Exploiting Cache Affinity in Software Cache Coherence

Hui Li and Kenneth C. Sevcik

Technical Report CSRI-299
April 1994

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Exploiting Cache Affinity in Software Cache Coherence *

Hui Li and Kenneth C. Sevcik

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A4
Email: {*hui*|*kcs*}@*csri.toronto.edu*

Abstract

Cache affinity is important to the performance of scalable shared memory multiprocessors. For multiprocessors without hardware cache coherence support, software cache coherence is the only alternative. Most existing software cache schemes ignore cache affinity across parallel loops. In this paper, we propose a new scheme, Cache Affinity-based Software cache coherence scheme (CAS), that exploits cache affinity across parallel loops to achieve high cache hit ratios without requiring extra hardware support. The experimental results show that the new scheme outperforms other existing schemes.

1 Introduction

Private caches are widely used in scalable shared memory multiprocessors for reducing memory access latency. Multiple copies of data in caches introduce a coherence problem. Directory based hardware cache coherence schemes [26] have been used in many multiprocessor systems, such as the KSR-1 [18], Dash [20], and Alewife [1]. They provide the user a coherent memory model, but require substantial hardware support. There are also a number of multiprocessors, including Cedar [17], Hector [27], BBN TC2000 [4], and the recently announced Cray T3D, which do not have hardware support for cache coherence. On these machines, software cache coherence is the only alternative.

Cache coherence can be maintained by the OS through virtual memory paging [21, 5, 11, 6]. However, the page size is too large to avoid false-sharing and the cost of page-fault handling is too high to amortize. These problems can be alleviated if users or compilers maintain cache coherence by inserting explicit cache operations. The shared regions approach [25] provides

*This research has been supported by research grants from the Natural Sciences and Engineering Research Council of Canada and from the Information Technology Research Center of Ontario.

a high-level abstraction making it easy for users to maintain cache coherence. However, it still requires users to identify which memory references require a remote read/write, which is nearly as tedious as inserting send/receive primitives for message passing systems.

To relieve the user from maintaining cache coherence, several compiler-based software schemes have been proposed. These schemes use compiling techniques to identify potential stale data and to insert invalidate statements to enforce coherence. Thus, the effectiveness of software cache coherence schemes depends on the cache hit ratios and the invalidation overhead.

To reduce the invalidation overhead, Cytron, Karlovsky, and McAulife (CKM) [12] attempt to minimize the amount of data to be invalidated, and Darnell, Mellor-Crummey, and Kennedy [13] apply vectorization techniques to eliminate redundant invalidations. The second scheme is called Software Cache Coherence through Vectorization (SCTV). Both of these schemes, however, only consider single parallel loops, so the cached data cannot be reused across parallel loops. Without exploiting such cache affinity, unnecessarily high miss ratios may result.

Cache affinity is very important to performance on scalable shared memory multiprocessors. The KSR-1 Parallel Fortran [19] provides a special directive, *Affinity Region*, to annotate such cache affinity. Rothber, Singh, and Gupta [24] show that in most computationally intensive applications, the working sets are much smaller than the data sets, and they grow slowly as the problem size. Therefore, with increasing cache sizes and numbers of processors, cache affinity becomes more important.

Several exiting software schemes attempt to retain cache affinity across parallel loops. They include the Fast Selective Invalidation (FSI) [9, 10] the Version Control [7], the Time Stamp [22, 23], and the Life Span [8] schemes. But all these approaches require additional bits to be added into each cache line. Thus they cannot be implemented on current NUMA multiprocessors (such as BBN TC2000 and Hector).

In this paper, we propose a new software cache coherence scheme that uses cache affinity to avoid unnecessary cache operations, so it is called Cache Affinity-based Software cache coherence scheme (CAS). The CAS scheme exploits cache affinity across parallel loops to achieve high cache hit ratios without requiring extra hardware support. We discuss cache affinity and its impact on cache hit ratios in existing software cache coherence schemes in the next section. We describe the CAS scheme in Section 3 and present its details in Section 4. Section 5 presents experimental results on the Hector multiprocessor and compares performance with some existing schemes.

2 The Effect of Cache Affinity

We use an example shown in Figure 1 to illustrate cache affinity across parallel loops. This example contains two parallel loops enclosed in a sequential loop. Arrays are stored in row-major fashion. The elements of arrays A and B are heavily reused between the two parallel loops. So most cache lines fetched in loop j_1 may still be valid in loop j_2. To exploit such cache affinity (temporal locality), both parallel loops j_1 and j_2 should be scheduled in a block

```
DO I = 0, Iters
  DOALL j1 = 1, 1000            /* loop j1 */
    DO k = 1, 1000
      A(j1, k) = f(B(j1, k), B(j1 - 1, k), B(j1 + 1, k),
                   B(j1, k - 1), B(j1, k + 1))
    END DO
  END ALL

  DOALL j2 = 1, 1000            /* loop j2 */
    DO k = 1, 1000
      B(j2, k) = A(j2, k)
    END DO
  END ALL
END DO
```

Figure 1: An Example of Cache Affinity Across Parallel Loops

fashion so that the corresponding blocks of these loops can be executed by the same processor. In this way:

- Each element of array A is read and written by the same processor, so array A does not cause any cache coherence problem.
- Each element of array B, except those in boundary rows, is also read and written by the same processor. The boundary rows on processor p are shared by processors $p-1$ and $p+1$ in loop j_1. To maintain cache coherence for array B, each processor needs to invalidate two neighboring rows before parallel loop j_1 and to update its two boundary rows after parallel loop j_2.

Assume that the total size of all caches is large enough to accommodate the entire arrays A and B, the cache line size is two words, and each processor executes 100 iterations of each parallel loop. Then under the optimal scheme discussed above, the cache hit ratios for arrays A and B are one and 0.99 respectively. Table 1 indicates how various proposed schemes do relative to the optimal.

The SCTV scheme [13] does not exploit cache affinity across parallel loops. It conservatively assumes that

- every variable (except read-only variables) may be modified by other processors, so upon entering each parallel loop, the cache lines corresponding to the variable are invalidated before the variable is used; and
- every modified variable in the current parallel loop may be used by other processors in the next parallel loop, so the corresponding updated cache lines are written into main memory.

	Hardware Support	A	B
Optimal	Full Hardware Directory	1.00	0.99
SCTV	No	0.50	0.80
FSI	1 bit/cache line	0.50	0.80
Life Span	2 bits/cache line	1.00	0.99
Version Control	several bits/cache line	1.00	0.99
Time Stamp	several bits/cache line	1.00	0.99
CAS	No	1.00	0.99

Table 1: Estimated Cache Hit Ratios

Under SCTV, in loop j_1 of this example, each processor invalidates the rows of array B to be read before loop j_1 and writes back the modified rows of array A after loop j_1. In loop j_2, similarly, each processor invalidates the rows of array A to be read and writes back the modified rows of array B. Thus, no cached data can be reused between these two parallel loops. In this example, each cache line for array B is accessed approximately eight times in loop j_1 and twice in loop j_2. Thus, the cache hit ratio is 8/10 (see Table 1).

The FSI scheme [9] uses an extra bit in each cache line to indicate valid cache lines within parallel loops, but it cannot exploit cache affinity beyond a single parallel loop. So FSI obtains the same cache hit ratios as those in SCTV. To exploit cache affinity across parallel loops, the Life Span strategy adds two bits to each cache line to identify the cache lines accessed in the last parallel loop that can be used in the current loop. Since the cache affinity in the example is between the two parallel loops, the Life Span strategy [8] can also capture such cache affinity and achieve the same cache hit ratios as those in the optimal scheme. The Version Control and the Time Stamp strategies [7, 22] use several bits per cache line, thus they can retain cache affinity across many parallel loops. The Cache Affinity-based Software cache coherence scheme (CAS) captures cache affinity across parallel loops by software. We will disucss it in the next section.

