Eliminating Stale Data References through Array Data-Flow Analysis

Lynn Choi
University of Illinois at Urbana-Champaign
Center for Supercomputing R & D
1308 West Main Street
Urbana, IL 61801-1351
lchoi@csrd.uiuc.edu

Pen-Chung Yew
University of Minnesota
Department of Computer Science
200 Union Street, SE
Minneapolis, MN 55455-0519
yew@cs.umn.edu

Abstract

In this paper, we develop a compiler algorithm for detecting references to stale data in shared-memory multiprocessors. The algorithm consists of two key analysis techniques, stale reference detection and locality preserving analysis. While the stale reference detection finds the memory reference patterns that may violate cache coherence, the locality preserving analysis minimizes the number of such stale references by analyzing both temporal and spatial reuses. By computing the regions referenced by arrays inside loops, we extend the previous scalar algorithms [7, 9] for more precise analysis. We have implemented the algorithm on the Polaris parallelizing compiler [19], and using execution-driven simulations on Perfect Perfect benchmarks we demonstrate how unnecessary cache misses can be eliminated by the automatic stale reference detection.

1. Introduction

Directory-based hardware coherence protocols have been studied to enforce the cache coherence in several large-scale research machines [2, 17]. However, due to complex and expensive hardware cache/directory controllers required by such schemes, most of recent MPP systems do not provide hardware coherent caches, which prohibits the usefulness of caches for remote memory access. For example in Cray T3D, lack of cache coherence mechanism forces each cache line loaded by a remote read not to be cached (by an uncacheable load instruction) or to be flushed [16].

Several compiler-directed coherence schemes have been proposed [6, 8, 9, 12, 18]. In this approach, cache coherence is maintained locally without directory hardware, avoiding the complexity and the overhead associated with the hardware directories. Although the performance of such schemes have been demonstrated through simulations, most of those studies assume either perfect compile-time analysis or analytic models without real compiler implementations [1, 18]. It is still unknown how effectively the compiler can detect potential stale references and what kind of performance can be obtained by using a real compiler.

In this paper, we develop and implement a compiler algorithm on Polaris [19] parallelizing compiler to test the feasibility and the performance of a compiler scheme, which can be easily incorporated to implement cache coherence for existing MPP systems that do not have hardware directories. Using such a software approach, we not only can reduce overhead in cache coherence management, such as cache flush or barrier operations, but also can decrease the frequency of remote memory operations to take full advantage of on-chip caches.

We use a combination of interval and data-flow analysis techniques to determine memory reference patterns which can lead to stale data accesses. To obtain more precise array access information, we compute the array region referenced by each array reference. For symbolic analysis involving arrays, we use the gated single assignment (GSA) [3].

Two key analysis techniques are used to identify potential stale references: (1) *stale reference detection*, and (2) *locality preserving analysis*. The stale reference detection algorithm finds memory reference sequences that may violate cache coherence by using a def-use chain analysis. The algorithm is based on a new condition for a stale access. It considers implicit RAW (read-after-write) and WAW (write-after-write) dependences caused by multi-word cache lines (see section 1.1). To further refine reference marking, two locality preserving analysis techniques are used to exploit both temporal and spatial reuses[24] in a program. To refine reference marking for both group temporal and spatial reuses, we mark the initial occurrence of upwardly-exposed uses in a program region for potential stale data references. In addition, we use a code generation technique, *guarded execution*, to further remove unnecessary cache misses at runtime for self temporal and spatial reuses as well as to further utilize array region information computed at compile time. All these compiler algorithms have been implemented in the Polaris parallelizing compiler, and experimentation results on Perfect benchmarks [4] are discussed.

1.1. Stale reference condition

Memory event ordering Let's first define the ordering of events which leads to a stale reference. The following sequence of events [23] creates a stale reference at runtime: (1) Processor P_i reads/writes to a memory location *x* at time T_a; (2) Another processor P_j ($j \neq i$) writes to *x* at time T_b ($> T_a$); (3) Processor P_i reads the copy of *x* in P_i's cache at time T_c ($> T_b$). The event (1) will create an initial cache copy of *x* in P_i, and the event (2) will create a new copy of *x* in P_j's cache, making the copy in P_i's cache stale. The following read of *x* by P_i becomes a stale reference.

Stale reference sequence In our parallel execution model, the execution of a parallel program is viewed as a sequence of *epochs*. An epoch is either a parallel loop or a serial section of the code between parallel loops. Assuming only DOALL types of parallelism (no dependences among concurrent tasks), the memory events (1) to (3) should occur in different epochs. Otherwise, there are dependences among concurrent tasks. To detect stale data reference from a source program, the previous compiler algorithms [7, 9, 23] look for the following memory reference patterns that consist of (a) a read or a write, (b) one or more epoch boundaries, (c) a write, (d) one or more epoch boundaries, and (e) a read.

