# Compiler Analysis for Cache Coherence:

Interprocedural Array Data-Flow Analysis and Its Impacts on Cache Performance[1]

**Lynn Choi**
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 West Main Street
Urbana, IL 61801
Email: *lchoi@csrd.uiuc.edu*

**Pen-Chung Yew**
4-192 EE/CS Building
Department of Computer Science
University of Minnesota
200 Union Street, SE
Minneapolis, MN 55455-0159
Email: *yew@cs.umn.edu*

**Abstract**

In this paper, we present compiler algorithms 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 [8] for more precise analysis. We develop a full interprocedural array data-flow algorithm, which performs both bottom-up *side-effect analysis* and top-down *context analysis* on the procedure call graph to further exploit locality across procedure boundaries. The interprocedural algorithm eliminates cache invalidations at procedure boundaries, which were assumed in the previous compiler algorithms [9]. We have fully implemented the algorithm in the Polaris parallelizing compiler [27]. Using execution-driven simulations on Perfect Club benchmarks, we demonstrate how unnecessary cache misses can be eliminated by the automatic stale reference detection. The algorithm can be used to implement cache coherence in the shared-memory multiprocessors that do not have hardware directories, such as Cray T3D [20].

**Keywords :** Compiler, Interprocedural Analysis, Data-Flow Analysis, Cache Coherence, Shared-Memory Multiprocessors.

[1] A preliminary version of some of this work appears in [12, 13, 15].

# 1 Introduction

Reducing memory latency is critical to the performance of large-scale parallel systems. Due to the temporal and spatial locality of memory reference patterns, private caches can eliminate redundant memory accesses, reducing both average memory latency and network traffic. Having multiple cached copies of a shared memory location, however, can lead to erroneous program behavior unless they are maintained coherent. Existing solutions for large-scale multiprocessors include hardware directory-based coherence protocols, which have been studied in many research machines [2, 21, 22]. Although these hardware schemes can precisely identify stale data by maintaining sharing information at runtime, they substantially increase the hardware cost for the directory storage and require complex directory and cache controllers.

As an alternative, compiler-directed techniques [7, 9, 10, 15, 11, 17, 23, 24, 31] can be used to maintain coherence. In this approach, cache coherence is maintained locally without directory hardware, thus avoiding the complexity and overhead associated with hardware directories. They usually require compile time analysis to detect possible stale data accesses and to invalidate stale cache entries. Although the performance of such schemes have been demonstrated through simulations, most of those studies assume either perfect compile-time analysis or analytical models without real compiler implementations [1, 6, 17, 23, 25, 26]. 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 both intraprocedural and interprocedural compiler algorithms on the Polaris parallelizing compiler [27] to test the feasibility and performance of compiler-directed coherence schemes. 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. Gated single assignment (GSA) [3] form is used to compute equality and comparison between the array regions involving symbolic expressions.

Two key analysis techniques are used to identify potentially stale references: (1) *stale reference pattern 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 considers implicit RAW (read-after-write) and WAW (write-after-write) dependences caused by multi-word cache lines (see section 2.1). To further refine reference marking, two locality preserving analysis techniques are used to exploit both temporal and spatial reuses[32] 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 potentially stale data references. In addition, we use a code generation technique, *guarded execution*, to further remove unnecessary cache misses at runtime, by utilizing self temporal and spatial reuses, as well as to further exploit array access information computed by the compiler.

We develop a full interprocedural array data-flow algorithm that performs bottom-up and top-down analysis on the procedure call graph to further exploit locality across procedure boundaries. First, the bottom-up *side effect analysis* eliminates side effects by summarizing the access information at each call site. Second, the top-down *context analysis* allows the context information of a procedure to be visible by passing the summary access information of its previous activation records. This two-pass analysis avoids redundant computation by performing incremental update of reference marking with a minimal number of computations per procedure. This algorithm eliminates cache invalidations, which are assumed by all previous compiler-directed coherence schemes, and allows the locality of programs to be preserved across procedure boundaries.

