Cache Coherence Using Local Knowledge *

Ervan Darnell † Ken Kennedy

Computer Science Department, Rice University, Houston, TX 77251-1892

Abstract

Typically, commercially available shared memory machines have addressed the cache coherence problem with hardware strategies based on global inter-cache communication. However, global communication limits its scalability and efficiency.

"Local knowledge" coherence strategies, which avoid global communication at run-time, offer better scalability, at the cost of some additional cache misses. The most effective local knowledge strategies described in the literature are those based on generation timestamps (TS).

We propose a new strategy, TS1, that requires less extra storage than TS, only one extra bit per cache line, and can produce more cache hits by exploiting sophisticated compiler analysis. TS1 handles common synchronization paradigms including `DOALL`, `DOACROSS`, *and critical sections.*

Early results show TS1 is, worst case, slightly slower than TS. Best case, TS1's flexibility allows for significant improvements.

1 Introduction

Data race free programs executing on a shared memory multiprocessor are expected to have the same semantics as if they were executing on a sequential processor. For this to be the case, memory must appear sequentially consistent [10]. Caches on shared memory multiprocessors must have some global knowledge about executing programs otherwise they could fail to preserve sequential consistency by retaining stale values.

Most approaches to the cache coherence problem have focused on hardware mechanisms to maintain coherence. Unfortunately, the overhead of maintaining coherence in hardware can be high; scaling systems based on hardware coherence is a difficult problem [15]. Snoopy cache strategies, which monitor some common bus, are now in common use for small scale systems [16, 18]; however, snoopy strategies are problematic for large-scale machines because such machines cannot be based on a single, central broadcast medium for lack of sufficient bandwidth. Directory strategies [2, 8, 11, 19], in which a directory entry associated with each memory location (or cache line) indicates which processors have cached values for that location, seem more promising for large-scale systems. However, directories can require large amounts of additional storage and directory maintenance operations may substantially increase network traffic.

As an alternative to using a hardware mechanism that supports global communication between caches, a compiler could perform global analysis and augment the code with cache control directives to maintain coherence. This approach does not hinder the scalability of the machine. However, since compiler analysis must be conservative, some valid values will be unnecessarily removed from cache with this approach, thus reducing hit rates.

In this paper, we propose a local knowledge only coherence strategy, TS1 (Time Stamping with 1 bit), that, for a particular granularity of compiler analysis, achieves the best hit rate that any such strategy can at a cost of one additional bit per cache line, sufficient logic to set, reset, or copy this bit to the *valid* bit, and a fast invalidate. We compare TS1 to previous local knowledge only coherence strategies. We show that time-stamping and TS1 achieve the same effect with naive compiler support but that TS1 can more readily utilize improved compiler analysis.

Section 2 provides an overview of some of the terminology we use. Section 3 discusses some of the better previous local strategies. Section 3.5 shows that time stamping strategies provide optimal hit ratios for a given granularity of compiler analysis. Section 4 describes our new strategy for achieving the same effect as time-stamping at lower hardware cost. It also discusses the impact of the accuracy of compiler analysis. Section 5 gives some preliminary results comparing TS and TS1. Section 6 discusses extensions to other types of synchronization. Section 7 concludes and discusses possible future directions.

*This work was supported in part by the National Science Foundation under Cooperative Agreement CCR-9120008 through its research contracts with the Center for Research on Parallel Computation at Rice University.

†corresponding author: ervan@cs.rice.edu

```
DOALL I
  A(I)=...        A(2) write allocated on processor 2
ENDDO

DOALL I
  A(I+1)=...      A(2) write allocated on processor 1
ENDDO

DOALL I
  B(I)=A(I)+1     Access stale A(2) on processor 2
ENDDO
```

Figure 1: Example of stale access in the absence of coherence control

2 Definitions and framework

Fork-join programs are composed of a series of *epochs*[4]. Each *epoch* consists of one or more *instances* which run in parallel. Each epoch is either a (fork-join) parallel loop with no internal synchronization, e.g. a Fortran **DOALL**, or a serial region between parallel loops. Serial regions can be nested serial loops and/or those parts of serial loops enclosing parallel loops. A serial region is an *epoch* with one *instance*.

The instances (iterations) of a parallel loop are scheduled on processors at run time. The compiler must assume that a given instance could be executed on any processor. All coherence strategies known to us make this assumption. If scheduling is known at compile time, a different set issues arises and different techniques are applicable.

We regard non-unit cache line size aliasing as an orthogonal problem. Our discussion assumes that the cache line size is one word.

Staleness occurs when a location is accessed on processor p_i, written subsequently on processor $p_j, j \neq i$, and read again still later on processor p_i (without its cache being updated to reflect the new value). For example, in figure 1, if processor p_i gets iteration i, the reference to `A(2)` in the last epoch would be stale (without coherence). Any additional references that happen between the first and second epochs will not change the situation. If `A(2)` were written on p_i between the second and third epochs (not shown in the figure) it would again be valid (on p_i) without coherence control.

Numerous previous authors have tried to capture this notion in various analytical ways [4, 6, 9]. Here we simply note the dynamic behavior that causes staleness without addressing its detection at compile time.

Coherence is maintained by making sure that values are communicated between caches when necessary. Values are updated from cache to main memory either by using write-thru cache or write-back cache trigged by synchronization events. Values are moved from main memory to cache on demand when they are not found in the cache. Stale values are removed from caches at the correct time so that subsequent reads will load the new values. This removal may be done by anything from explicit operating system calls that the compiler inserts to the hardware failing to register a hit because of some combination of bits.