However, with multi-word cache lines, there can be implicit dependences due to false sharing [13]. Let's look at the program example in Figure 1(a) and the corresponding memory events (Figure 1(b)) at runtime. The figure also shows the content for each cache. It assumes two-word cache lines and a write-allocate policy. All caches are empty at the beginning of epoch 1. The read reference to Y(2) by processor 1 in epoch 3 is a stale data reference since the cache copy is read in epoch 1 but a new copy is created by processor 2 in epoch 2. Similarly, the same memory reference pattern makes a read reference to Y(3) by processor 2 in epoch 3 stale. However, note that there are three additional stale data references, which do not conform to the above memory reference pattern. Let's first look at the write reference to X(1) by processor 1 in epoch 1. The reference will causes a cold start miss, and will load the entire cache line from memory, creating a copy for both X(1) and X(2). This causes implicit RAW(read-after-write) or WAR(write-after-read) dependences when they are written in epoch 2. Therefore, the following read reference to X(2) by processor 1 in epoch 3 becomes a stale data reference. The read-write dependence among the variables in the same cache line creates a false sharing effect. These implicit dependences can also occur between concurrent tasks in the same epoch. For example, in epoch 2, the write reference to Z(2) by processor 1 will cause a cache miss and brings the entire cache line, including a copy of Z(1). However, in the same epoch, processor 2 writes to Z(1), making the cache copy in processor 1 stale. The following reference to Z(1) by processor 1 in epoch 3 becomes a stale reference. A similar case happens for the reference to Z(2). Therefore, to handle the line aliasing with multi-word cache lines, the sequence of events *(a) a write, (b) one or more epoch boundaries, and (c) a read* is sufficient to create a potential stale reference. We call this sequence of events a *stale reference sequence*.

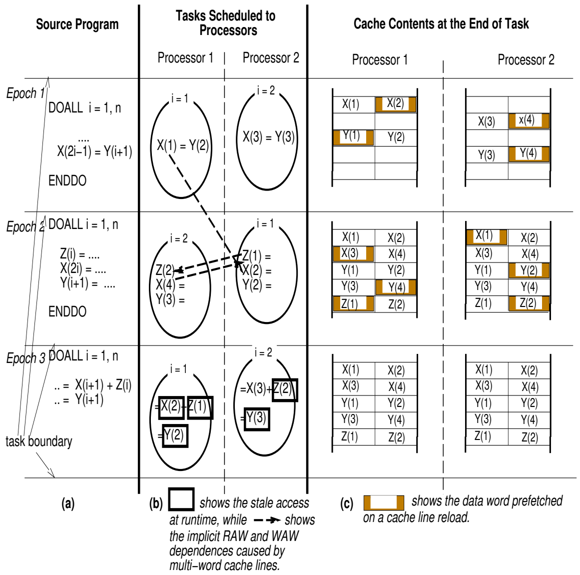

Figure 1. Stale accesses resulting from implicit dependences with multi-word lines.

1.2. Array data-flow analysis

Previous compiler algorithms [7, 9] treat an entire array as a single variable, which leads to conservative estimation of potential stale references. Let's look at the previous example in Figure 1(a) and assume that the iteration space of loop in epoch 3 is now from n+1 to 2n instead of 1 to n. Treating an array as a single variable, the compiler will mark all the read references in epoch 3 as potentially stale since the variables X, Y, and Z are all modified in previous epochs. However, a careful array flow analysis will mark read references to Y and Z as safe because the array regions accessed in epoch 3 and the regions accessed in the previous epochs are distinct. In a scalar analysis, even a write to a single element of an array is interpreted as a write to the entire array. This conservative scalar analysis often creates unnecessary cache misses either through invalidations or by redundant accesses to main memory [6, 8, 9]. These unnecessary memory accesses can be avoided by a more precise analysis.

In the following, we will demonstrate how array access information can refine the stale reference detection. First, we describe our framework for array data-flow analysis such as GSA and subarray descriptors in section 2. Then, in section 3, we present two key compiler techniques: the stale reference detection and the locality preserving analysis. In section 4, we discuss the results of applying both scalar and array flow analysis on Perfect benchmarks and show their performance using execution-driven simulations. Section 6 concludes the paper.

2. A framework for array data-flow analysis

2.1. Gated Single Assignment (GSA) form

To perform effective array analysis, the symbolic manipulation of expressions is necessary since the computation of array regions often involves the equality and comparison tests between symbolic expressions. static single assignment (SSA) [11] is a representation of a program in which each use of a variable is reached by exactly a single definition of the variable. It allows us to track the value of a variable by its name. Gated single assignment (GSA) introduces three types of pseudo-assignment functions, which are extensions of the ϕ functions used in SSA:

- γ**(cond, value1, value2)** : for ϕ function located immediately after an IF statement. It includes the predicate of the IF statement as the first parameter.

- μ**(value1, value2)**: for ϕ function at the head of a DO loop which merges the values coming from the outside (value1) and the inside (value2) of the loop.

- η**(cond, value)**: for ϕ function at the exit of a loop, which selects the last value produced by a μ function. Since the μ function at the header of the loop can be used to find the last value of the loop, the η function is omitted in our GSA form.

By transforming a source program into its GSA form, we can treat arrays with different reference regions as different symbolic variables. We use two symbolic analysis techniques. Global symbolic forward substitution propagates information until it terminates at the confluence points in the control flow graph. A backward demand-driven symbolic analysis is used next to compute values and conditions across the confluence points of the control flow graph [22].