In summary, by considering cache affinity across parallel loops, the cache hit ratios for arrays A and B are increased by 100% and 24% respectively. In the next section, we propose a new scheme that achieves these gains without requiring any hardware support.

3 Cache Affinity-based Scheme

In this section, we present the Cache Affinity-based Software cache coherence scheme (CAS). We first describe assumptions and introduce notation, then define the CAS scheme.

```
DO I = 1, N
  DOALL J_1 = ...
    ...
  END ALL
  DOALL J_2 = ...
    ...
  END ALL
  ...  ...
  DOALL J_n = ...
    ...
  END ALL
END DO
```

Figure 2: Loop Structure

3.1 Assumptions and Notation

We consider systems in which each processor has a private cache without hardware cache coherence support, but the memory management unit (MMU) provides three cache operations: *invalidate*, *update*, *flush*, which operate on the private cache at the location indicated by a virtual address and with a size of a cache line, a page, or the entire cache. Typically, the costs of these operations are not linear with the amount of data affected. Thus, the overhead can be reduced by aggregating operations on small units into fewer operations on larger units.

We assume that parallelism is expressed strictly via `DOALL` parallel loops, in which there are no data dependencies between their iterations [28]. The general form of loop nest treated in this paper is shown in Figure 2. The outermost sequential loop contains several parallel loops, where each parallel loop may contain a sequence of sequential loop nests. The outermost loop bounds are constants, inner loop bounds are linear functions of outer loop index variables, and all strides are constants. We further assume that array indices are linear functions of loop index variables. Read-only arrays and scalar variables are not discussed because read-only arrays do not introduce cache coherence problems. Since multiple arrays can be treated one by one, for simplicity, we assume that there is only one array A in the loop nest and it appears in every parallel loop. If array A is not used in some parallel loops, these loops can be ignored in applying the cache coherence algorithm.

Let $Inv(A, J_k, I, p)$ and $Update(A, J_k, I, p)$ repectively represent the minimum portions of the array which have to be invalidated before and updated after parallel loop J_k of the Ith iteration of the sequential outermost loop on processor p. To compute $Inv(A, J_k, I, p)$ and $Update(A, J_k, I, p)$, we introduce the following notation similar to that used in data flow analysis [2]:

j_1	$Def(B, j_1, 2, p_1)$	empty
	$Def(B, j_1, 2, \overline{p_1})$	empty
	$Use(B, j_1, 2, p_1)$	rows 100 to 201
	$Use(B, j_1, 2, \overline{p_1})$	rows 0 to 101, 200 to 1000
	$Ref^{down}(B, j_1, 2, p_1)$	rows 101 to 200
	$Def^{up}(B, j_1, 2, p_1)$	rows 101 to 200
j_2	$Def(B, j_2, 2, p_1)$	rows 101 to 200
	$Def(B, j_2, 2, \overline{p_1})$	rows 1 to 100, 201 to 1000
	$Use(B, j_2, 2, p_1)$	empty
	$Use(B, j_2, 2, \overline{p_1})$	empty
	$Ref^{down}(B, j_2, 2, p_1)$	rows 100 to 201
	$Def^{up}(B, j_2, 2, p_1)$	rows 101 to 200

Table 2: Data Sets for the Example in Figure 5.1

- $Use(A, J_k, I, p)$ and $Def(A, J_k, I, p)$ are the portions of array A read/written in parallel loop J_K within sequential loop I by processor p respectively, and $Ref(A, J_k, I, p)$ is defined as $Use(A, J_k, I, p) \cup Def(A, J_k, I, p)$;
- $Use(A, J_k, I, \overline{p})$ and $Def(A, J_k, I, \overline{p})$, are the portions of array A read/written in parallel loop J_K within sequential loop I by processors other than p respectively, and $Ref(A, J_k, I, \overline{p})$ is $Use(A, J_k, I, \overline{p}) \cup Def(A, J_k, I, \overline{p})$;
- $Ref^{down}(A, J_k, I, p)$ is the portion of array A referenced (*i.e.*, read or written) before parallel J_K in sequential loop I by processor p, but which has not been written by other processors after the references until the parallel loop;
- $Def^{up}(A, J_k, I, p)$ is the portion of array A written after parallel loop J_k in sequential loop I by processor p, but which has not been referenced by other processors from the parallel loop until the writes; and
- $Use^{up}(A, J_k, I, \overline{p})$ is the portion of array A read after parallel loop J_k in iteration I of sequential loop I by processors other than p, but which has not been written by processor p from the parallel loop until the references.

To illustrate these definitions, we list in Table 2 the data sets for the example in Figure 1, where processor 1 executes iterations 101 to 200 in both of the parallel loops j_1 and j_2.

3.2 The Precise Algorithms

To maintain cache coherence for array A in parallel loop J_k in the Ith iteration of the outermost sequential loop, it is sufficient that the portion of array A to be read (*i.e.*, $Use(A, J_k, I, p)$) is invalidated before the parallel loop and the portion of array A written (*i.e.*, $Def(A, J_k, I, p)$)

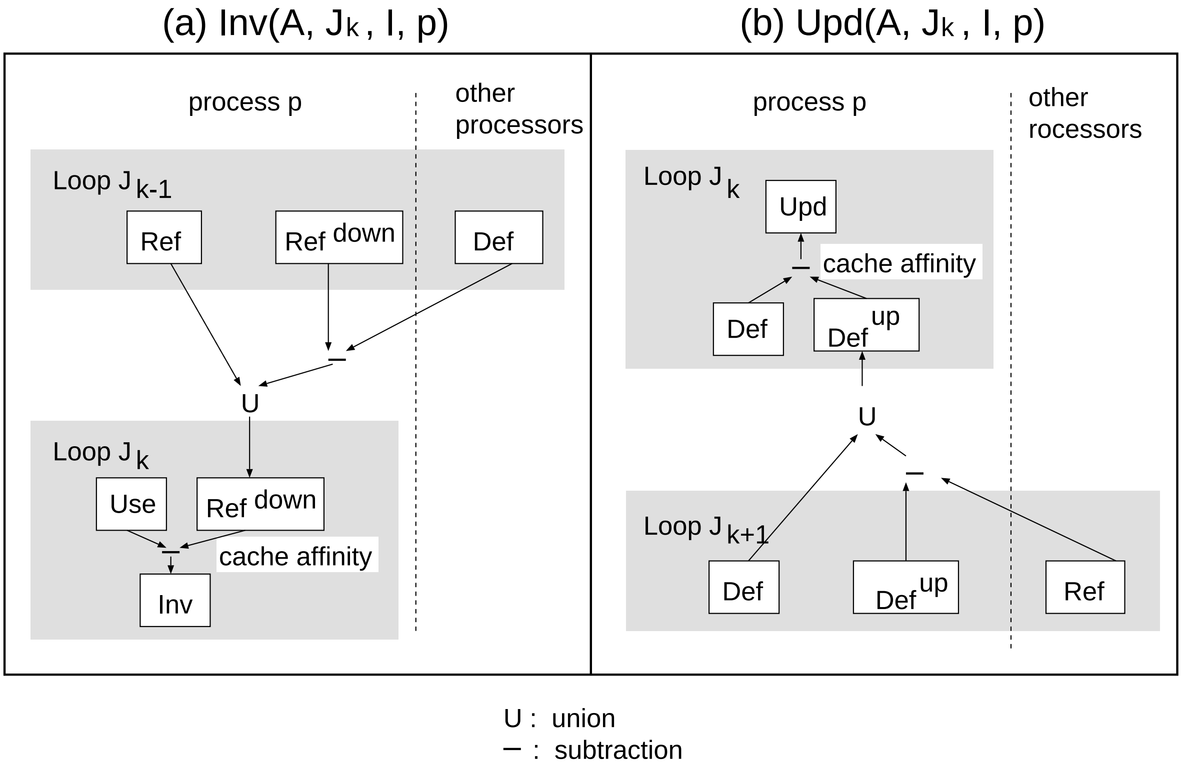

Figure 3: Calculations of Invalidation and Update Sets under CAS

is copied back to main memory after the parallel loop. Under this conservative scheme, however, data cannot be reused in the cache across parallel loops.