2.1 Semantics of fork-join

The semantics of fork-join, e.g. **DOALL**, parallelism is that the fork-join loop be data race free; there can be no carried dependence, i.e. a value cannot be written in one instance (iteration) and read on another within the same epoch (**DOALL**).

A value can be read on several processors in the same epoch so long as none write it. The lack of carried dependence might be proven by the compiler or asserted by the programmer. In either case, it has the following important implication: If a value, x, is accessed in epoch e on p_i it will be coherent in epoch $e+1$ on p_i without any coherence control or communication with main memory. This is true because x being accessed on p_i during e means that it could not have been written on p_j during e. By the definition of staleness, x is still coherent in epoch $e+1$ on p_i.

The same value being read on different processors during epoch e does not present a problem because it cannot be written on any processor during epoch e. These properties are trivially preserved by serial regions.

2.2 Local versus global knowledge

To avoid staleness, each cache must account for what is written by every processor, including those to which it is not directly connected. This set of writes can be exactly the locations written or any approximating superset. If this global knowledge of what is written is shared at run time, we would characterize it as a *global strategy*. Global strategies are usually thought of as hardware strategies, e.g. snoopy caches [16, 18] and directory based caches [2, 11, 19]. Other global strategies, e.g. OS level page strategies [17], are software strategies.

Cache misses have four causes: initial loading, cache size, cache organization (e.g. associativity), and invalidation to preserve coherence between processors (sharing induced). Coherence strategies are concerned only with the last category, sharing induced misses. No currently existing global strategy causes a logically unnecessary sharing miss. They are in that sense optimal. The drawback of global strategies is the that scalability is impaired by the cost of maintaining global knowledge at run time.

If no global knowledge is shared at run time, then coherence must rely on locally collected knowledge plus whatever global knowledge was collected at compile time. Previous local knowledge strategies have been referred to in the literature as software [4, 6] or hardware [14] strategies depending on whether *most* of the work was done in software or hardware. We consider all of these strategies to be similar and refer to them collectively as *local strategies*.

A local strategy will likely never result in a globally optimal hit ratio for a processor because some useful runtime knowledge will be unavailable. The effectiveness of local strategies can vary widely depending on the program being run and the strategy being used The principal advantage of local strategies is that they are scalable because no global knowledge need be com-

```
DOALL I=1,N
   A(I)=...
ENDDO

DOALL I=1,N
   IF (c1) A(I+1)=...
ENDDO

DOALL I=1,N
   B(I)=A(I)+1
ENDDO
```

Figure 2: Global versus Local coherence

```
DOALL I=1,N
   A(I)=...
ENDDO

DOALL I=1,N
   A(I)=A(I)+...
ENDDO

DOALL I=1,N
   B(I)=A(I)+1
ENDDO
```

Figure 3: Dynamic versus static coherence

municated at run time to maintain coherence. They rely on what *could* happen not what *does*. We use *could* happen to mean that the compiler cannot disprove it.

The fundamental limitation of local strategies is that if an *instance* (task) could write a value (and it cannot be determined for certain at compile time), other *instances* must assume that it has. A task need make no assumptions about what it itself does because sufficient marking bits can be used locally to make whatever determinations are useful regardless of the incompleteness of compiler analysis.

Figure 2 gives an example of where a local strategy would fail to achieve the same hit ratio as a global strategy. If condition `c1` is always false (for a given execution) but is not analyzable by the compiler, all potential reuse of cached values of `A` between the first and third epochs will be lost using a local strategy but preserved using a global strategy.

2.3 Dynamic versus static coherence

Local strategies are either static or dynamic. Static local strategies decide which cache lines to invalidate and when that invalidation should occur, using only compile time knowledge. In contrast, dynamic strategies can use run time information as well; which data actually gets invalidated depends on the actual execution path of the program. This comes in two forms, knowing (part of) the processor schedule and intra-instance control flow. Dynamic strategies take advantage of reuse that exists only because of a particular run time schedule. Static strategies must assume worst case scheduling (though they can still exploit some inter epoch reuse).

Dynamic strategies strictly improve on static strategies. Any static strategy can be changed to a dynamic strategy that will not cause any more sharing misses and will usually do much better. The tradeoff is that dynamic strategies require some additional hardware support to handle marking bits. Static strategies require no hardware support other than the ability to invalidate cache lines under software control. Static strategies can be used on some existing machines, such as the BBN TC2000[6, 7].

Figure 3 shows an example for which static strategies are inherently inefficient. The value of `A` written in the first epoch cannot be allowed to reach the third epoch. Since the second epoch (with the write) *might* have a different schedule than the first, an invalidate must remove the value from cache that was loaded in the first epoch. This will cause either the read in the second epoch or the third epoch to miss (depending on where the invalidate occurs). But if the schedules for all epochs are actually the same at run time this would be unnecessary. A dynamic strategy could recognize this and preserve reuse in both the second and third epochs.

3 Previous approaches

We briefly survey some previous local strategies in order to give credit to previous authors in this area and to show what improvement TS1 makes. Knowing the different approaches helps for understanding the core issues of local strategies and not being distracted by often disparate implementation approaches. Those aspects of previous strategies not directly concerned with coherence are covered in section 6.3.