In addition to the above 3 functions, another function called α(array, subscript, value) [11] is used to replace the array assignment statement. The semantics of the α function is that a part of the array will take the value for the specified subscript while the rest of the array will remain as before. This representation maintains the single assignment property for the arrays. Figure 2 shows an example program and its GSA form which will be used in this section.

```
     PROGRAM example                  PROGRAM example
     DOUBLE PRECISION x, y            DOUBLE PRECISION x, y
     INTEGER*4 i, num                 INTEGER*4 i, num
     DIMENSION x(200), y(200)         DIMENSION x(200), y(200)

S1   PRINT *, 'PROGRAM START'         PRINT *, 'PROGRAM START'
S2   XDOALL i = 1, 100, 1             XDOALL i1 = 1, 100, 1
                                        y1 = μ(y0, y2)
                                        x1 = μ(x0, x2)
S3     x(i) = i                         x2(i1) = α(x1, i1)
S4     y(i) = i                         y2(i1) = α(y1, i1)
S5   END XDOALL                       END XDOALL
S6   DO num = 1, 10, 1                DO num1 = 1, 10, 1
                                        y3 = μ(y1, y6)
                                        x3 = μ(x1, x7)
                                        i2 = μ(i1, i4)
S7     IF (15+5*num.LE.30) THEN         IF (15+5*num1.LE.30) THEN
S8      DO i = 1, 15+5*num, 1            DO i3 = 1, 15+5*num1, 1
                                          y4 = μ(y3, y5)
                                          x4 = μ(x3, x5)
S9       y(i) = x(i+(15+5*num))           y5(i3) = α(y4,
                                            x4(i3+(15+5*num1)))
S10      x(i+i) = x(i+(15+5*num))         x5(i3+i3) = α(x4,
                                            x4(i3+(15+5*num1)))
S11     ENDDO                            ENDDO
S12    ELSE                             ELSE
S13    XDOALL i = 1, 15+5*num, 1         XDOALL i5 = 1, 15+5*num1, 1
                                          x8 = μ(x3, x9)
S14      x(14+i+5*num) =                  x9(14+i5+5*num1) = α(x8, x8(
          x((-1)+i)+y(14+i+5*num)           (-1)+i5) +y3(14+i5+5*num1))
S15     END XDOALL                       END XDOALL
S16    ENDIF                            ENDIF
                                        y6 = γ(15+5*num1.LE.30, y4, y3)
                                        x6 = γ(15+5*num1.LE.30, x4, x8)
                                        i4 = γ(15+5*num1.LE.30, i3, i5)
S17  x(15+5*num) = (15+5*num)+1         x7(15+5*num1) = α(x6,
                                          (15+5*num1)+1)
S18  ENDDO                            ENDDO
S19  PRINT *, 'PRINT RESULT'          PRINT *, 'PRINT RESULT'
S20  DO i = 1, 200, 1                 DO i6 = 1, 200, 1
S21    PRINT *, x(i), y(i)              PRINT *, x3(i6), y3(i6)
S22  ENDDO                            ENDDO
S23  STOP                             STOP
S24  END                              END
```

Figure 2. A program example and its GSA form

2.2. Data structure for array references

The data sets propagated during the flow analysis are implemented as sets of data descriptors. For each memory reference, we associate a data descriptor D, which contains the following fields: (1) **name(D)**, (2) **subarray(D)**, and (3) **mark(D)**. The name field records the variable name. The subarray field represents the region of the array that is accessed by the array reference. It is initially set to the index expression of the reference, representing the array element accessed by the reference. When the reference has enclosing loops, it represents the array element accessed in a single iteration instance. After we aggregate over the iteration space of its enclosing loops, the subarray field of the reference represents the array region accessed by the reference in the range of the loops. The mark field is a single bit field to mark a descriptor propagated across an epoch boundary, and used exclusively for stale reference detection.

We define three binary operations: (1) **union** ("$\cup$, $\oplus$"), (2) **intersection** ("$\cap$, $\otimes$"), and (3) **difference** ("-, $\ominus$") on data descriptors. Two kinds of operators are defined for each binary operation. "$\cup$", "$\cap$", and "-" are called *must* operators. They take the smallest result of union, intersection, and difference when they are involved with unknown symbolic constants or when we need to approximate the result. On the other hand, "$\oplus$", "$\otimes$", and "$\ominus$" are called *may* operators. They assume the largest set for the result of their operations. Both type of operators have the same semantics for scalar references. Depending on the type of analysis, one version of each binary operator should be used for safe analysis.

In addition to the binary operations, one unary aggregate operation called **aggregate** ("$\int_{inner}^{outer}$") summarizes the region spanned by the subarray field by aggregating the index expression of the array reference over the iteration space of the enclosing loops. The operands *inner* and *outer* in the aggregate operator specifies the range of the enclosing loops. When the bounds are omitted, the innermost loop and the outermost loop enclosing the reference is used for the range. Because we only consider do loops, the aggregation is a relatively straightforward interpretation of the index and the boundaries of the loops [22].

These operators are called *subarray operators* and are applied to data descriptors with the same variable name. It will generate a single data descriptor of the variable by taking the subarray operations on their corresponding subarray fields. Subarray operators are also defined on descriptors of different variables, or sets of data descriptors. In this case, they perform subarray operations on each pair of the data descriptors of the same variable, or perform set operations on the data descriptors of different variables.

The subarray field represented in the GSA form allows us to treat different references to an array as different symbolic variables. For example, let's look at the read references to the variable X in statements S9 and S10. Since both memory references refer to the same variable with the same index expression, we know that both references always refer to the same element in every iteration of the enclosing loops. However, if any of the variables in the index expression is modified between the two statements, they may refer to different elements of the variable although both memory references have the same index expressions. Since GSA representation automatically renames the variable, we can determine whether the two array references are referring to the same element of X by comparing their index expressions. Similarly, by representing the subarray fields in the GSA form, we can perform subarray operations involving symbolic loop bounds.