The cache affinity for parallel loop J_k can be precisely defined in terms of $Ref^{down}(A, J_k, I, p)$ and $Def^{up}(A, J_k, I, p)$.

- $Ref^{down}(A, J_k, I, p)$ represents the valid data set on processor p before parallel loop J_k, so the cache lines for the data set can be reused in parallel loop J_k by processor p.

- $Def^{up}(A, J_k, I, p)$ represents the data set that will be rewritten by processor p before being referenced by other processors, so the dirty cache lines for the data set do not need to be written back to the main memory.

We propose a Cache Affinity-based Software cache coherence scheme (CAS) to exploit cache affinity by avoiding unnecessary invalidations and updates as follows (see Figure 3):

$$Inv(A, J_k, I, p) = Use(A, J_k, I, p) - Ref^{down}(A, J_k, I, p) \quad (1)$$

$$Upd(A, J_k, I, p) = Def(A, J_k, I, p) \cap Use^{up}(A, J_k, I, \overline{p}) - Def^{up}(A, J_k, I, p) \quad (2)$$

<table>
<tr><th></th><th>$Ref^{down}(A, J_k, I, p)$</th><th>$Def^{up}(A, J_n, I, p)$</th></tr>
<tr><td>$k = 1$</td><td>$Ref(A, J_n, I-1, p) \cup$
$(Ref^{down}(A, J_n, I-1, p) -$
$Def(A, J_n, I-1, \overline{p}))$</td><td rowspan="2">$Def(A, J_{k+1}, I, p) \cup$
$(Def^{up}(A, J_{k+1}, I, p) -$
$Ref(A, J_{k+1}, I, \overline{p}))$</td></tr>
<tr><td>$1 < k < N$</td><td rowspan="2">$Ref(A, J_{k-1}, I, p) \cup$
$(Ref^{down}(A, J_{k-1}, I, p) -$
$Def(A, J_{k-1}, I, \overline{p}))$</td></tr>
<tr><td>$k = N$</td><td>$Def(A, J_1, I+1, p) \cup$
$(Def^{up}(A, J_1, I+1, p) -$
$Ref(A, J_1, I+1, \overline{p}))$</td></tr>
</table>

Table 3: Computing Ref^{down} and Def^{up}

where "$-$" is the subtraction operation on sets. In Eq. (1), $Ref^{down}(A, J_k, I, p)$ represents the useful cache context for loop J_k that need not be invalidated. In Eq. (2), the term $Def^{up}(A, J_k, I, p)$ prevents redundant updates by the same processor, and the term $Use^{up}(A, J_k, I, \overline{p})$ prevents unnecessary updates where other processors do not use the updated values. Because of the complexity of $Use^{up}(A, J_k, I, \overline{p})$, we replace it by the total data set read by other processors after parallel loop J_k, $\sum_{i=I}^{N} \sum_{j=k+1}^{n} Use(A, J_j, i, \overline{p})$. Then Eq. (2) becomes:

$$Upd(A, J_k, I, p) \subseteq Def(A, J_k, I, p) \cap \sum_{i=I}^{N} \sum_{j=k+1}^{n} Use(A, J_j, i, \overline{p}) - Def^{up}(A, J_k, i, p) \quad (3)$$

The remaining issue is to compute the cache affinity (Ref^{down} and Def^{up}) for each parallel loop. Figure 3 shows that Ref^{down} can be computed forwards and Def^{up} backwards from the Def and Use sets, respectively. The detailed calculations are specified in Table 3. For parallel loop J_k $(k > 1)$, $Ref^{down}(A, J_k, I, p)$ may include

- $Ref(A, J_{k-1}, I, p)$: the data referenced by processor p in parallel loop J_{k-1}, and
- $Ref^{down}(A, J_{k-1}, I, p)$: the valid data set before parallel loop J_{k-1}.

However, the valid data set on processor p should not include the data set written by other processors in parallel loop J_{k-1}, $Def(A, J_{k-1}, I, \overline{p})$. Since

$$Def(A, J_{k-1}, I, \overline{p}) \cap Ref(A, J_{k-1}, I, p) = \emptyset$$

we do not need to subtract the Def set from the Ref set. If there exist control dependencies in the loop nest, the data flow analysis for array accesses proposed by Granston and Veidenbaum [14] can be incorporated into our algorithms.

Figure 4 shows the relationships with respect to cache affinity for invalidation among several software cache coherence schemes. The SCTV and FSI schemes consider only single

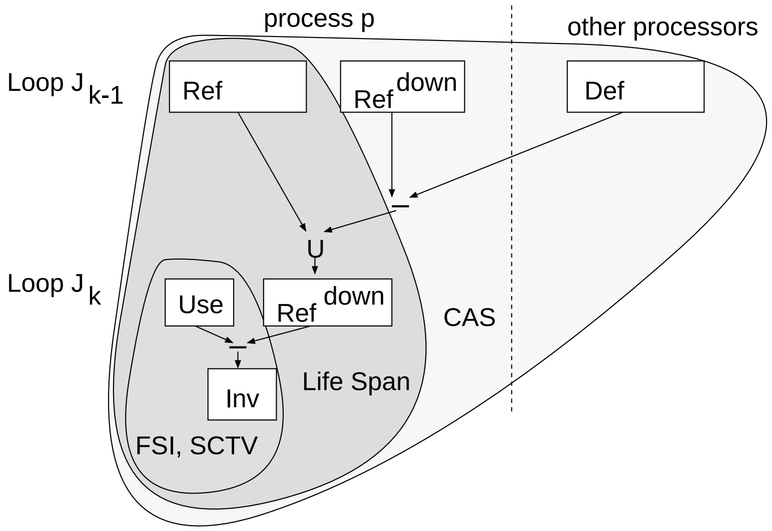

Figure 4: Cache Affinity in Software Cache Coherence Schemes

parallel loops without exploiting cache affinity across parallel loops. The Life Span strategy uses a *change* bit and a *stable* bit so that data accessed in the last parallel loop need not be invalidated. So its cache affinity is the data set accessed by the same processor in the last parallel loop. The Version Control and the Time Stamp strategies retain cache affinity across many parallel loops. So their cache affinity can be the same as in the CAS scheme. However, they require substantial hardware support.

3.3 The Approximate Algorithms

In Section 2, we assume that Def and Use sets can be accurately obtained and that operations on these sets are closed. However, if we use regular sections [3, 15] to represent Use and Def data sets, neither of these two assumptions holds. Therefore, we relax the assumptions and introduce approximate algorithms.

Let us use subscripts max and min to represent supersets and subsets. For example, Def_{max} is a superset of Def such that $Def_{max} \supseteq Def$. We also introduce the following approximate set operations:

- "$\cup_{max}$", "$\cup_{min}$" give a superset and a subset of the union of two sets respectively.
- "$-_{max}$", "$-_{min}$" give a superset and a subset of the difference between two sets respectively.
- "$\cap_{max}$" gives a superset of the intersection of two sets.