3.1 Fast Selective Invalidation

One of the first strategies was Fast Selective Invalidate (FSI) [4]. FSI determines at compile time which references access shared variables that might have been previously written. These are designated *memory reads*. For *memory reads* to be cache hits they must be found in the cache and have a special *epoch* bit set (originally called a change bit). Accesses to shared variables set the *epoch* bit. All of the *epoch* bits are reset at every epoch boundary. With this strategy, no shared value crosses an epoch boundary in cache, ensuring that caches are coherent. The penalty is that no inter-epoch reuse is preserved for shared variables. FSI is a static strategy.

3.2 Life Span Strategy

Life Span Strategy (LSS) [3] is an improvement over FSI. Instead of resetting the *epoch* bit after every epoch, the *epoch* bit for a given cache line is reset

after the end of the next epoch. This is implemented with an extra bit in each line, the stale bit, that marks the passage of one epoch boundary. Thus any referenced value is preserved into next epoch. This is valid by DOALL semantics as previously noted (section 2.1). LSS is a dynamic strategy.

The LSS paper describes an extension to preserve a value in cache for any fixed number of epochs. The number of epochs it survives must be determined at compile time. Its maximum value is the shortest path (in number of epochs) that could be taken before another write. The actual path between writes at run time could be much longer. The count is stored in unary so that updating the count can be done with a shift and checking for validity can be done by examining the last bit. Values which are not referenced will eventually be removed from cache. If, in practice, values stay in cache for only a few epochs due to cache size limitations, a small number of bits can be used for the extended LSS at no great cost. Additional bits would not help because the values would already be evicted before the count runs out.

3.3 Parallel Explicit Invalidation

Parallel Explicit Invalidation (PEI) [12] works by combining an invalidate with each write instruction. Writing an element in an array invalidates everything in the array except for the element itself. This achieves coherence because anything written on a different processor will be removed from the cache of this processor before the next epoch (sufficient dummy writes are added to make sure this invariant is maintained in the presence of uncertain control flow and serial epochs). The PEI strategy preserves inter-epoch reuse by leaving the value written on this processor in cache. PEI is a dynamic strategy.

The implementation uses a bit mask to control the region invalidated. This requires that write instructions have enough additional bits to contain the mask. It does, however, allow for essentially constant time invalidation. It also allows for many special cases to be handled with more precision (assuming the compiler analysis is sufficient). In some instances, it can improve on the hit ratio time-stamping would achieve. However, in general, it fails to preserve intra-epoch reuse when there are multiple references to the same array in one instance. Those values that are lost to intra-epoch reuse are also lost to inter-epoch reuse. The use of an *epoch* bit alone would not suffice to prevent this. For PEI to achieve good results, arrays, or in the worst case, each dimension of an array, must occupy an amount of memory equal to a power of two.

3.4 Time Stamping

Time stamping (TS) strategies [4, 14] are more effective at preserving reuse than any of the previously mentioned strategies. For a given quality of compiler analysis, it is impossible to achieve a better hit ratio with any other local strategy. The trade-off is that they require several extra bits per cache line, extra bits per memory access instruction, several extra counters per processor, and extra logic in the cache controller. Time stamping is a dynamic strategy.

In time stamping, there is a counter (referred to as a clock) for each array which tracks the number of epochs in which the array was possibly written. Each processor has a copy of all of the clocks. At the *end* of an epoch, each processor increments its clocks for each array that might have been written during that epoch on *any* processor. This requires no global communication. The clock value for an array represents the last epoch where the array could have been written.

Each cache line has a time stamp. When a value is accessed, its time stamp is set to what the current clock value for its array will be in the next epoch. Example, if the current clock value for array `A` is 5, a write to `A(1)` is loaded into cache with a time stamp of 6 because the clock will be incremented at the end of this epoch. A read of `A(1)` in a loop with no write would set the time stamp to 5 because the clock will not be incremented at the end of the epoch.

A value becomes stale when its time stamp is less than the corresponding clock. If the time stamp equals the clock value, there has been no write since the last access to the cache line. By DOALL semantics (section 2.1) this also holds for the epoch where the value is written. For an epoch after a write, the cache line will contain its prior time stamp value, but the clock will have been incremented. The cache hardware will find that time stamp $<$ clock, conclude the value is stale, and issue a miss.

Both previous time stamping strategies operate on the whole array level. This is not necessary. It would be possible for time stamping to operate on the section level. However, it would require a separate clock for each section of each array. This would not only require extra clocks but extra bits in the instruction word to specify which clock was relevant.

There is a peculiar limitation to TS. The clocks can overflow. When that happens all cache lines which depend on that clock must be invalidated. This is the same problem extended LSS suffers. Time stamping uses binary counters and the impact is much less. Time stamping ages its cache lines by incrementing clocks on the processor while leaving the cache lines unchanged. Extended LSS ages its cache lines by decrementing counters in the cache line. This distinction will prove useful as explained in section 4.

3.5 A unified view of previous approaches

Despite very different implementations, all of these approaches are variants of the same essential strategy. During epoch e_w location x (some array element) is written on processor p_j; at the end of epoch e_w, processor $p_i, j \neq i$, invalidates location x in its cache Processor p_i might not know exactly what location x is. It approximates with the smallest set of locations sure to contain x.

FSI and LSS make the pessimistic assumption that x could be anywhere. TS assumes all of the array which x is in could have been written. PEI uses the best available compiler analysis, which is at least as good as the same array analysis of TS. Extended

LSS also uses the best available analysis to determine where x could be, but it unions the results over all paths causing it to invalidate more. TS and PEI are concerned only with the run time (inter epoch) path and are in this respect superior to extended LSS.