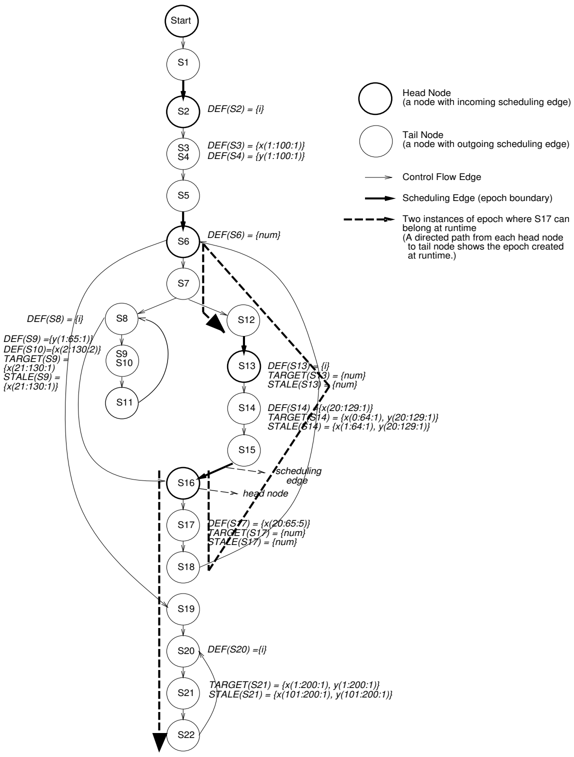

Figure 3. Epoch flow graph for the program example.

3. Algorithms

3.1. Stale reference detection

The algorithm for stale reference detection is an improved version of the algorithm shown in [9]. We refine the algorithm both to accommodate multi-word cache blocks as well as to refine analysis for array regions.

Definition: potentially stale A read reference *u* of a variable *v* in statement S1 is *potentially stale* if there is a directed path in the epoch flow graph from a definition *d* of variable *v* in statement S2 to S1 including at least one scheduling edge. When a definition reaches a read reference across at least one scheduling edge, there should exist the stale reference sequence (a) to (c), and the read reference to *v* should be marked as potential stale. We call it *potentially stale* since it may or may not lead to a stale reference at runtime.

```
Algorithm 3.1: Stale Reference Detection
Input:        DEF(S) and USE(S) for each statement S
Output:       STALE(S) for each statement S
Begin
 for each statement S do
     IN(S) ← ϕ;
     reset mark bits for all the definitions in DEF(S);
     OUT(S) ← DEF(S);
     STALE(S) ← ϕ;
 end for
 while changes to OUT(S) occur for any statement S do
       for each statement S do
           for each incoming edge p and predecessor P of S do
               if p is a scheduling edge
                  mark all the definitions in OUT(P);
               IN(S) ← IN(S) ⊕ OUT(P);
           end for
           for each use u of variable v in USE(S) do
               for each marked definition d of variable v in IN(S) do
                   if (subarray(u) ⊗ subarray(d)) ≠ ϕ then
                      subarray(u) ← subarray(u) ⊗ subarray(d);
                      STALE(S) ← STALE(S) ⊕ {u} where
                   end if
               end for
           end for
           OUT(S) ← OUT(S) ⊕ IN(S);
       end for
 end while
End
```

Figure 4. Flow analysis algorithm for stale reference detection. The USE(S) used in the algorithm can be replaced by TARGET(S) after the target reference detection.

For each statement S, we define the following sets. [1]

- $DEF(S)_{mark}$ is a set of definitions generated by S. Since all the definitions created in S are assigned the same mark, the mark field can be denoted collectively for S as $DEF(S)_{mark}$.
- *IN(S)* is a set of definitions reaching the beginning of S, IN'(S) = $\bigoplus_P OUT'$(P) where P is a predecessor of S.
- *OUT(S)* is a set of definitions reaching the end of S, $OUT(S) = IN(S) \oplus DEF(S)_{unmarked}$.
- *STALE(S)* is a subset of $USE(S)$ where there exists a definition reaching at the beginning of S across epoch boundaries, $STALE(S)$ = {use *u* of the variable *v*: *u* $\in USE(S)$ and $\exists$ a marked definition *d* of the variable *v* in IN(S) where (subarray(*u*) $\otimes$ subarray(*d*)) $\neq \phi$ }.

Figure 4 shows the detail algorithm for the stale reference detection. Because the epoch boundary information is essential in identifying potential stale references, we include parallel constructs in our control flow graph, called the *epoch flow graph*[9]. Figure 3 shows the epoch flow graph and STALE(S) computed by the algorithm for the program example in Figure 2.

[1] Because of the processor reassignments at each epoch boundary, a definition in an epoch can reach any following epoch without being *killed*. Therefore, the notion of *kill* used in the traditional reaching definition algorithm is no longer valid.

Reuse	Description	Locality Preserving Analysis
self-temporal	a reference in a loop accesses the same location in different iterations	guarded execution
self-spatial	a reference in a loop accesses the same cache line in different iterations	
group-temporal	different references access the same location	upwardly-exposed uses
group-spatial	different references access the same cache line	

Figure 5. Reuses and corresponding locality preserving analysis techniques.

3.2. Locality Preserving analysis

Not all the stale reference patterns lead to a stale reference at runtime. Since each potential stale reference implies a remote memory access instead of a cache hit, we should minimize the number of potential stale references marked at compile time by utilizing both the temporal and spatial locality in a program as much as possible. Wolf [24] discussed 4 different types of reuses in a loop as shown in Table 5. Note that the self reuses are inherently loop-specific while group reuses are not. Since we further want to take advantage of reuse in an epoch, we extend the range of group reuses beyond a loop nest. We use two techniques to exploit the locality: *upwardly exposed use* for group reuses and *guarded execution* for self reuses.

Upwardly exposed use An *upwardly-exposed use* of a variable in a block B is a use which is not preceded by any previous definition of the same variable in B.