All of these compiler algorithms have been implemented in the Polaris parallelizing compiler, and experimentation results on Perfect Club benchmarks [4] are discussed. Execution-driven simulations are used to verify the compiler marking and to demonstrate the performance of automatic stale reference detection. The techniques developed here are general enough to be applicable to other compiler-directed coherence schemes [7, 9, 15].

# 2 Background

## 2.1 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 [31] creates a stale reference at runtime: (1) Processor $P_i$ reads or writes to a memory location $x$ at time $T_a$; (2) Another processor, $P_j$ ($j \neq i$) later 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 second write reference 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 (*parallel epoch*) or a serial section of the code (*serial epoch*) between parallel loops. Figure 1 shows a sample program and its epochs at runtime.

Assuming only DOALL types of parallelism (i.e. no dependences among concurrent tasks), 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 [8, 7, 31] 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. 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 that 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 causes a cold start miss and loads the entire cache line from memory, creating a copy

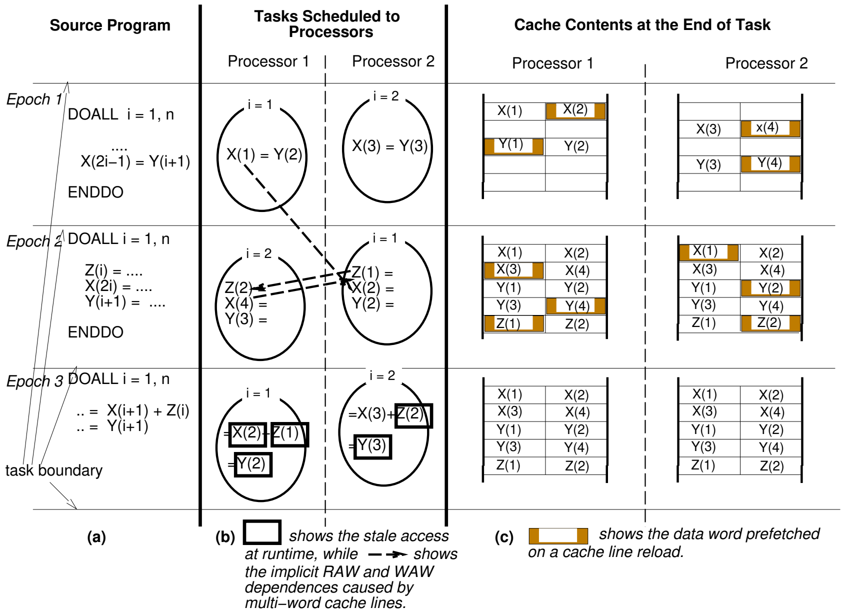



Figure 1: Stale accesses resulting from implicit dependences with multi-word lines.

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 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) will cause a cache miss and brings the entire cache line, including a copy of Z(1). However, in the same epoch, processor 1 writes to Z(1), making the cache copy in processor 2 stale. The following reference to Z(1) by processor 2 in epoch 2 becomes a stale reference. A similar case occurs in the reference to Z(2). Therefore, 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.*

## 2.2 Coherence mechanism and hardware support

**Software cache-bypass scheme (SC)** [2] Once all the potentially stale data references are identified by a compiler, cache coherence can be enforced if we guarantee that all such references access up-to-date data from main memory rather than potentially stale cache copies. This can be accomplished by using a bypass cache operation. This operation bypasses the cache to avoid accessing the potentially stale cached data, and replaces the cached data with the up-to-date copy by directly accessing the main memory.

[2] For a detailed description of these compiler-directed coherence schemes, please refer to the companion paper [14].