Dynamic and static strategies both must make the same estimate about how large the set is that encompasses x. Static schemes must also estimate the processor schedule, which means they must assume the worst case. Dynamic schemes, however, know part of the schedule, the part that occurs on the local processor. If x is accessed on p_i in epoch e_w, a dynamic scheme knows that p_j never actually wrote x (section 2.1) regardless of the compile time analysis. It can then avoid the invalidation of x. LSS, TS, and PEI all take advantage of this.

After the end of epoch e_w, if x was not referenced on p_i during e_w, p_i has no run time information about x and it must rely entirely on the compiler's analysis for whether or not p_j might write x in e_w. If the compiler then indicates that p_j appears to write x, a local strategy must invalidate x on p_i before the next read of x on p_i (if the next reference to x on p_i is a write, then it does not matter whether or not x was invalidated). If a strategy invalidates *only* when these two conditions are met (x not referenced on p_i and apparently written on p_j), it is optimal in the sense that no local strategy, with the same granularity of compiler analysis, could have a better hit rate. This is how time-stamping behaves. It is an optimal local strategy. PEI is not optimal because it invalidates before it knows that x is not referenced on p_i. LSS is optimal only in the trivial sense that it uses a know-nothing compiler.

In practice, this utilization of **DOALL** semantics can make a dramatic difference because it captures reuse when subsequent loops have the same schedule and same reference pattern, for instance a **DOALL** inside of a serial loop will likely meet this condition. Deliberate attempts to increase loop affinity [13] will further improve the benefit of dynamic strategies.

4 One-Bit Time Stamping

Even though both proposed time-stamping strategies [4, 14] are hit-rate optimal local strategies, they require substantial additional hardware. TS1 achieves the same optimal hit ratio with fewer special bits per cache line and no special bits per instruction word. It also avoids the need for special hardware to load and compare the proper time stamp in a cache line. But, it does require a more sophisticated invalidate.

4.1 Hardware support

TS1 requires a *valid* bit per cache line and an additional bit, the *epoch* bit. In TS1, caches set the *epoch* bit on any reference to that line (read, write, hit, or miss). At the end of a given epoch, e, a special instruction resets the *epoch* bit for every line in the cache. We assume that the cache implementation can do this in $O(1)$ time by having every cache line respond in parallel. Since all of the *epoch* bits were reset on entry to epoch e from the end of epoch $e-1$, the *epoch* bit reflects which cache lines have been accessed during epoch e.

By the assumed semantics of **DOALL** loops (section 2.1), any cache line with its *epoch* bit set in epoch e can be left in cache for epoch $e+1$ without causing a stale access. We use this observation in defining a special *invalidate* that operates optimistically. When a particular cache line is the object of an invalidate, it is actually invalidated only if the *epoch* bit is reset, otherwise it remains valid and in cache, i.e. the invalidate copies the *epoch* bit to the *valid* bit.

4.2 Implementation of the invalidate

There are several choices for the actual implementation of the invalidate that trade-off hardware cost for run time efficiency.

A slow but inexpensive implementation would be to have a low level invalidate instruction which could invalidate either a particular line or a particular page. The high level invalidate would then loop over the proper range of pages and lines. Even though this would take $O(|section|)$, acceptable performance could still be achieved. Previously, we examined the efficiency of this kind of invalidate for a static strategy [7].

A faster, but more complex, invalidate could work by using a bit mask to determine which addresses to invalidate. With only '=' comparators and no extra storage, a section could be invalidated in $O(\log(|section|))$ time. Special layouts and strides could reduce this further. This is similar to what PEI does.

Other authors have proposed $O(1)$ time invalidation implementations which work by accessing cache row and column addresses [1].

4.3 Software support

To determine what to invalidate, TS1 uses compile time analysis to determine what is written for each epoch. The compiler makes its best estimate that is sure to include every address actually written. The main task of this analysis is to determine which parts of shared arrays are written. A naive analysis could simply note which arrays appear on the left hand side of an assignment and then conclude that every element of any such array is modified. More sophisticated analysis could try to determine which sections of arrays are actually modified. For every section (or whole array) that is modified, the compiler inserts an invalidate for that range of addresses at the end of the epoch being analyzed. Since schedules are not known, the same set of invalidates is used for every processor.

At run time, for each epoch, some accesses occur, setting the *epoch* bits, then the invalidates are executed as the next to last instruction, removing soon to be stale values, and finally all of the *epoch* bits are reset in preparation for the next epoch.

For a value, x, to be stale for epoch e_r on processor

Operation	Applies to	Bit Assignments	
		Valid Bit	*Epoch* Bit
Read	word	1	1
Write	word	1	1
Invalidate	section	*Epoch* Bit	-
end of epoch	all of cache	-	0

Table 1: Effect of operations on TS1 control bits

State	TS	TS1	
	Time Stamp	Epoch Bit	Valid Bit
just accessed	= clock[+1]	= 1	= 1
not yet stale	= clock	= 0	= 1
stale	< clock	= 0	= 0

Table 2: Possible 'ages' of a cache line

p_i, it must have been written during epoch $e_w, e_w < e_r$ on $p_j, j \neq i$ and have been in p_i's cache on epoch $e_w - 1$. TS1 prevents staleness because x would appear in an invalidate on p_i at the end of epoch e_w. The value in cache in epoch $e_w - 1$ would be removed since p_i did not access x on epoch e_w and left its epoch reset from the end of epoch $e_w - 1$.