Definition : target reference The first occurrence of the *upwardly-exposed uses* of a variable in an epoch is called a *target reference*. Note that only the target reference can be a potential stale access. The following references in the same epoch will not be stale because the cache copy will have been made up-to-date by the target reference if it was stale. Therefore, these references can be marked as regular reads or writes.

In our analysis, we are interested in finding only the first occurrence of the upwardly-exposed uses. To simplify the discussion, we simply refer the first occurrence of the upwardly-exposed uses as an upwardly-exposed use. A read following the first read will not be marked as an upwardly exposed use in our analysis. The detection of upwardly exposed uses of a scalar variable can be done precisely using the variable names. However, for array variables, a reference to an array may not access the same region as the one accessed by a preceding reference to the same variable. Hence, both references may have some upwardly-exposed uses in part of their access regions. In such a case, both array references should be marked as having upwardly-exposed uses. For those partially upwardly-exposed references, we keep only the region of the array that is actually upwardly-exposed and use the subarray information for later analysis. Figure 6 shows the marking algorithm for upwardly-exposed uses. To perform this analysis, we compute the following data sets from DEF and USE sets of each statement.

- *ACCESS_IN(S)* is a set of data descriptors that are unconditionally accessed at the beginning of S, ACCESS_IN(S) = $\bigcap_{predecessor\ P}$ ACCESS_OUT(P).
- *ACCESS_OUT(S)* is a set of data descriptors that are unconditionally accessed at the end of S, ACCESS_OUT(S) = $ACCESS\_IN(S) \cup DEF(S) \cup USE(S)$.

ACCESS_IN(S) denotes all the scalar variables and the subarrays that are referenced unconditionally before reaching the statement S. When there is more than one edge entering the statement S, we use "$\cap$ as the *confluence operator* to compute ACCESS_IN(S) since a reference to a variable belongs to ACCESS_IN(S) only when every predecessor of S contains the reference to the variable. Note that we use *must* operators to compute minimal sets.

To determine upwardly-exposed uses, we should not propagate the data sets through back edges since every reference in a cycle is covered by itself. In general, the set of values are propagated only through forward edges. A bypass edge from the end of DO loop to the next statement following the loop is created to propagate the values from the ENDDO statement to the outside of the DO loop. When the ENDDO statement is encountered, the array references inside the loop are aggregated over the iteration space of the DO loop. The detection of upwardly-exposed uses can be done in a single pass over the epoch flow graph.

A use R of a scalar variable in S is marked as upwardly exposed if the variable is not included in ACCESS_IN(S). For an array reference, we use the aggregate information to determine whether the region referenced by the use R has all been referenced already by the previous references to the array. This can be performed by checking the condition, subarray(R) $\leq$ subarray(ACCESS_IN(S)). To perform this analysis, we need to perform aggregation for array references in the data sets ACCESS_IN and ACCESS_OUT during the flow analysis. Initially, all the subarray fields in ACCESS_OUT(S) are initialized to the index expressions of the array references. When we finish a loop, we summarize

Algorithm 3.2: Target Reference Marking

```
Input:    Epoch flow graph G, DEF(S), and USE(S)
Output:   TARGET(S)
 Procedure initialize
  for each statement S do
      ACCESS_IN(S) ← φ;
      ACCESS_OUT(S) ← DEF(S) ∪ USE(S);
      initialize subarray fields in ACCESS_OUT(S)
         to index expression for array references;
      TARGET(S) ← φ;
  end for
Begin
 call initialize;
 while changes to ACCESS_OUT(S) occur for any statement S do
  for each statement S do
      if there exits an incoming scheduling edge
         call initialize; /* skip the propagation and reinitialize */
      ACCESS_IN(S) ← ⋂_{predecessor P} ACCESS_OUT(P);
      ACCESS_OUT(S) ← ACCESS_OUT(S) ∪ ACCESS_IN(S);
      for each upwardly-exposed read reference R in S do
          if R is a scalar reference then
             if !(R ⊆ ACCESS_IN(S)) then
                TARGET(S) ← TARGET(S) ∪ R;
          else /* for array references */
             if ((subarray(R) ⊖ subarray(ACCESS_IN(S)) != φ) then
                subarray(R) ← subarray(R) ⊖ subarray(ACCESS_IN(E))) ;
                TARGET(S) ← TARGET(S) ⊕ R;
             end if
          end if
      end for
      if (S == ENDDO statement)
         for each array reference R in ACCESS_OUT(S)
             that are inside the current loop L do
             R ← ∫_L^L R;
      end if
  end for
 end while
End
```

Figure 6. Target Reference Marking algorithm.

the loop with a single node, aggregating all the array references inside the loop over its iteration space. For multiply nested loops, we perform aggregation from the innermost loop to the outer loops level by level. The subarray field at a particular point of the analysis represents the region referenced in a single iteration instance of the outer loops that we are currently aggregating. Figure 3 shows the target references marked for the example program.

Since we represent the program in GSA form, during the analysis we must ignore all the extra definitions and uses created by the GSA representation. This includes all the references in the μ and the γ functions and the first argument of the α function. For simplicity, this is not shown in the algorithm in Figure 6.

Guarded execution A single data reference inside loops can generate multiple instances at runtime. Often only some of them actually lead to a stale data reference. The *guarded execution* can be used to refine the compile time reference marking for such cases. It can be used to exploit both self-temporal and self-spatial locality as well as to utilize the

```
DO i = 1, 200, 1
  PRINT b-read(X(i)), b-read(Y(i))
ENDDO
```

(a) code generated for potential stale reference

```
DO i = 1, 200, 1
  if (i <= 100) then
    PRINT X(i), Y(i)
  else
    PRINT b-read(X(i)), b-read(Y(i))
ENDDO
```

(b) code generation considering subarray information

```
DO i = 1, 200, 1
  if ((i <= 100) || (i mod 4 != 0))
    PRINT X(i), Y(i)
  else
    PRINT b-read(X(i)), b-read(Y(i))
ENDDO
```

(c) code generation exploiting self-spatial reuses

Figure 7. Code generation using guarded execution for statement S22 in the example.

subarray access information.