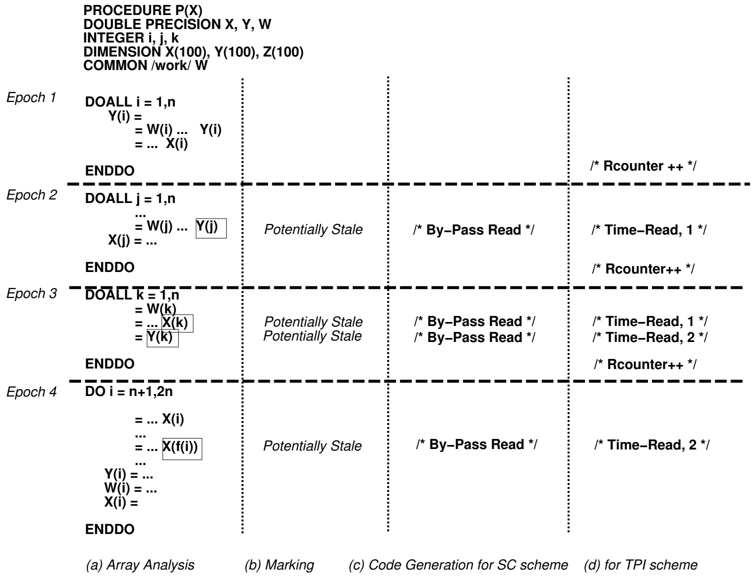



Figure 2: A program example of compiler marking.

**Two-phase invalidation scheme (TPI)** [2] This scheme improves the SC scheme by keeping track of runtime cache states locally. This is done by maintaining the current epoch number in a register ($R_{counter}$), and by associating every cache word with the epoch number when the cache copy is created. A special memory operation, called Time-Read, is used exclusively for a potentially stale reference to determine whether the cached data is really stale. To perform this operation, the compiler provides additional information called *offset*, which is the number of epoch boundaries between the current epoch and the epoch in which the data was last updated. The value of the "$R_{counter}$ - *offset*" denotes the previous epoch number when the last version of the data was created. And, the cache hit is determined by comparing the value with the timetag of the addressed cache copy. Figure 2 shows a sample program and the memory operations generated by a compiler for both SC and TPI schemes.

## 2.3 Array data-flow analysis

Previous compiler algorithms [8, 15] treat an entire array as a single variable, leading to a conservative estimation of potential stale references. Let's look at the program example in Figure 2(a). Treating an array as a single variable, the compiler will mark the read references to X in both epochs 3 and 4 as potentially stale since the variable X is modified in epoch 2 and read in the following epochs. However, a careful array flow analysis will mark these read references as safe because the array regions accessed in epoch 2 (from X(n+1) to X(2n)) and the following epochs (X(1) to X(n)) are distinct. Thus, the array elements addressed by the read references to X in epochs 3 and 4 have not been modified. Similarly, the read reference to Y in epoch 2 can be marked as safe with array analysis. In 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, through either invalidations or redundant accesses to the main memory [7, 9, 15]. These unnecessary memory accesses can be avoided by using a more precise analysis.

In the following, we will demonstrate how automatic stale reference detection can be used to maintain coherence by developing both intra- and interprocedural array data-flow algorithms. First, we describe our framework for array data-flow analysis, such as GSA and subarray descriptors, in section 3.1. Then, in sections 3.2 and 3.3, we present two key compiler techniques: stale reference detection and locality preserving analysis. In section 4, we discuss complications caused by procedure calls and develop a full interprocedural stale reference detection algorithm that performs 2-pass bottom-up/top-down analysis on the procedure call graph to fully exploit locality across procedural boundaries. In section 5, we discuss the results of applying the compiler algorithms on Perfect Club benchmarks and show their performance using execution-driven simulations. Section 7 concludes the paper.

# 3 Eliminating stale data references through array data-flow analysis

## 3.1 A framework for array data-flow analysis

In our analysis, we identify *the regions* of an array that are referenced by each array reference and treat them as distinct variables. The data values to be analyzed include scalar variables, *subscripted variables*, and *subarrays* [30]. A subscripted variable consists of an array identifier and a subscript expression, representing a single array element referenced. A subarray consists of a subscripted variable and one or more ranges for some of the indices in its subscript expression. A range is represented by a lower bound, a upper bound and a stride. The notion of a subarray is an extension to the *regular section* used in [5]. A subarray can represent a triangular region, a banded region, as well as a strip, grid, column, row, or block of an array.