We use Inv_{max} and Upd_{max} to represent the data sets which will be invalidated and updated respectively in our approximate scheme. They can be computed by the following set of

<table>
<tr><th></th><th>$Ref^{down}_{min}(A, J_k, I, p)$</th><th>$Def^{up}_{min}(A, J_n, I, p)$</th></tr>
<tr><td>$k = 1$</td><td>$Ref_{min}(A, J_n, I-1, p) \cup_{min}$
$(Ref^{down}_{min}(A, J_n, I-1, p) -_{min}$
$Def_{max}(A, J_n, I-1, \overline{p}))$</td><td rowspan="2">$Def_{min}(A, J_{k+1}, I, p) \cup_{min}$
$(Def^{up}_{min}(A, J_{k+1}, I, p) -_{min}$
$Ref_{max}(A, J_{k+1}, I, \overline{p}))$</td></tr>
<tr><td>$1 < k < N$</td><td rowspan="2">$Ref_{min}(A, J_{k-1}, I, p) \cup_{min}$
$(Ref^{down}_{min}(A, J_{k-1}, I, p) -_{min}$
$Def_{max}(A, J_{k-1}, I, \overline{p}))$</td></tr>
<tr><td>$k = N$</td><td>$Def_{min}(A, J_1, I+1, p) \cup_{min}$
$(Def^{up}_{min}(A, J_1, I+1, p) -_{min}$
$Ref_{max}(A, J_1, I+1, \overline{p}))$</td></tr>
</table>

Table 4: Approximate Methods for Ref^{down}_{min} and Def^{up}_{min}

equations:

$$
\begin{aligned}
Inv_{max}(A, J_k, I, p) &= Use_{max}(A, J_k, I, p) -_{max} Ref^{down}_{min}(A, J_k, I, p) & (4)\\
Upd_{max}(A, J_k, I, p) &= Def_{max}(A, J_k, I, p) \cap_{max} \\
&\quad \sum_{i=I}^{N} \sum_{j=k+1}^{n} Use(A, J_j, i, \overline{p}) -_{max} Def^{up}_{min}(A, J_k, I, p) & (5)
\end{aligned}
$$

Using the approximate operations and sets, we can derive from Table 3 approximate methods for computing Ref^{down}_{min} and Def^{up}_{min} as shown in Table 4.

4 Parametric Section Analysis

In Section 3, we have developed the CAS cache coherence scheme. In this section, we discuss the details of this scheme. We first extend bounded regular sections to parametric regular sections, then use parametric regular sections to summarize array references using $Def^{min}, Def^{max}, Use^{min}$, and Use^{max} for each parallel loop. Finally, we define approximate set operations on parametric regular sections required by the approximate algorithm given in Section 3.3.

4.1 Parametric Sections

Bounded regular sections [15] use the triplet notation $[l : u : s]$ to represent a rectangular array region, where l and u are often called the lower and upper bounds respectively, and s is the stride. The data set of a reference in a loop can be summarized by a bounded regular section. For example, in the loop:

```
DO J = L, U, S
    A(2*J + 1) = B(J, J)
END DO
```

the data set accessed by $A(2*J+1)$ is represented as $A[(2*L+1):(2*U+1):(2*S)]$.

Bounded regular section analysis has been used in inter-procedural analysis [15] and in data distribution analysis [16]. To use it in the CAS scheme, we need to make some extensions for the following reasons:

- The algorithms for bounded section analysis give only supersets of sections for inter-procedural data dependence analysis [15]. However, our approximate cache coherence algorithms in the CAS scheme require both supersets and subsets of sections.
- The previous algorithms assume that the upper and lower bounds of sections are constants [15]. Since a parallel loop is scheduled among processors, parallel loop index variables should remain in sections for computing Inv and Upd sets so that the cache affinity on each processor can be determined.

We propose *parametric sections* that allow the index variables of the sequential loop and the parallel loop to appear in their upper and lower bounds. Parallel sections are in the form of $[L:U:s]$, where $L = a_l I + b_l J + c_l$ is the lower bound, $U = a_u I + b_u J + c_u$ is the upper bound, I and J are the index variables of the sequential loop and the current parallel loop respectively, and $a_l, b_l, c_l, a_u, b_u, c_u$ and s are constants. If both b_l and b_u are zero, the section is independent of the the parallel loop, and thus is called a *sequential* (parametric) section. A parametric section without index variables I and J (*i.e.* where $a_l = b_l = a_u = b_u = 0$) degenerates to a bounded regular section.

Two parametric sections in the same parallel loop are said to be in the same *parallel class* if they have equal a_l's and equal a_u's. They are called in the same *class* if they also have equal b_l's and equal b_u's. Sections within the same (parallel) class have some potential cache affinity. For example, consider $S_1 = [2I+3J : 3I+4J : 1]$, $S_2 = [2I+3J : 3I+4J+3 : 3]$, and $S_3 = [2I+3J : I+3J : 1]$. Then S_1 and S_2 are in the same class, but S_3 is in the same parallel class with S_1 and S_2, not in the same class.

We use parametric sections in our cache coherence algorithms. The parametric section analysis consists of two major steps. The first step represents the portion of an array accessed by each reference using a parametric section, and merges sections that are in the same class. The second step identifies cache affinity between parallel loops and computes Inv and Upd.

4.2 Summarizing References Using Parametric Sections

For each parallel loop, read and write references to each array are classified into four sets:

$$Use_{max}, Use_{min}, Def_{max}, Def_{min}$$

These sets are represented by a list of parametric sections. For a reference to a one-dimensional array, we compute two array sections, a superset and a subset of the array region touched by the reference (S_{max} and S_{min}), based on loop bounds and loop strides. If the reference is a read, S_{max} and S_{min} are added to Use_{max} and Use_{min} respectively. Otherwise, they are added to Def_{max} and Def_{min}. For a reference to a multi-dimensional array, we compute

```
real A[N][N]
DO 10 i = 1, N
   DOALL j = i+1, N
      A(j, i) = A(j, i)/A(i, i)
      DO k = i+1, N
         A(j, k) = A(j, k) - A(j, i)*A(i, k)
      END DO
   END ALL
END DO
```

Reference	Section	Merged Section
A(i, i) A(i, k)	A[i:i:1][i:i:1] A[i:i:1][i+1:N:1]	A[i:i:1][i:N:1]
A(j, i) A(j, k)	A[j:j:1][i:i:1] A[j:j:1][i+1:N:1]	A[j:j:1][i:N:1]

Table 5: Summarizing References in LU

array sections for each dimension independently. If the index expressions of the reference are coupled, the array section for the subset, S_{min}, is assumed to be empty.

After references are summarized as parametric sections, two sections (S_1 and S_2) in the same class are merged if they satisfy one of the following five conditions:

1. $S_1 \subseteq S_2$
2. $S_2 \subseteq S_1$
3. $S_1||S_2$: they can be concatenated if
   - they have the same stride,
   - the difference of their lower bounds are a multiple of the stride, and
   - the upper bound of S_1 is not less than the lower bound of S_2.
4. $S_2||S_1$: similar to 3.
5. S_1 and S_2 have the same (or close) upper and lower bounds, but with different strides. Since strides are usually small constants, we use bit vectors to represent strides. So the merging in this case is to set some bits in the bit vector.