For program example shown in Figure 2, after stale reference detection and target reference detection, our algorithms find X(101:200:1) and Y(1:200:1) as potentially stale for read references in statement S22. Figure 7 shows the code generation after considering subarray information as well as self-spatial locality by using guarded execution. We assume a bypass read operation (b-read) is issued for each potential stale data reference. As shown in Figure 7(b), half of remote memory accesses can be eliminated by checking the array range information available by our algorithms. Since each memory access implies a significant latency, the performance gained by the guarded execution far outweighs the pipeline disruption caused by the additional branch operation. Moreover, note that only the first access to a cache line causes a cache miss at runtime. Assuming a 4-word cache line size, we can further eliminate 75% of the remaining memory accesses by exploding the self-spatial reuses as shown in Figure 7(c). A similar optimization can be done for self-temporal reuses.

Overall algorithm First, we transform the source program to a GSA form. Then, we construct an epoch flow graph, which contains the epoch boundary information. Next, we mark target references in each epoch, and compute regions of array that are referenced. Then, a stale reference detection algorithm determines the existence of stale memory reference sequences. If a stale reference sequence is found for a target reference, the reference is marked as potentially stale. We transform the program in its GSA form back to the original program with the reference marking information, and appropriate cache and memory operations are generated. Finally, the guarded execution technique can be used to further optimize the code generation.

3.3. Handling procedure calls

Until now, the algorithm presented assumes single program units, i.e. single procedures, and therefore single flow graphs. The presence of procedures and procedure calls introduces two complications in this analysis.

- **side effect** The execution of a procedure can have a side effect on variables at the point from which the procedure is called. The variables include the actual parameters given at the call site as well as the global parameters visible to both the calling procedure and called procedure. This in turn affects the analysis of the procedure called since any of the global variables and formal parameters can be read or written previously at the beginning of a procedure.
- **aliases** The call-by-reference parameter passing mechanism can cause two distinct variables to refer to the same memory location.

To address these problems, we use the following two methods: cache invalidation and selective inlining.

Cache invalidation at procedure boundary We can avoid the side effects of procedure calls by invalidating the entire cache after each call site. Since we start from a clean cache after the call, the side effect from the procedure need not be considered. We also need to invalidate the cache at the beginning of the procedure. Without such invalidation, we have to consider the definitions of all the global variables and the formal parameters before the procedure is called.

The second problem, aliasing due to parameter passing, cannot be resolved even if we invalidate the entire cache at procedure boundaries. This is because our analysis is no longer valid if there is aliasing. However, ANSI Fortran does not allow programs written with aliases. If aliases do occur in a source program, then we need to treat all the aliased variables as a single variable, or use the next technique, selective inlining.

Selective inlining By inlining the procedures, we can eliminate both the aliases and the side effects caused by the procedure calls. It also increases the cache locality since we can avoid frequent invalidations at the procedure boundaries. However, unconditional inlining can be prohibitively expensive due to code expansion. Since inlining is most effective for small procedures, we selectively inline a procedure whenever its size (determined in terms of references) is less than a threshold value.

4. Experimentation

We have implemented these compiler algorithms in the Polaris compiler [19]. Perfect benchmark suites [4] are chosen as our target applications. They are first parallelized by the Polaris compiler. Then, we process the parallelized source codes using both scalar and array flow analysis version of the algorithms given in section 3. Cache invalidate operations are inserted at the beginning of procedures and after each call site, while inlining is used selectively.

Simulation Execution-driven simulations [20] are used to verify the compiler algorithms and to determine the performance of the stale reference detection. All the simulations assume a 16-processor, distributed shared-memory architecture similar to Cray T3D. Each processor contains a 64-KB direct-mapped cache with 4-word cache lines. Simulated events include global and local memory accesses, parallel loop setup and scheduling operations, and synchronization operations. A static cyclic scheduling is used. The memory events are generated from a source code level simulation, which does not consider back-end compiler code generation and optimizations. For comparison studies, the following coherence schemes are simulated:

1. **Base Architecture (BASE)** This scheme does not cache shared data to avoid the cache coherence problem. All shared memory references are treated as cache misses. They will access main memory directly. This architecture is similar to Cray T3D with uncacheable loads for shared memory references and cacheable loads for private references.

2. **Compiler Scheme (CP)** In this scheme, potential stale references are identified at compile time, and for each potential stale reference, a bypass-cache memory operation is issued to avoid accessing stale cached data. The bypass cache operation will directly access main memory, and replace the cached data with up-to-date data from the main memory. The scheme shows the potential performance gain from our compiler algorithm.

Dynamic reference statistics Table 8 shows the dynamic reference counts for 6 Perfect benchmarks. Note that due to the automatic array privatization in Polaris, percentage of shared memory references are modest, ranging from 18.3% (QCD2) to 61.1% (TRFD). Note that except TRFD, which is fully-inlined, the potential stale references detected are on a per-procedure basis. For SPEC77 and QCD2, the improvement by the array data-flow algorithm is significant, eliminating 76.3% and 75.4% of potential stale references marked by the scalar algorithm. For other benchmarks, the additional improvement is modest.