### 3.1.1 Gated single assignment (GSA) form

To perform an effective array flow 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) [16] 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) [3] introduces three types of pseudo-assignment functions, which are extensions of the $\phi$ functions used in SSA:

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

- $\mu$**(value1, value2)**: for $\phi$ function at the head of a DO loop. It merges the values coming from the outside (value1) and the inside (value2) of the loop.

- $\eta$**(cond, value)**: for $\phi$ function at the exit of a loop. It selects the last value produced by a $\mu$ function. Since the $\mu$ function at the head of the loop can be used to find the last value of the loop, the $\eta$ function is omitted in our GSA form.

```
        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) =
                  x((-1)+i)+y(14+i+5*num)              α(x8, x8((-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 3: A sample program and its GSA form

By transforming a source program into its GSA form, we can treat arrays with different reference regions as different symbolic variables. In the global symbolic forward substitution, information is propagated 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 [30].

In addition to the above 3 functions, another function called $\alpha$(array, subscript, value) [16] is used to replace the array assignment statement. The semantics of the $\alpha$ 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. Hence, the def-use chain is still maintained by the links associated with each unique array name. Figure 3 shows an example of a program with its GSA form. We'll refer to this example throughout the discussion in this section.

#### 3.1.2 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 containing the following fields:

- **name(D)** the variable name
- **subarray(D)** the region of the variable being referenced
- **offset(D)** the offset to mark a descriptor propagated across an epoch boundary

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 offset field is reset (to value "0") initially, and incremented when a descriptor is propagated across an epoch boundary during the flow analysis. This field is used exclusively for stale reference detection.

We define the following three binary operations and one unary operation on data descriptors:

1. **union ("$\cup$, $\oplus$")**
2. **intersection ("$\cap$, $\otimes$")**
3. **difference ("-, $\ominus$")**
4. **aggregate ("$\int_{inner}^{outer}$")**

We define two kinds of operators 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 the may operators and must 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.

The unary aggregate operation "$\int$" 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 "$\int_{inner}^{outer}$" specifies the range of the enclosing loops. When the bounds are omitted, the innermost and outermost loops enclosing the reference are used. Because we consider only do loops, the aggregation is a relatively straightforward interpretation of the index and the bounds of the loops. To aggregate a subarray, we just need to concatenate the loop index and the bounds with the subscripted variable of the index expression. For instance, if I is a loop index or an induction variable with value [1:N:1], then A(I,J) will be aggregated as (A(I,J), [I=1,N]), and A(I,1:I) will be aggregated as (A(I,1:I), [I=1,N]).

These operators are called *subarray operators* and are applied to data descriptors with the same variable name. They 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.

For the program example in Figure 3, let D1 and D2 be the data descriptors of the references to the variable X in the statements S3 and S9, and D3 be the data descriptor of the reference to the variable Y in S9. Table 1 demonstrates the subarray operations and their results[3]. The name field is represented by the original variable name rather than its renamed equivalent in the GSA form. By keeping the original names, we can perform the subarray operations for the same variables during our flow analysis.