If compile time analysis were perfect, TS1 would have the same hit rate as a global scheme. The conservative assumptions that must be made at compile time cause some reuse to be missed. This is the loss that any local scheme must suffer.

4.4 Contrast between TS1 and previous strategies

The best way to understand why TS1 and TS have the same behavior is to return to TS and see it from a different point of view. There are only three states that a cache line can be in with respect to its clock, *just referenced*, *not yet stale*, and *stale* (Table 2). Taking these in reverse order, the *stale* state indicates that for a value x in epoch $e_w + 1$ a new value might have been written in epoch, e_w, after x was loaded in the cache in an earlier epoch, e_a. The *not yet stale* state persists from *after* epoch e_a when x was last accessed *through* epoch e_w which actually makes the line stale (DOALL semantics, section 2.1). The *just accessed* state sets the time stamp so that it will be in the *not yet stale* state in the next epoch. The *just accessed* state persists only for epoch e_a. The time stamp will be either clock+1 or clock depending on whether or not there is a write in epoch e_a.

TS1 implements these same three states by using the *epoch* bit in addition to the *valid* bit. This economy is possible because TS1 invalidates only those locations which could have been written. TS1 ages cache lines explicitly by updating bits in the cache line. TS ages cache lines implicitly by updating a processor clock for later comparison. TS is a lazy strategy.

TS as proposed enforces coherence on the whole array level. TS1 can be used to enforce coherence at the finest available resolution of compiler analysis. This is no worse than the whole array level and often better. TS could in principle do this well too by having a distinct clock for every section of an array which can be recognized at compile time. The cost of that could grow large.

LSS could utilize the same high level of analysis but it makes the pessimistic assumption that all paths are taken. Also, its counters quickly overrun regardless of the path taken. TS1 achieves at least the same hit rate as PEI because both can use the best available compiler analysis.

Another way to view this distinction of different strategies is the manner in which global information is passed. In FSI and LSS, global knowledge is never passed. In TS, global knowledge is passed implicitly by each processor incrementing an array clock for those arrays which might have been modified. In TS1 and PEI, global knowledge is passed implicitly by invalidating a section of memory that could have been written on a different processor. For local strategies, there is no way to avoid the invalidate because it is responsible for conveying the global information. The invalidate can be implicit, explicit, pessimistic, or reasonably precise, but it still has the same function. Table 3 summarizes the costs and capabilities of the different strategies.

4.5 Example

In figure 4, DOALLs are expanded into the worksharing part (PDO) where each processor gets some number of iterations, the common part that all processors execute, and the BARRIER, which is the end of the DOALL. Applying the TS1 compile time phase inserts the *INVALIDATE*'s. For each of the three DOALL epochs, there is an invalidate to cover what was written in that epoch. Table 4 shows the effect on TS1 control bits for the simple schedule, processor 1 gets iteration 1 on each epoch. At line 11, `A(1)` and `B(1,*)` have been referenced on p_1 in this epoch. So all cache lines holding these values are valid and have the *epoch* bit set. At line 12, the invalidate removes everything written on p_2 from p_1's cache. If `B(2,1)` were present on p_1 it would be removed at this point. `B(1,1)` however has its *epoch* bit set and stays in the cache on p_1. All *epoch* bits are reset at the barrier. At line 21 only `B(1,1)` has its *epoch* bit set on p_1 since it was the only reference. The invalidate does not reference `A` or columns of `B` other than the first. So, all of those stay in cache. The invalidate of `B(1,1)` finds the *epoch* bit set and leaves it in cache. At line 30, all references are then hits.

Using the same example for TS, the write to `B` in the second epoch would cause all columns of `B` to be invalidated. Thus, in the third epoch, all but one element of `B` would be a miss on p_1.

For LSS, `A` would suffer the same as the other columns of `B` and would miss in the third epoch.

For PEI, the second epoch would be handled perfectly by only invalidating the first column of `B`. How-

	FSI	LSS	PEI	TS	TS1
Inter Epoch Reuse	No	Yes	Yes	Yes	Yes
Granularity of analysis	N/A	whole program	array section	array	array section
bits/cache line	2	3	0	$2 + n_{clock}$	1
bits/instruction	1	2	n	$5 + r_{clock}$ (for reads) $2 + r_{clock}$ (for writes)	0
special bits/processor	0	0	0	$s * n_{clock}$	0
cost of invalidate	O(1)	O(1)	O(1)	O(S)	$O(\sum_{i=1}^{S} \log(s_i))$
handles DOACROSS	No	No	No	Yes	Yes

Notation

n	number of address bits
n_{clock}	number of bits needed to hold clock value
r_{clock}	number of bits to designate a clock
S	number of distinct sections (or arrays) which are written in an epoch
s_i	size of the ith section out of S total sections

Table 3: Comparison of methods

```
   PDO I=1,N
      DO J=1,N
         B(I,J)=A(I)+1
      ENDDO
   ENDDO
11
   CALL INVALIDATE (B(1,1),B(N,N))
   BARRIER
12
   PDO I=1,N
      B(I,1)=0
   ENDDO
21
   CALL INVALIDATE (B(1,1),B(N,1))
   BARRIER
22
   PDO I=1,N
      DO J=1,N
30       C(I,J)=B(I,J)+A(I)
      ENDDO
   ENDDO
   CALL INVALIDATE (C(1,1),C(N,N))
```

Figure 4: Example compiler output for TS1

ever, the first epoch would leave only the last column in cache. In the third epoch, the first and last column of B plus all of A would hit. The rest of B would miss.