4.1. Miss Rates

Impact of stale reference detection Figure 9 shows the miss rates of the compiler scheme using our array data-flow algorithm against the miss rates of the base architecture for the Perfect benchmarks. The miss rates are classified into sharing nonsharing misses. A sharing miss occurs when the address tag matches but either it has been invalidated or the reference is marked as potentially stale. All the other misses account for nonsharing misses as in a uniprocessor system. For all the 6 benchmarks chosen, sharing misses dominate

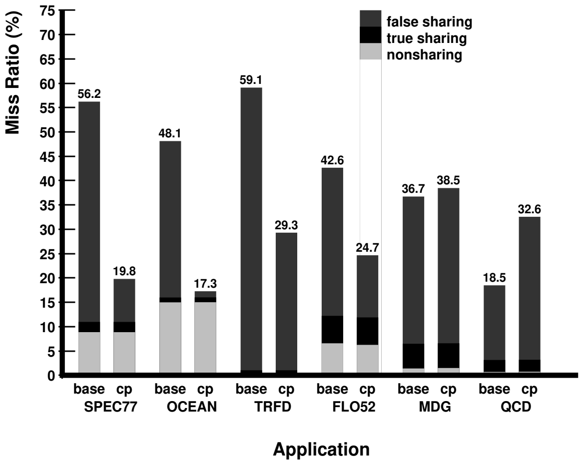

Figure 9. Reduction of cache misses after stale reference detection for Perfect benchmarks.

the overall miss rates. The sharing misses are further classified into true and false sharing misses. The true sharing misses are necessary misses to avoid cache coherence, while the false sharing misses show the unnecessary misses caused by the conservative compiler decisions.

With compile time analysis, a significant number of unnecessary sharing misses in the base architecture are eliminated, decreasing the miss rates by 17.9% (FLO52) up to 36.4% (SPEC77). This demonstrates how compile time stale reference detection can improve cache utilization by caching remote shared data. In particular, the stale reference detection eliminates most of the sharing misses for OCEAN. However for other benchmarks, there are still significant number of false sharing misses. This is due to two reasons. First, the stale reference detection algorithm must be conservative when it is involved with unknown compile time information such as branches, memory disambiguation, and scheduling. Second, the invalidations at procedure boundaries can sometimes increase the cache misses significantly especially when there are many small procedures in the critical path of the execution. Interestingly, the stale reference detection did not decrease the miss rate for benchmark MDG, and it even increases the miss rate significantly for QCD. In QCD, only 18.3% of memory references are to the shared data. However, invalidations at procedure boundaries remove local cached data unnecessarily, increasing the overall miss rates. This suggests that cache invalidations should be avoided as much as possible to effectively exploit the locality across procedural boundaries.

Impact of array analysis Figure 10 shows the improvement by the array data-flow algorithm over the scalar algo-

Program	Size (lines)	Read (%)	Write (%)	Shared (%) (total refs)	Private (%) (total refs)	Potential Stale Access (%)	
						Using scalar analysis	Using array analysis
SPEC77	5346	79.3%	20.7%	55.3% ($5.0*10^6$)	44.7% ($4.0*10^6$)	16.0%	3.8%
OCEAN	3101	72.6%	27.4%	34.9% ($3.2*10^6$)	65.1% ($6.0*10^6$)	0.10%	0.10%
FLO52	3182	73.5%	26.5%	39.4% ($2.8*10^6$)	60.6% ($4.3*10^6$)	13.1%	10.9%
MDG	1523	75.0%	25.0%	36.5% ($1.8*10^6$)	63.5% ($3.1*10^6$)	1.73%	1.71%
QCD2	3505	78.4%	21.6%	18.3% ($1.2*10^6$)	81.7% ($5.2*10^6$)	0.57%	0.14%
TRFD	572	80.4%	19.6%	61.1% ($5.9*10^6$)	38.9% ($3.8*10^6$)	32.7%	29.0%

Figure 8. Memory reference statistics for Perfect benchmarks.

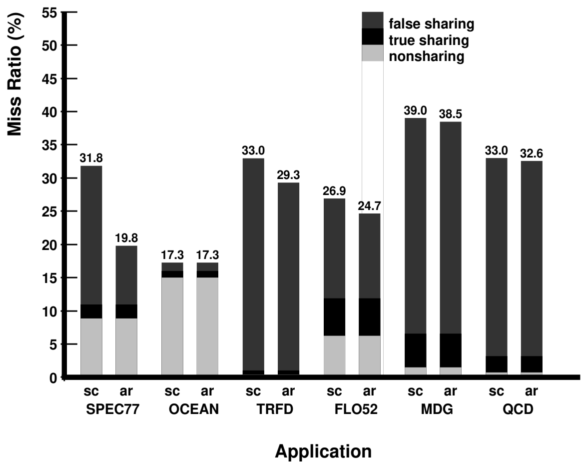

Figure 10. Array (ar) vs. scalar (sc) analysis.

rithm. For SPEC77, the array algorithm eliminates 37.8% of the false sharing misses of the scalar algorithm, decreasing the overall miss rates by 12.0%. For OCEAN, both scalar and array algorithms effectively eliminate most of the false sharing misses, and achieves very good cache utilization. For TRFD and FLO52, the improvement by the array algorithm is modest, eliminating 11.2% and 8.2% of the false sharing misses incurred by the scalar algorithm respectively. For MDG and QCD, the improvement is minimal. In these benchmarks, the overall false sharing miss rates are dominated by cache invalidations at procedure boundaries. Note that the performance improvement by the array algorithm is directly proportional to the percentage of the shared memory references and the relative decrease in the potential stale references marked by the array algorithm over the scalar one.