| Descriptor D | name(D) | $subarray_{initial}(D)$ | $\int subarray(D)$ |
|---|---|---|---|
| D = D1 | X | i1 | (1:100:1) |
| D = D2 | X | i3+(15+5*num1) | (21:130:1) |
| D = D3 | Y | i3 | (1:65:1) |
| D = D1 ∪ D2 | X | {i1, i3+(15+5*num1)} | (1:130:1) |
| D = D1 ∩ D2 | X | $\phi$ | (21:100:1) |
| D = D1 - D2 | X | {i1} | (1:20:1) |

Table 1: Subarray operations. The $subarray_{initial}$ shows the initial value of the subarray field (i.e. the index expression), while the $\int subarray(D)$ displays the value after aggregation. For simplicity, the example is chosen to have constant subscript bounds for all the subarray fields.

The subarray field represented in 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 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 even though 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 GSA form, we can perform subarray operations involving symbolic loop bounds.

[3]The values of $\int subarray(D)$ for D2 and D3 are based on our implementation. More accurate values are (21:60:1) for D2 and (1:30:1) for D3, which can be computed if we consider the loop bounds of S6 to interpret the predicate of the IF statement in S7.

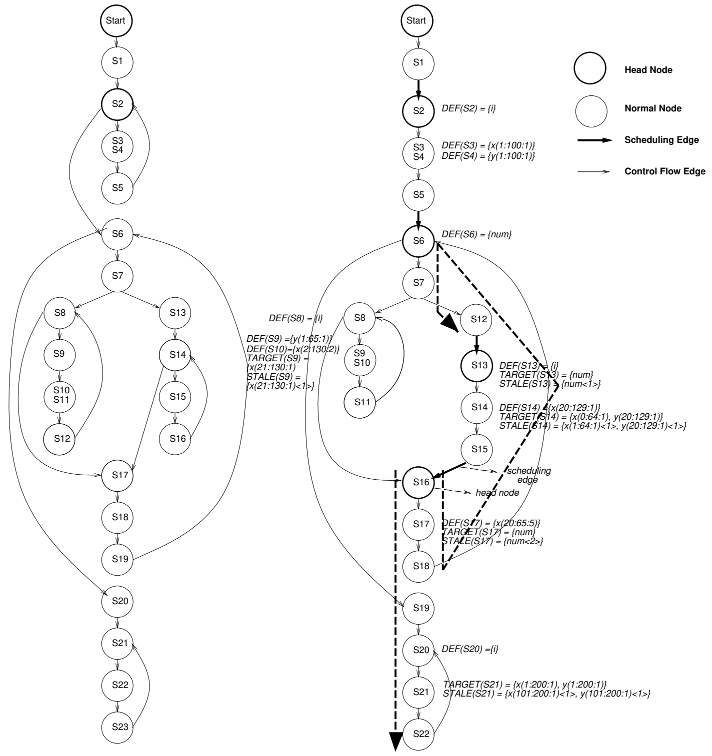



Figure 4: Control flow graph and the corresponding epoch flow graph for the program example. The bold-dotted path shows two instances of the epoch where statement S17 can belong at runtime. The epoch flow graph also shows DEF(S), TARGET(S), and STALE(S) computed for each statement according to the stale reference detection and upwardly-exposed use algorithms. The offset field is show in "<>".

### 3.2 Stale reference detection

Because the epoch boundary information is essential in identifying potentially stale references, we extend the flow graph used in a sequential program to include parallel constructs.

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

- *Head Node* A basic block with an incoming *scheduling* edge. Start node is a special head node that does not have any incoming *scheduling* edge.

- *Tail Node* A basic block with an outgoing *scheduling* edge.

- *Epoch Level* A subset L of an epoch flow graph G' that includes only a single head node H, and all the nodes and edges that have a directed path from H without crossing a scheduling edge. The set L is called an *epoch level from H.* There is a one-to-one relationship between head nodes and epoch levels. Multiple tail nodes can exist in an epoch level L from H. A directed path from H to each tail node T is called an *epoch from H to T*.

Figure 4 shows the control flow graph for the program example in Figure 3 and its corresponding epoch flow graph. Note that, to reflect the control flow of the parallel execution, back edges of parallel loops are removed in the epoch flow graph. A directed path from each head node to tail node shows the epoch created at runtime. Also, note that a node can belong to more than one epoch at runtime. For example, in Figure 4, the statement S17 can belong to the epoch from S16 to S22 (consisting of nodes S16, S17, S18, S19, S20, S21, and S22) and the epoch from S16 to S15 (consisting of nodes S16, S17, S18, S6, S7, S12, S13, S14, S15). This implies that the statement S17 can belong to different epochs at runtime depending on the branches taken.

**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 the variable $v$ in statement S2 to S1, and the path includes at least one scheduling edge.

When a definition reaches a read reference across at least one scheduling edge, there should exist a stale reference sequence (a) to (c), and the read reference to $v$ should be marked as potentially stale. We call it *potentially stale* since it may or may not lead to a stale reference at runtime. Note that because of the dynamic task scheduling at each epoch boundary, a definition in an epoch can reach any subsequent epoch without being *killed*. Therefore, the notion of *kill* used in the traditional reaching definition algorithm is no longer valid. For each statement S, we define the following sets.

