cache after the middle level boundary. Therefore, the fetch by P_2 from X[2] in the `Doacross loop` will result in a miss. Our assumption of cross-iteration flow-dependence guarantees that P_1 has written the new value of X[2] into the global memory before the synchronization operation. The miss by P_2 will then be satisfied by the up-to-date copy of X[2] in the global memory. In the illustration, we assume that the next task level after the `Doacross loop` contains no sink-fetch of X in a cross-iteration flow-dependence. Therefore, the write and the fetch of X in the `Doacross loop` are marked `Write` and MRRS respectively.

8.3 Sink-writes and source-writes

A sink-write and a source-write constitute an output-dependence across iterations. By proper synchronization, the sink-write must overwrite the value produced by the source-write. Therefore, in such a case, the cache copy produced by the source write must not be used in the next task level, and the source write should be marked `Write_set_stale`. The setting of the Stale bit will make sure that the set Change bit of the cache copy will force a miss in the next task level. The sink-write in a cross-iteration output-dependence can be marked either as `Write_set_stale` or `Write` for reasons similar to the case of sink-fetches. For a write access that is both a sink or a source in cross-iteration output-dependence, it is marked `Write_set_stale`. The writes in cross-iteration anti- or flow-dependence are marked `Write_set_stale` only if there exists a sink-fetch of the same variable in a cross-iteration flow-dependence in the next task level.

8.4 N-Level Life Span strategy

To support `Doacross` loops with the N-Level Life Span strategy, the only modification of the above is the method to select the number for the extra field in `MRRS(X,f)` and `W(X,f)`. This is needed to cover the sink-fetch case. Recall that in the case without `Doacross`, the number selected is the minimum number of `Invalidates` between the access in question and any of its reachable write accesses in a different task level.

In the case with `Doacross`, for any of the above flow paths that yield the same minimum number of `Invalidates`, the algorithm should check for sink-fetches to the same variable in the task level containing the reachable writes. If such a sink-fetch exists, the number used for the extra field should be the minimum number minus one. This ensures that the sink-fetch will not use any cache copy extended from a previous task level. The number is guaranteed to be nonnegative. Also notice that `MRRS(X,0)` and `Write(X,0)` are equivalent to `Memory_read` and `Write_set_stale` respectively.

9 Summary and Future Work

This paper introduces a new compiler-based cache coherence strategy that offers attractive cost-performance potentials. This coherence strategy requires no run-time shared resource which may become a potential bottleneck and limit scalability. Avoiding the cost of implementing version numbers and the overhead of version number overflow in the Version Control and Time Stamp strategies, this strategy can exploit the inter-task-level temporal locality which has eluded low-cost compiler-based cache coherence strategies. The addition of a few Stale bits can potentially utilize inter-task-level temporal locality of short next inter-task-level access distances, which accounts for most of the hits from this kind of locality. Although a large field of Stale bits intuitively may not result in substantially large additional improvement in a finite cache with replacement problem, future work will be required to determine a practical number of Stale bits.

Successful utilization of inter-task-level temporal locality depends on many factors such as next inter-level access distances, granularity of tasks, and the scheduling algorithm. The first factor varies among programs. Granularity of tasks can inversely affect the significance of the inter-task-level temporal locality. In the absence of algorithms to exploit such a locality, coarse grain tasks with good spatial locality tend to reduce the number of misses by new fetches, while fine grain tasks tend to have more misses.

Also, if tasks that access approximately the same set of variable elements are scheduled on the same processor, it will enhance the potential of cache coherence strategies that aim at exploiting inter-task-level temporal locality. In the worst case, in which tasks with very few accesses in common are scheduled on the same processor, none of these strategies will perform much better than the Simple Invalidation strategy. With this in mind, a low-cost coherence strategy that can exploit inter-task-level temporal locality is important, and scheduling strategies to exploit this locality deserves future research efforts. Previous simulations [6, 18, 20, 26] showed different results on the amount of performance gain from inter-task-level temporal locality. Whether such gain is significant will require further study on a broader spectrum of programs.

10 Acknowledgment

The author would like to thank Alex Veidenbaum, David Sehr, Sam Midkiff, Ervan Darnell, and the referees for their valuable suggestions. This work is supported in part by the U.S. Department of Energy under Grant No. US DOE DE-FG02-85ER25001, and NSF MIP 89-20891.