5 Performance

The execution time of TS versus TS1 depends on two factors, the hit rate and the additional cycles used by a more sophisticated invalidate. We present experimental data on the former. We analyze the latter using reference traces from a real program combined with a hypothetical implementation of an invalidate

	Array Element on Processor 1		
Statement	A(1)	B(1,1)	B(1,2)
11	1 , 1	1 , 1	1 , 1
12	1 , 0	1 , 0	1 , 0
21	1 , 0	1 , 1	1 , 0
22	1 , 0	1 , 0	1 , 0
30	hit	hit	hit

Table 4: Example of bit handling in TS1, Entries: *valid* bit,*epoch* bit

We compared TS and TS1 on a small test suite of scientific Fortran programs. These were chosen because they were available, familiar to the authors, and easily convertible to use with simulator. Our methodology was to apply the TS and TS1 algorithms by hand to parallel Fortran programs. For TS1, the same invalidate calls were added at the end of each epoch as the compiler would have produced. We assumed the compiler could recognize only affine subscript expressions. For TS, invalidate calls were applied to whole arrays in an epoch for which the array appeared on the left hand side of an assignment. This has the same effect on hit rate as the suggested TS implementation These modified programs were then run through the the RPPT [5] simulator. This simulator operates by modifying the assembly code to trap at every global memory reference which is then passed off to a particular architecture simulator.

For identical runs of the test programs, we compared TS, TS1, and hardware coherence. For hardware coherence, we simulated write back caches with an invalidate protocol (WB).

Cyclic work distributions were used. Statistics reflect only shared data and not local data or instruction caching. Caches of sufficient size were simulated

	Procs	Size	TS	TS1	WB
LU	10	100	88.7	89.8	90.8
Heat Flow	20	60	62.6	63.5	63.5
Direct	4	4	97.1	97.1	97.7
Erlebacher	10	20	96.0	97.2	97.6

Table 5: Hit Ratios (%) for different strategies

so that no evictions occurred due to cache size or organization limitations.

Our test programs were:

LU decomposition - a blocked right looking LU decomposition with a blocking factor of 5.

Heat Flow - a simple 2-D heat flow relaxation

Direct - a simplex solver

Erlebacher - a tridiagonal solver for finding derivatives

5.1 Hit rates

Table 5 shows the hit rates for our test suite. Similar relative hit ratios resulted from different combinations of processor and block sizes, except for extreme cases. For larger problem sizes with evictions, the gap narrows.

In both LU and Heat Flow, TS1 managed to find extra hits by not invalidating the whole array in loops that set border elements of the (sub-)arrays. Direct made heavy use of indirection arrays which defeat all attempts at analysis. TS1 could do no better than TS in this case. For Erlebacher, TS1 was able to find substantial benefit because the main computation was distributed through several loops, many of which only modified a small section of a given array.

5.2 Invalidate overhead

Erlebacher is more than a computational kernel (so is Direct, but it showed no improvement). So, we focus on Erlebacher.

To better analyze this case, we looked more carefully at the *miss margin*, the number of extra misses per processor that TS suffers compared to TS1. For a fixed problem size, TS1 does worse as the number of processors increase because the total hit rate is only slightly affected causing the number of misses per processor to drop almost linearly. For a fixed number of processors, TS1 is favored by the same reasoning (for all but the simplest of invalidate implementations). Most importantly, as main memory latency increases TS1 is favored. For an invalidate cost model(section 4.2), we assumed that a contiguous section can be invalidated in $1 + \lfloor \log_2(|section|) \rfloor$ "invalidate cycles" (by "invalidate cycle" we mean the time it takes to invalidate a single, aligned, power-of-2-sized block)

We applied this cost model to a series of Erlebacher runs where the problem size was varied from 2 to 30 as the number of processors varied from 1 to 15. We

Procs	Size	Refs	Miss Margin	Cost	Penalty
		1,000's / processor			
1	2	7.8	0.1	0.8	0.2
2	4	16.2	0.1	1.9	0.3
3	6	31.9	0.3	3.6	0.4
4	8	52.8	0.6	5.6	0.5
5	10	79.2	0.9	8.2	0.5
6	12	110.8	1.3	11.6	0.5
7	14	147.9	1.7	15.0	0.6
8	16	190.3	2.2	19.0	0.7
9	18	238.1	2.8	24.1	0.7
10	20	291.2	3.4	29.1	0.7
11	22	349.7	3.7	34.5	0.8
12	24	413.6	4.8	40.3	0.8
13	26	482.8	5.1	49.0	0.8
14	28	557.45	6.4	56.2	0.9
15	30	637.33	7.4	63.9	0.9

Cost: Invalidate cycles for $\log_2(s_i)$ metric
Penalty: Invalidate cycles for whole array invalidates

Table 6: Erlebacher Profitability

chose this as a natural scalability condition because the hit rate for TS stayed fairly constant with this condition. The raw data is summarized in table 6. For each run, it lists the total shared data references, the miss margin, the invalidate cost (for invalidating precise sections), and worst case TS1 penalty (in invalidate cycles). All of these are normalized to be per processor. Profitability depends on the relative cost of a miss versus the cost of an invalidate cycle. Figure 5 shows the profitability region for some hypothetical miss costs. The profitability is expressed as the percent speed up for a completely memory bound program (compute time is completely overlapped) with the assumption that cache hits take 1 processor cycle and an invalidate cycle takes 2 processor cycles. For different invalidate cycle costs, figure 5 would look essentially the same by scaling the miss cost the same amount. For real machines, we expect the cost of a miss to be 10's of cycles. We expect an implementation of invalidate to be possible where one invalidate cycle takes only 2 processor cycles.