5. Previous work

There have been several studies on the compiler algorithms for stale reference detection [7, 9]. Among them, only Cheong and Veidenbaum included a compiler implementation study using Parafrase [7]. They pioneered a combination of scalar data-flow analysis and graph algorithms to find potential stale references. In [9], we proposed a simpler algorithm that also produces additional information for a new compiler-directed cache coherence scheme. That algorithm eliminates the graph construction phase of [7], but it may overestimate the potential stale references by summarizing information from multiple control flow paths. Both algorithms treat each array as a single variable.

There have been many studies on array data-flow algorithms. Granston proposed algorithms to detect redundant array references [15]. Feautrier [14] gave an algorithm to calculate them exactly. Pugh [21] developed some exact techniques that are substantially faster than Feautrier's. Our implementation is based on the regular section analysis[5], which is less accurate but allows large programs to be analyzed efficiently. In [10], we discuss how our algorithm can be applied to existing compiler-directed coherence schemes [6, 9].

6. Conclusion

Private caches can greatly improve the performance of large-scale shared-memory multiprocessors if they can be used to cache remote shared data. However, maintaining cache coherence for such systems is still a challenge. Hardware directories can be used to maintain coherence but require complicated hardware directory/cache controllers. In this paper, we develop and implement a compiler algorithm that maintains cache coherence without requiring such expensive hardware. The algorithm eliminates unnecessary remote memory accesses by detecting potential stale references at compile time. This algorithm can be incorporated into existing MPP systems that do not have hardware coherence support such as Cray T3D.

The algorithm uses two analysis techniques. First, stale reference detection finds the memory reference patterns that may violate cache coherence. The stale reference detection is based on a new condition for stale access. It considers both implicit RAW and WAW dependences caused by false sharing from multi-word cache lines. Second, locality preserving analysis minimizes the number of such references by analyzing both temporal and spatial reuses. For more precise array access information, we compute the regions referenced by arrays inside loops. For symbolic analysis for arrays, we use the GSA for demand-driven symbolic analysis. We implemented the algorithm on the Polaris parallelizing compiler, and demonstrate the performance gained by the algorithm through execution-driven simulations using Perfect benchmarks.

Our results show that, for most benchmarks, the compiler algorithm can improve cache utilization significantly by caching remote shared data. However, cache invalidations at procedure boundaries can sometimes increase false sharing misses unnecessarily because of the invalidation on both local and shared data. This suggests that it is important to exploit data locality across procedural boundaries. We also demonstrate how the array data-flow algorithm can further improve cache utilization over a scalar algorithm.

Acknowledgements The research described in this paper was supported in part by the NSF Grant No. MIP 89-20891 and MIP 93-07910. Special thanks go to David Poulsen at KAI for his numerous help for the development of execution-driven simulations. We also wish to thank the Polaris group members, Peng Tu, Jay Hoeflinger and Prof. David Padua for their guidance on the implementation of the array data-flow analysis on the Polaris compiler. Finally, we thank Lawrence Rauchwerger and IBM for providing us for the RS6000 clusters for simulations.

References

[1] S. V. Adve, M. D. Hill, and M. K. Vernon. Comparison of hardware and software cache coherence schemes. *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 298–308, May 1991.

[2] A. Agarwal and et al. The mit alewife machine: A large-scale distributed-memory multiprocessor. *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, 1991.

[3] R. Ballance, A. Maccabe, and K. Ottenstein. The program dependence web: a representation supporting control- data- and demand-driven interpretation of imperative languages. *Proceedings of the Conference on Programming Language Design and Implementation*, pages 257–271, June 1990.

[4] M. Berry et al. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[5] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

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

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

[8] T. Chiueh. A generational approach to software-controlled multiprocessor cache coherence. *Proceedings of the International Conference on Parallel Processing*, 1993.

[9] L. Choi and P.-C. Yew. A compiler-directed cache coherence scheme with improved intertask locality. *Proceedings of the ACM/IEEE Supercomputing'94*, November 1994.

[10] L. Choi and P.-C. Yew. Hardware and compiler support for cache coherence in large-scale multiprocessors. *Technical Report, University of Illinois, Computer Science Department, Ph.D. Thesis*, Feb. 1996.

[11] R. Cytron, J. Ferrante, and B. K. Rosen. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[12] E. Darnell, K. Kennedy, and Mellor-Crummey. Automatic software cache coherence through vectorization. *Proceedings of the International Conference on Supercomputing*, Nov. 1992.

[13] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988.

[14] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1), Feb. 1991.

[15] E. D. Granston and A. V. Veidenbaum. Detecting redundant accesses to array data. *Proceedings of the Supercomputing '91*, Nov. 1991.

[16] C. R. Inc. *Cray T3D System Architecture Overview*. Mar. 1993.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash computer. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

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

[19] D. A. Padua and et al. Polaris: A new-generation parallelizing compiler for mpps. *CSRD Rept. No. 1306. Univ. of Illinois at Urbana-Champaign*, June 1993.

[20] D. K. Poulsen and P.-C. Yew. Execution-driven tools for parallel simulation of parallel architectures and applications. *Proceedings of the Supercomputing 93*, Nov. 1993.

[21] W. Pugh and D. Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[22] P. Tu. Automatic array privatization and demand-driven symbolic analysis. *Technical report, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, Ph.D. Thesis*, 1995.

[23] A. V. Veidenbaum. A compiler-assisted cache coherence solution for multiprocessors. *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[24] M. E. Wolf. Improving locality and parallelism in nested loops. *Technical report, Stanford University, Dept. of Computer Science, Ph.D. Thesis.*, August 1992.