Table 5 summarizes the sections for read references in the parallel LU decomposition. The supersets and subsets are the same. Among these sections, $A[i : i : 1][i : i : 1]$ and

$A[i:i:1][i+1:N:1]$ are in one class, and $A[j:j:1][i:i:1]$ and $A[j:j:1][i+1:N:1]$ are in the other, and they are merged into $A[i:i:1][i:N:1]$ and $A[j:j:1][i:N:1]$, respectively.

4.3 Approximate Operations for the CAS Scheme

In the CAS scheme, the Inv and Upd sets can be computed from the Use and Def data sets using equations in Section 2.3. These equations require the following approximate operations on parametric sections: "$\cup_{min}$", "$-_{min}$", and "$-_{max}$".

The "$-_{min}$" and "$\cup_{min}$" operations are used in computing both $Ref_{min}^{down}(A, J_k, I, p)$ and $Def_{min}^{up}(A, J_k, I, p)$. Since they are applied to the parametric sections from different parallel loops, cache affinity between parallel loops should be taken into account.

Consider two parametric sections from parallel loops J_k and J_l:

$$S_1 = [a_l^1 I + b_l^1 J_k + c_l^1 : a_u^1 I + b_u^1 J_k + c_u^1 : s_1]$$

$$S_2 = [a_l^2 I + b_l^2 J_l + c_l^2 : a_u^2 I + b_u^2 J_l + c_u^2 : s_2]$$

The two sections have "good" cache affinity if the following two conditions are satisfied:

- The loop scheduling schemes for the two parallel loops retain affinity. Three scheduling schemes are considered: block, cyclic, and block-cyclic. Block scheduling will be used to illustrate our method. To preserve affinity in block scheduling for parallel loops J_k and J_l, iterations J_k and $J_l + D$ are mapped onto the same processor, where D is a small constant such that $J_k = J_l + D$

- The two sections are in the same parallel class so that they can share most of the data.

In both Ref_{min}^{down} and Def_{min}^{up}, we need to compute parametric sections from different loops: $S_1(p) -_{min} S_2(\overline{p})$, where $S_1(p)$ is the portion of the section S_1 on processor p and $S_2(\overline{p})$ is the portion of the section S_2 on processors other than p.

- If $S_1(p)$ and $S_2(\overline{p})$ do not have "good" cache affinity, we can simply check whether $S_1(p)$ and $S_2(\overline{p})$ are disjoint. The result is set to $S_1(p)$ if they are disjoint, or $\emptyset$ otherwise.

- If they have "good" cache affinity, *i.e.*, $S_1(p)$ and $S_2(\overline{p})$ are in the same parallel class and use block scheduling such that $J_k = J_l + D$, then $S_2(\overline{p})$ may share a small portion of $S_1(p)$. For example, $S_1 = [j:j:1]$ and $S_2 = [j+1:j+1:1]$, then $S_1(p) -_{min} S_2(\overline{p})$ is close to $S_1(p)$ except that the lower bound will be increased by $D+1$ if $D+1 > 0$, or the upper bound will be reduced by $D+1$ otherwise. So a pair of such offsets, $F = (f_l, f_u)$, are added to the specification of parametric sections in the form of $[L:U:s:F]$.

To compute $S(p) = S_1(p) \cup_{min} S_2(p)$, we extend the merging algorithm in section 3.2 by considering the offsets.

The last operation in computing $Inv_{max}(A, J_k, I, p)$ is "$-_{max}$":

$$Inv_{max}(A, J_k, I, p) = Use_{max}(A, J_k, I, p) -_{max} Ref_{min}^{down}(A, J_k, I, p)$$

```
invalidate(&A(0,0), sizeof(real)*N*N)
DO 10 i = 1, N
   invalidate(&A(i, i), sizeof(real)*(N-i))
   DOALL j = i+1, N
      A(j, i) = A(j, i)/A(i, i)
      DO k = i+1, N
         A(j, k) = A(j, k) - A(j, i)*A(i, k)
      END DO
      IF (j == (i+1))
      THEN update(&A(i+1, i+1), sizeof(real)*(N-i))
      END IF
   END ALL
END DO
update(&A(0, 0), sizeof(real)*N*N)
```

Figure 5: LU Decomposition after Applying CAS

We first check the offsets of parallel sections in Ref_{min}^{down}. If they intersect with Use_{max}, the offsets should be in the invalidation set. We then find out the difference ("$-_{max}$") between sections in Use_{max} and Ref_{min}^{down} without considering offsets, which is similar to computing "$-_{min}$".

Computing Upd requires examining whether the data written in the current parallel loop will be read by other processors: $\sum_{i=I+1}^{N}\sum_{j=k+1}^{n} Use(A, J_j, I, \overline{p}))$ (see Eq. 5). It is not practical to compute the summation over all the sequential iterations. To simplify the task, we classify the sections in $Use(A, J_j, i, \overline{p})$ into two groups based on whether they contain the outermost sequential loop index variable. All the sections without the outermost sequential loop index variable are moved out from the summation. For a section with the sequential index variable, we determine whether the section increases or decreases with the index variable.

The last step is to generate code for invalidations and updates. For the invalidation sections defined by offsets, we merge them to obtain the largest offsets for the current parallel loop, and insert *invalidate* immediately before the parallel loop. For a sequential section in Inv, invalidating the section is necessary on every processor. For a sequential section in Upd, *update* is inserted inside of the parallel loop, but a guard condition is added before the *update* for each processor to check whether the sequential section is modified in the iterations of the parallel loop scheduled on the processor.

Figure 5 gives the parallel LU decomposition after applying our CAS method. The read sections in iteration i include

$$A[j:j:1][i:N:1], \text{ and } A[i:i:1][i:N:1]$$

where the second one is the pivot row. Clearly, the first section is in the reference set of iteration $i-1$, so only the pivot row may need to be invalidated. For *update*, the write section is $A[j : j : 1][i : N : 1]$ in iteration i, where all rows except the first in the section will be written by the same processor in iteration $i + 1$. This row is thus updated immediately within the parallel loop.

5 Cache Line False Sharing

The cache line size on most machines is much larger than a single word. When a cache line is written by several processors within a parallel loop, in the absence of corrective measures, one processor will successfully write its updated value in the cache line into main memory by an *update* operation, and other writes will be lost. Therefore, cache line false sharing causes incorrect results. Several schemes such as SCTV and FSI avoid false sharing by using a write-through cache policy, which makes writes slower. We reduce false sharing in CAS as follows:

1. All dimension sizes are enlarged to be a multiple of the cache line size.
2. If there is at least one dimension in which the index expressions do not contain the parallel loop index, this dimension is permuted to the last dimension to order to avoid false sharing.
3. If the parallel loop index appears in every dimension, but there is one of the dimensions where the index expressions are the same, then this dimension is permuted to the last dimension so that the parallel loop can be partitioned to ensure writes be cache line aligned on every processor.
4. If the alternatives of 2 and 3 fail, we identify false shared pages (cache lines) and set these pages to the write-through policy.

6 Evaluation of CAS

The CAS, SCTV, and FSI schemes were applied to three applications, LU decomposition, Successive Over Relaxation (SOR), and the Conjugate Gradient program (CG). These applications were then implemented on the Hector multiprocessor [27].

Hector is a hierarchical shared-memory multiprocessor with 16 MHz MC88100 CPU's with 16 K byte data and 16 K byte instruction cache, and 4 M byte local memory. The interconnection is 4 processors per bus connected by a hierarchy of rings. The cache line size is 16 bytes. It provides the *invalidate, update* and *flush* cache operations for software to maintain coherence. The costs of *invalidate* and *update* on Hector are given in Figure 6. The invalidation costs are relatively insensitive to data size, but the update costs increase with the data size. Our experiments were conducted on a 16-processor, 4-station configuration. Cache hit ratio and speedup were used as performance metrics.