For this test case, if the cost of a miss is almost negligible (1 invalidate cycle) then careful invalidation gains nothing and loses in overhead. If the cost of a miss is 20 cycles, it is a break even proposition. For higher miss costs, TS1 improves performance by paying for the invalidate overhead with time saved from more cache hits.

For miss costs less than 20 cycles, it is possible to switch from a precise invalidate to one which invalidates the whole array. This has the same hit rate as TS, but greatly reduces overhead. For instance, in the 15 processor case the overhead of invalidating whole arrays is about 0.3% (0.9/637*(2 processor cycles/ invalidate cycle)), even if every reference were a hit. The "loss lower bound" line in figure 5 represents this.

TS1 can often perform better than TS. Where it does worse, there is a fall back option, whole array invalidates, to cushion the loss to a tolerable amount.

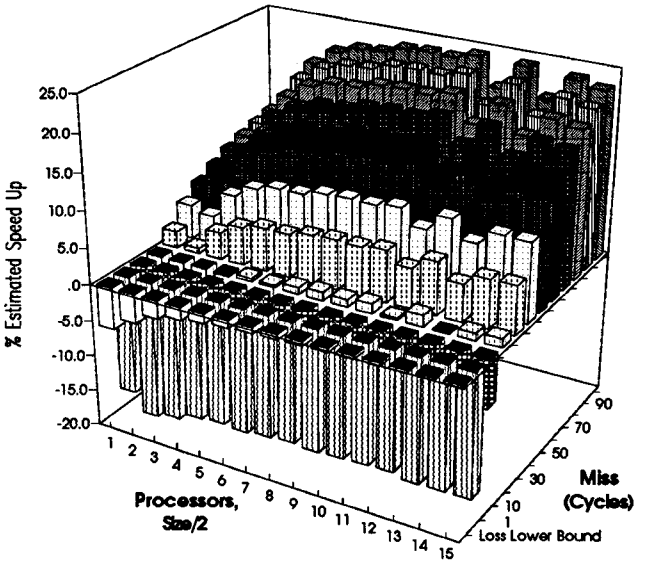

Figure 5: Erlebacher Speed Up

6 Other issues

6.1 DOACROSS

The assumed model for DOACROSS is that later iterations (instances) can wait on earlier iterations. There can be multiple waits on the same previous iterations or several different previous iterations. The compiler cannot necessarily determine anything about the nature of the synchronization. The only guarantee is that no dependences go from later to earlier iterations. A legal schedule for any DOACROSS would be to do the first p iterations with proper posting and waiting, then synchronize all p processors, and do the next p iterations, etc. Unlike DOALL, there can be carried dependence between iterations of a DOACROSS. So, each instance of a DOACROSS epoch must conceptually be treated like a different mini-epoch.

The leverage that local strategies get from the semantics of DOALL (section 2.1) must be abandoned here. For DOACROSS, the *epoch* bit in TS1 indicates that values can be reused on subsequent instances (not epochs). Since two subsequent iterations are almost certain to be scheduled to different processors, the *epoch* bit is useless.

Of the previous strategies surveyed, only Min and Baer's TS [14] handles DOACROSS. It increments the version number for an array at the end of a DOACROSS epoch. To preserve the semantics inside of the DOACROSS, any reference which could be overwritten in a later instance of the same DOACROSS epoch, is marked so that it will be removed at the end of the instance. Conversely, any read which could have been preceded by a write in a previous instance is invalidated on entry to this instance (intra-instance locality is still preserved).

Min and Baer's TS still preserves inter-epoch reuse if it can be proven (via the best available compiler analysis) that a given access will not be over-written on a later instance in the same epoch. Likewise, a read need not be forced to miss if it can be proven that no write on a previous instance of this epoch could reach it. Min and Baer's TS handles this situation with extra bits in the instruction word to specifically mark this condition. This is no longer optimal, even in the restricted sense of local strategies being optimal. Certain kinds of inter-instance intra-epoch reuse could be recognized by a local strategy, but are lost here.

TS1 could work in essentially the same way as TS for DOACROSS by adding the same extra bits. These extra bits could be avoided by changing the invalidation strategy. Instead of invalidating at the end of each epoch, the invalidate could be moved to the start of each instance. The invalidate would then handle those values written since this processor was last scheduled. For instance, if processors are assigned to iterations in strictly cyclic order and there are 5 processors. Then, processor 5, when it gets assigned iteration 11, would invalidate everything written on iterations 7 through 10. Iteration 6 writes were previously performed on processor 5 and do not need to be invalidated. Iterations before 6 were handled when processor 5 was assigned iteration 6. At the end of the DOACROSS, every processor must invalidate writes that occurred since it was last scheduled. This preserves the same inter-epoch reuse as Min and Baer's TS strategy.

In some cases involving DOACROSS, a *live* value is guaranteed to be invalidated before its next reference. In this case, there is no need to allocate a cache line. Read-thru and write-thru could selectively be used to advantage. Min and Baer [14] discuss this at length. This can be done with the bits already present in their strategy. TS1 could accommodate this with extra instruction bits performing the same function as in the Min and Baer strategy.