References

[1] A. J. Smith, "CPU cache consistency with software support and using 'One Time Identifiers'," *Proceedings Pacific Computer Communications Symposium*, pp. 153–161, October 1985.

[2] A. Veidenbaum, "A compiler-assisted cache coherence solution for multiprocessors," *Proceedings 1986 International Conference on Parallel Processing*, pp. 1029–1036, August 1986.

[3] K. P. McAuliffe, "Analysis of cache memories in highly parallel systems," Tech. Rep. No. 269, Ph.D. dissertation, Courant Institute of Mathematical Sciences, NYU, 1986.

[4] R. L. Lee, "The effectiveness of caches and data prefetch buffers in large-scale shared memory multiprocessors," Tech. Rep. No. 670, Ph.D. dissertation, University of Illinois, Urbana, IL, August 1987.

[5] R. Lee, P. Yew, and D. Lawrie, "Multiprocessor cache design considerations," *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 253–262, June 1987.

[6] H. Cheong and A. Veidenbaum, "A cache coherence scheme with fast selective invalidation," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, p. 299, June 1988.

[7] H. Cheong and A. V. Veidenbaum, "Stale data detection and coherence enforcement using flow analysis," *Proceedings of the 1988 International Conference on Parallel Processing*, vol. I, pp. 138–145, August 1988.

[8] R. Cytron, S. Karlovsky, and K. P. McAuliffe, "Automatic management of programmable caches," *Proceedings the 1988 International Conference on Parallel Processing*, vol. II, pp. 229–238, August 1988.

[9] C. K. Tang, "Cache system design in the tightly coupled multiprocessor system," *Proceedings NCC*, vol. 45, pp. 749–753, 1976.

[10] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Transations on Computers*, vol. C-27, no. 12, pp. 1112–1118, December, 1978.

[11] J. Archibald and J.-L. Baer, "An economical solution to the cache coherence problem," *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 355–362, June 5-7, 1984.

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

[13] E. McCreight, "The dragon computer system: An early overview," tech. rep., Xerox Corp., September 1984.

[14] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 348–354, June 1984.

[15] L. Rudolph and Z. Segall, "Dynamic decentralized cache schemes for MIMD parallel processors," *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 340–347, June 1984.

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

[17] Y.-C. Chen and A. V. Veidenbaum, "A software coherence scheme with the assistance of directories," *Proceedings 1989 International Conference on Supercomputing*, June 1991.

[18] H. Cheong and A. Veidenbaum, "A version control approach to cache coherence," *Proceedings International Conference of Supercomputing 89*, pp. 322–330, June 1989.

[19] S. L. Min and J.-L. Baer, "A timestamp-based cache coherence scheme," *Proceedings 1989 International Conference on Parallel Processing, I, Architecture:23-32*, August 1989.

[20] J. Fang and M. Lu, "A solution of cache ping-pong problem in RISC based parallel processing systems," *Proceedings 1991 International Conference On Parallel Processing, I, Architecture:238-245*, August 1991.

[21] H. Cheong, "Compiler-directed cache coherence strategies for large-scale shared-memory multiprocessor systems," Tech. Rep. CSRD No. 953, Ph.D. dissertation, University of Illinois at Urbana-Champaign, January 1990.

[22] J. R. Goodman, "Coherency for multiprocessor virtual address caches," *Proceedings of the Second International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS II)*, pp. 72–81, 1987.

[23] U. Banerjee, "Data dependence in ordinary programs," Tech. Rep. No. 76-837, M.S. thesis, University of Illinois at Urbana-Champaign, November 1976.

[24] S. L. Min and J.-L. Baer, "Design and analysis of a scalable cache coherence scheme basaed on clocks and timestamps," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 1, January 1992.

[25] M. Wolfe, "Optimizing compilers for supercomputers," Tech. Rep. No. UIUCDCS-R-82-1105, Ph.D. dissertation, University of Illinois, Urbana, IL, October 1982.

[26] J.-K. Peir, K. So, and J.-H. Tang, "Inter-section locality of shared data in parallel programs," *Proceedings 1991 International Conference On Parallel Processing, I, Architecture:278-286*, August 1991.