The FSI method uses an extra bit per cache line to exploit locality between iterations of a parallel loop and some additional hardware to invalidate in constant time. On Hector,

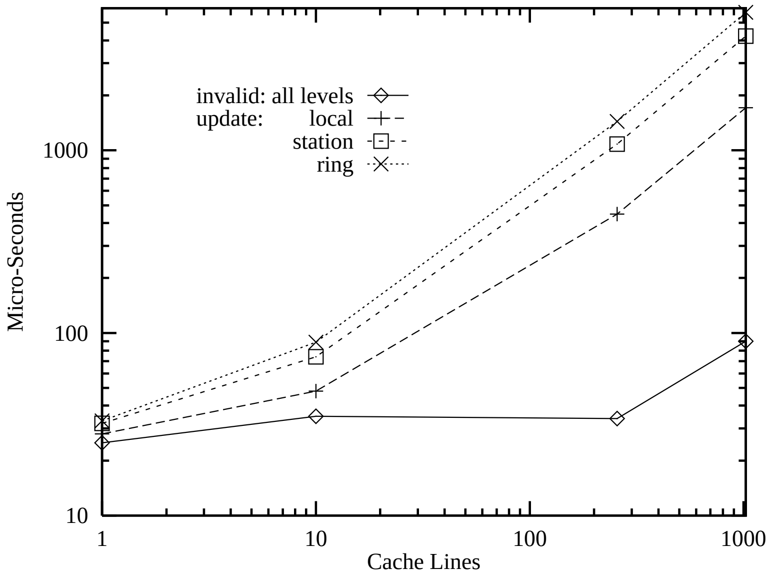

Figure 6: Cost of Cache Coherence Primitives (in μs)

invalidating the entire cache takes only 90 μs, so that approach can be used as the invalidation for FSI. Two extreme cases were investigated: the worst case is the No Cache version in which arrays are set to uncachable, and the optimal case is the No Coherence version in which data in caches will never be written back to memory except for evictions (so the results of the programs are incorrect). We first present cache hit ratios for the three applications under different cache coherence schemes, then discuss their speedups on Hector.

6.1 Cache Hit Ratios

Cache foot prints of these applications were collected using Hector's system calls and cache hit ratios were estimated from the collected data and the spatial locality.

LU Decomposition. Figure 7 gives the cache hit ratios for LU decomposition with a matrix size of 100×100. At each elimination step, the pivot row is used by all processors to reduce their own rows. So the pivoting row in the ith elimination step is reused $(N-i)/P$ times within parallel loops by every processor. All three methods can utilize such locality in cache. The cache hit ratios for pivoting rows are shown in Figure 7 (a). Other rows are used only once in each elimination step, but reused between elimination steps. Both FSI and SCTV fail to exploit such temporal locality across parallel loops, resulting in a cache hit ratio of .50 from spatial locality. By applying parametric section analysis, we can recognize sharing patterns and fully utilize the cache reuse so that the CAS scheme delivers a higher cache hit ratio. The 12 peaks indicate that there are 12 phases of cache conflict between pivoting rows and others when eight processors are used for LU decomposition of the matrix of the size 100×100. When 16 processors are used, the number of peaks is reduced to six (Figure 7 (b)).

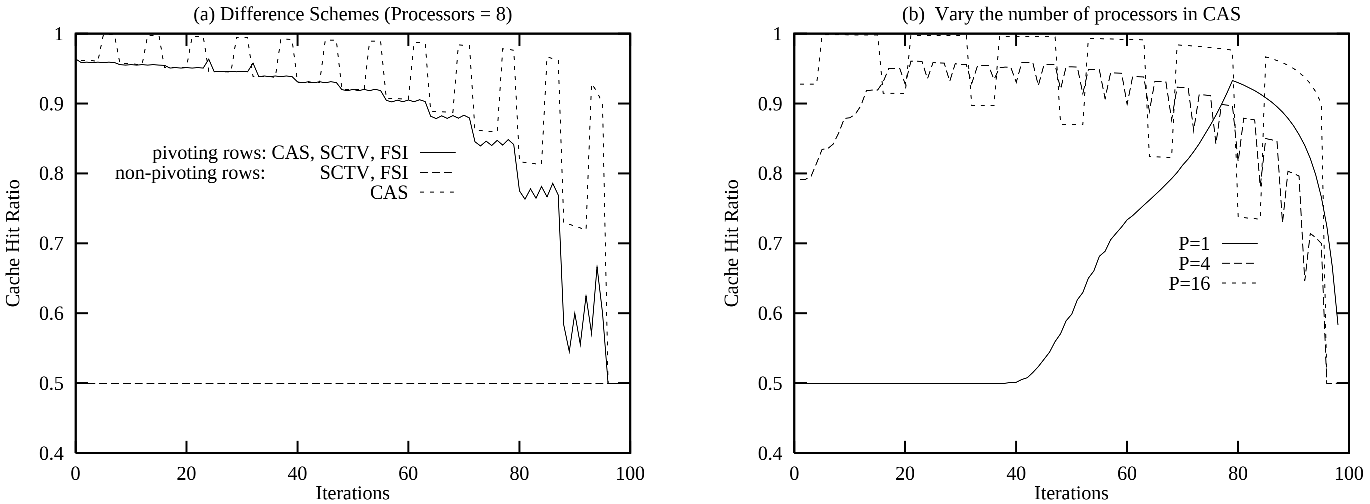

Figure 7: Cache Hit Ratios in LU Decomposition

Figure 7 (b) shows the cache hit ratios for non-pivoting rows for from 1 to 16 processors. When fewer processors are used, the caches cannot accommodate the working set (the current pivoting row and the currently used non-pivoting rows). Figure 7 (b) shows that, in the first stages of the LU decompositions, the cache hit ratios for non-pivoting rows is close to zero when the working set is much larger than the cache size, and the cache hit ratios are increased when the working set size is decreased.

SOR. Figure 8 gives cache hit ratios for square matrix sizes of 128 and 256. In the FSI and SCTV schemes, only spatial locality can be exploited for both sizes, so the cache hit ratio is estimated as 0.5. In the CAS scheme, data can be reused across parallel loops. However, the cache hit ratios depend on whether the cache size is large enough to hold the working set. For a matrix size of 128, the working set is 16K when eight processors are used. So the cache hit ratio is 0.5 for fewer than eight processors. The cache hit ratios decrease when the number of processors is greater than eight, because the remote memory access ratios increase with the number of processors. For the matrix size of 256, the cache hit ratio is also .5 for fewer than 16 processors since the working set is larger than the cache size. On 16 processors, the cache hit ratio for 256 is higher than that for 128, because the remote memory access ratios decrease with the problem size.

CG. Figure 9 gives cache hit ratios for vectors used in CG. It does not include cache hit ratios for the sparse matrix since the matrix is read-only and causes no cache coherence problems. Accesses to these vectors within parallel loops retain spatial locality, so both SCTV and FSI obtain a cache hit ratio of 0.5. The CAS method achieves a higher cache hit ratio by exploiting temporal locality across parallel loops.