6.2 Critical sections

For critical section semantics that require inter-instance dependences to be entirely within critical sections, it is a simple matter to maintain coherence. Whatever is written in the critical section must be updated before the end of the section. Whatever is read in the critical section that could be written in another critical section must be invalidated on entry to the critical section.

6.3 Cache line size

Cache lines larger than one word cause aliasing problems. Values are read when they do not appear to be (as seen by the compiler). These values must also be kept coherent. There are several ad hoc methods of dealing with the aliasing problem, e.g. padding of array dimensions, changing layout order, and stripping loops. In truly desperate cases, it may be necessary to always use write-thru or not use caching at all. We assume that such objects are allocated on special cache pages in order to indicate different handling.

7 Conclusions

We believe the proper way to consider coherence strategies is in a framework of local knowledge versus global knowledge, and not as software versus hardware. This paper contributes to that framework.

As local strategies continue to improve their hit rates and decrease their implementation costs, it becomes feasible, at least for scientific codes, to build shared memory multiprocessors which rely on local strategies instead of global strategies for cache coherence. These machines have fewer obstacles to scalability. They may also be less hardware intensive than the sophisticated global strategies which have been proposed for medium and large scale parallelism.

In this paper, we propose a new local strategy, TS1 that improves on the best previously existing local strategy, time-stamping, by achieving better hit ratios, requiring fewer bits per cache line, and no extra bits per instruction. For a given granularity of compiler analysis, no local scheme could ever achieve a higher hit ratio. TS1 requires an *epoch* bit per cache line, a mechanism to invalidate an address range of cache lines, and a compiler that can recognize which array sections are written in a given epoch. If the compiler can only recognize arrays, and not sections, TS1 will have the same hit rate as time-stamping.

Simulation studies show that TS1 almost always has better hit ratios than TS and never worse. TS1 occasionally has slightly worse performance, but often appreciably better performance. An open question is to determine how efficiently a range based invalidate can be implemented. If only inefficient or hardware intensive implementations can be found, TS1 will not be as effective as our data suggest.

Acknowledgments

We would like to thank U. Rajagopalan and S. Dwarkadas for their assistance in modifying the RPPT simulator to handle the specific needs of a local strategy. We would also like to especially thank John Mellor-Crummey for his many useful suggestions on improving this paper.

References

[1] D. A. Abramson, K. Ramamohanarao, and M. Ross. A scalable cache coherence mechanism using a selectively clearable cache memory. *The Australian Computer Journal*, 21(1), Feb. 1989.

[2] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, Dec. 1978.

[3] H. Cheong. Life-span strategy - a compiler-based approach to cache coherence. In *Proceedings of 1992 International Conference on Supercomputing*, July 1992.

[4] H. Cheong and A. Veidenbaum. Compiler-directed cache management for multiprocessors. *Computer*, 23(6):39–47, June 1990.

[5] R. Covington, S. Dwarkadas, J. Jump, J. Sinclair, and S. Madala. Efficient simulation of paralle computer systems. *International Journal in Comuter Simulation*, 1:31–58, 1991. overview of RPPT.

[6] R. Cytron, S. Karlovsky, and K. McAuliffe. Automatic management of programmable caches. In *Proc. of the 1988 International Conference on Parallel Processing*, pages 229–238, Aug. 1988.

[7] E. Darnell, J. Mellor-Crummey, and K. Kennedy. Automatic software cache coherence through vectorization. In *Proceedings of 1992 International Conference on Supercomputing*, July 1992. Also available as expanded Technical Report CRPC-TR92197, Center for Research on Parallel Computation, January 1992.

[8] D. James, A. Laundrie, S. Gjessing, and G. Sohi. Scalable coherent interface. *Computer*, 23(6), June 1990.

[9] S. Karlovsky. Automatic management of programmable caches: Algorithms and experience. Technical Report 89-8010, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, July 1989.

[10] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), Sept. 1979.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Standford DASH multiprocessor. *Computer*, 25(3):63–79, Mar. 1992.

[12] A. Louri and H. Sung. A compiler directed cache coherence scheme with fast and parallel explicit invalidation. In *Proc. of the 1992 International Conference on Parallel Processing*, pages 2–9, Aug. 1992.

[13] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Proceedings of 1992 International Conference on Supercomputing*, pages 104–113, Nov. 1992.

[14] S. Min and J. Baer. Design and analysis of a scalable cache coherence scheme based on clocks and timestamps. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):25–44, Jan. 1992.

[15] S. Min, J. Baer, and H. Kim. An efficient caching support for critical sections in large-scale shared-memory multiprocessors. In *Proc. of the 1990 International Conference on Supercomputing/Computer Architecture News*, pages 4–47, June 1990. Special issue of Computer Architecture News, 18(3), Sept. 1990.

[16] A. Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems*. Sequent Technical Publications, San Diego, CA, 1989.

[17] K. Peterson and K. Li. Cache coherence for shared memory multiprocessors based on virtual memory support. In *Proceedings of the 7th International Parallel Processing Symposium*, Apr. 1993.

[18] D. Schanin. The design and development of a very high speed system bus – the encore multimax nanobus. In *Proceedings of the Fall Joint Computer Conference*, pages 410–418, Nov. 1986.

[19] J. Willis, A. Sanderson, and C. Hill. Cache coherence in systems with parallel communication channels & many processors. In *Supercomputing '90*, pages 554–563, 1990