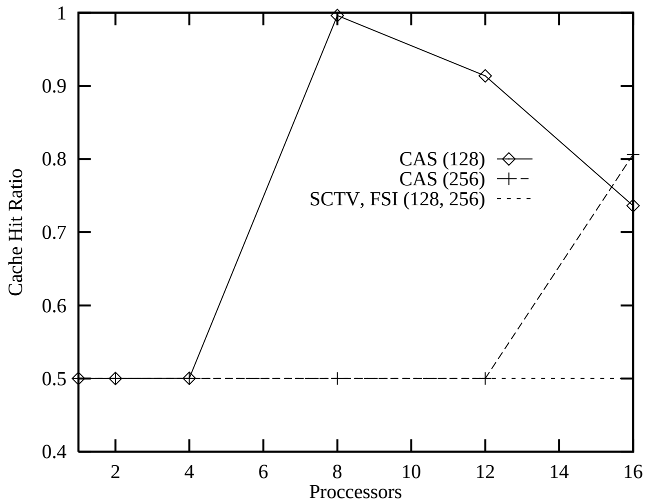

Figure 8: Cache Hit Ratios in SOR

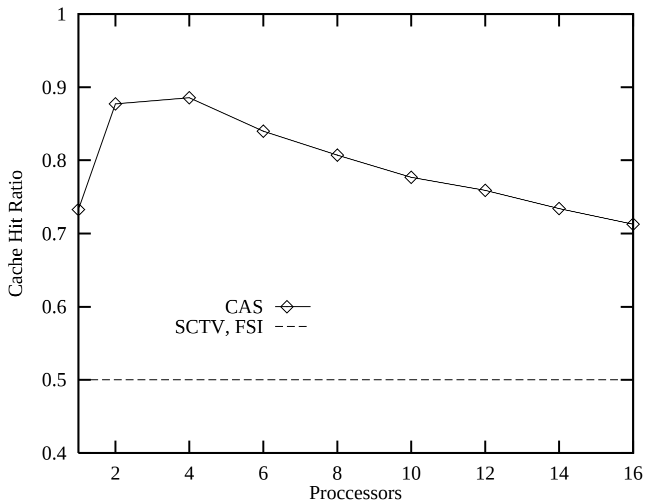

Figure 9: Cache Hit Ratios in CG

6.2 Speedups

Figures 10, 11, and 12 show speedups attained by the three applications. The speedups are calculated relative to the execution time of the sequential version.

LU Decomposition. The speedups of LU decomposition are shown in Figures 10. The No Cache version obtains limited speedups (less than two) for all the problem sizes. Both FSI and SCTV improve performance significantly over the No Cache version because they exploit spatial locality. Although they have the same cache hit ratio, the speedup in FSI is slightly higher because the overhead of invalidating the entire cache in FSI is less than that of invalidating individual rows in SCTV. The CAS version retains data locality across the elimination steps, thus outperforming FSI and SCTV. When the problem size increases, the performance of CAS becomes close to that of the No Coherence version, which is the upper bound. Once the problem size is sufficiently big so that the cache cannot hold the working set (for example, a matrix size of 300), all these schemes perform similarly, since no temporal locality can be exploited for non-pivot rows across the elimination steps.

SOR. Speedups of SOR for matrix sizes of 128 and 256 are shown in Figure 11. For the size of 128, all schemes except No Cache perform equally well when the number of processors is less than eight, because no temporal locality across parallel loops can be found in the cache. When the number of processors is more than eight, the working set can be accommodated in the cache under the CAS scheme, so CAS outperforms FSI and SCTV. For the size of 256, the working set is larger than the cache, thus all the schemes except No Cache obtain similar speedups.

CG. In CG, the sparse matrix is read-only, so this matrix is cached without the coherence problem in all the versions including No Cache. However, the No Cache version still performs poorly (see Figure 12). Although FSI and SCTV retain only spatial locality in this application, they perform much better than the No Cache version. The CAS scheme has speedups close to the upper bound given by the No Coherence version for all numbers of processors.

7 Summary

In this paper, we have proposed the Cache Affinity-based Software cache coherence scheme. This scheme exploits data locality across parallel loops, but it does not require modifications to the cache memory hardware. We have proposed parametric sections to represent cache affinity. We have developed detailed algorithms for computing cache affinity among parallel loops and inserting *invalidate* and *update* operations for cache coherence.

CAS has been evaluated on the Hector multiprocessors by comparing it with other schemes. The experimental results have shown that:

- Caching data is very important to performance. All three applications studied perform poorly without using caches.
- With a small cache size, simple software cache coherence schemes perform well since data reuse in small caches is dominated by spatial locality, which can be easily achieved by these simple schemes.

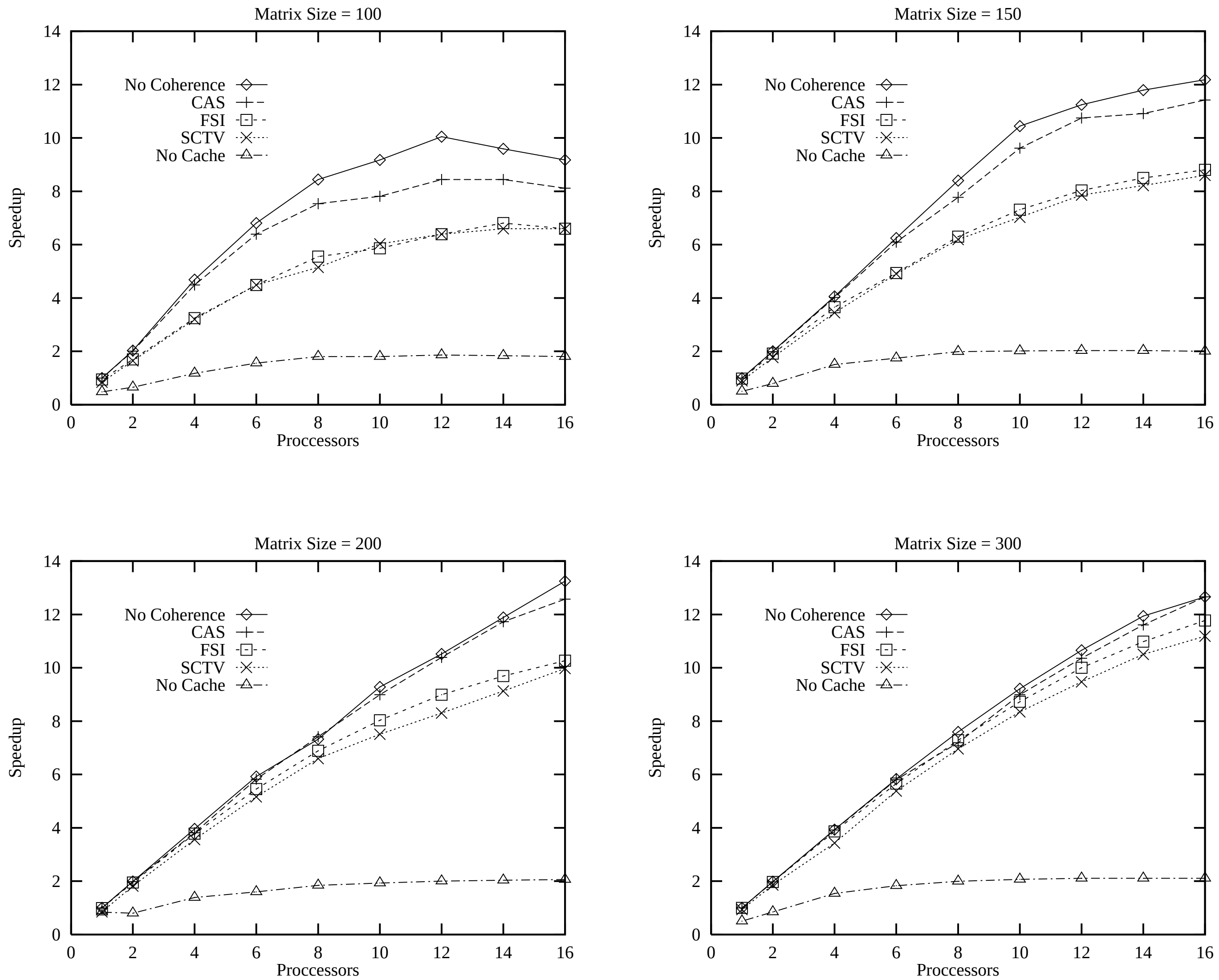

Figure 10: Speedups of LU Decomposition

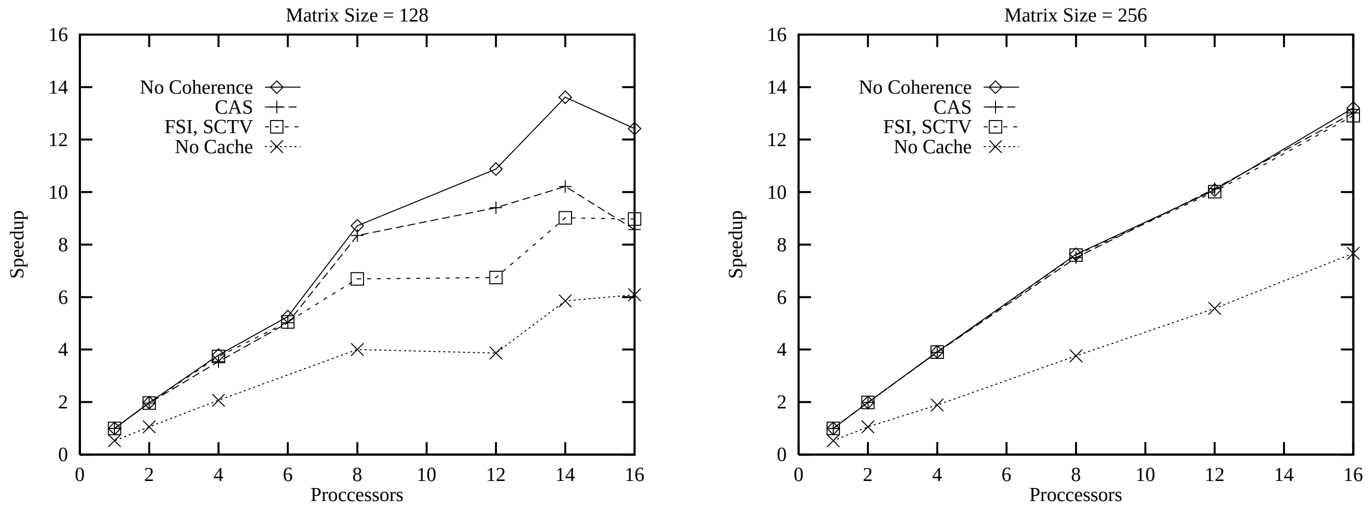

Figure 11: Speedups of SOR

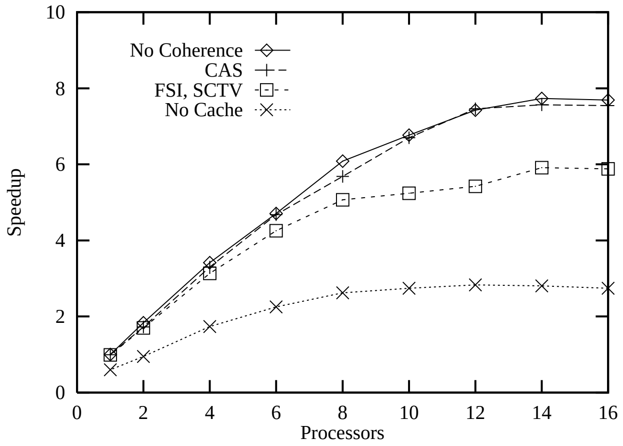

Figure 12: Speedups of CG

- When a large portion of the working set can be accommodated in the cache, temporal locality affects performance significantly. CAS exploits such cache affinity and achieves higher cache hit ratios, thus outperforming other schemes.

References

[1] Anant Agarwal, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, and Dan Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Architectures*. Kluwer Academic Publishers, 1991.

[2] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[3] Vasanth Balasundaram. A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor. *Journal of Parallel and Distributed Computing*, 9:154–170, 1990.

[4] BBN Advanced Computers Inc. *Inside the TC2000 Computer*, 1990.

[5] William J. Bolosky, Michael L.Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proc. of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, April 1991.

[6] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 152–64, October 1991.

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

[8] Hoichi Cheong. Lift Span Strategy – A Compiler-Based Approach to Cache Coherence. In *Proc. International Conference on Supercomputing*, 1992.

[9] Hoichi Cheong and Alexander Veidenbaum. Cache Coherence Schemes with Fast Selective Invalidation. In *Proc. of the 15th Annual International Symposium on Computer Architecture*, pages 299–307, 1988.

[10] Hoichi Cheong and Alexander Veidenbaum. Stale Data Detection and Coherence Enforcement Using Flow Analysis. In *Proc. of the International Conference on Parallel Processing*, pages 138–145, 1988.

[11] Alan L. Cox and Robert J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA M ultiprocessor: Experiences with PLATINUM. In *Proceedings of 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[12] Ron Cytron, Steve Karlovsky, and Kevin P. McAulife. Automatic Management of Programmable Caches (Extended Abstract). In *Proc. of the International Conference on Parallel Processing*, pages 229–238, 1988.

[13] Ervan Darnell, John M. Mellor-Crummey, and Ken Kennedy. Automatic Software Cache Coherence through Vectorization. In *Proc. International Conference on Supercomputing*, 1992.

[14] Elana D. Granston and Alexander V. Veidenbaum. Detecting Redundent Accesses to Array Data. In *Proc. of Supercomputing'91*, pages 854–865, 1991.

[15] Paul Havlak and Ken Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[16] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. In *Proc. Supercomputing'91*, pages 86–100, Albuquerque, NM, Nov. 1991.

[17] J. Konicek, T. Tilton, A. Veidenbaum, C. Q. Zhu, E. S. Davidson, R. Downing, M. Haney, M. Sharma, P. C. Yew, P. M. Farmwald, D. Kuck, D. Lavery, R. Lindsey, D. Pointer, J. Andrews, T. Beck, T. Murphy, S. Turner, , and N. Warter. The Organization of the Cedar System. In *Proc. of the International Conference on Parallel Processing*, pages Vol.I, 49–56, 1991.

[18] Kendall Square Research. *KSR1 Principles of Operation.* Waltham, MA, 1991.

[19] Kendall Square Research. *KSR Fortran Programming.* Waltham, MA, 1993.

[20] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, 1990.

[21] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Department of Computer Science, Yale University, September 1986.

[22] Sang Lyul Min and Jean-Loup Baer. A Timestamp-based Cache Coherence Scheme. In *Proc. of the International Conference on Parallel Processing*, pages Vol I: 23–32, 1989.

[23] Sang Lyul Min and Jean-Loup Baer. A Performance Comparison of Directory-based and Timestamp-Based Cache Coherence Schemes. In *Proc. of the International Conference on Parallel Processing*, pages Vol I: 305–311, 1990.

[24] Edward Rothber, Jaswinder P. Singh, and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[25] Harjinder S. Sandhu, Benjamin Gamsa, and Songnian Zhou. The Shared Regions Approach to Software Cache Coherence on Multiprocessors. In *Proc. of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–238, May 1993.

[26] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, 23(6):12–24, 1990.

[27] Zvonko G. Vranesic, Michael Stumm, Ron White, and David Lewis. The Hector Multiprocessor. *Computer*, 24(1), January 1991.

[28] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, 1989.