- $DEF(S)_{offset}$ is a set of definitions generated by S. Since all the definitions created in S are assigned the same offset, the offset field can be denoted collectively for S as $DEF(S)_{offset}$.

- *IN(S)* is a set of definitions reaching the beginning of S, $IN(\text{S}) = \bigoplus_P OUT(\text{P})$ where P is a predecessor of S.

- *OUT(S)* is a set of definitions reaching the end of S, $OUT(S) = IN(S) \oplus DEF(S)_0$.

- *STALE(S)* is a subset of $USE(S)$ where there exists a definition reaching the beginning of S that crosses one or more epoch boundaries, $STALE(S) = \{$use $u$ of the variable $v$: $u \in USE(S)$ and $\exists$ a definition $d$ of the variable $v$ in IN(S) where offset(d) $> 0$ and (subarray($u$) $\otimes$ subarray($d$)) $\neq \phi$ $\}$.

**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;
     OUT(S) ← DEF(S)_0;
     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 its corresponding predecessor P of S do
               if p is a scheduling edge then
                  increment the offset of 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
               create a new data descriptor new where
                   subarray(new) ← φ
                   offset(new) ← ∞
               for each definition d of variable v with offset(d) > 0 in IN(S) do
                   if (subarray(u) ⊗ subarray(d)) ≠ φ then
                      subarray(new) ← subarray(new) ⊕ (subarray(u) ⊗ subarray(d));
                      if (offset(d) < offset(new)) then
                         offset(new) ← offset(d); /* find the minimum offset */
                      end if
                   end if
               end for
               for each descriptor old of variable v in STALE(S) do
                   subarray(old) ← subarray(old) ⊕ subarray(new);
                   if (offset(new) < offset(old)) then
                      offset(old) ← offset(new);
               end for
               delete the data descriptor new;
           end for
           OUT(S) ← OUT(S) ⊕ IN(S);
       end for
 end while
End
```

Figure 5: 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.

Figure 5 shows the detailed algorithm for stale reference detection. Initially, all the definitions are associated with an offset 0. We propagate these definitions through the epoch flow graph and increment their offsets when they cross scheduling edges. For each target read reference, if there is a reaching definition with an offset greater than 0, then the reference is marked as a Time-Read operation. To refine stale reference marking for an array reference, we take the intersection of subarrays of reaching definitions and the target reference. When there exists a definition whose intersection with the target reference has nonempty subarrays, the target reference is marked as a Time-Read and its offset is determined by taking the minimum of all the offsets of such definitions.

**Single-word cache lines** In Appendix A.1, we present an algorithm for single-word cache lines. Since there is no false sharing (i.e., there are no implicit RAW and WAR dependences among the cache lines), the stale reference detection algorithm further refines reference marking

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

Table 2: Reuses and corresponding locality preserving analysis techniques.

based on a two-phase def-def and def-use chain analysis. The scalar version of this algorithm can also be used for multi-word cache lines if variables are aligned at the cache line boundaries. Algorithms for both the multi-word cache lines and the single-word cache lines have been implemented in Polaris.

## 3.3 Locality preserving analysis

Not all stale reference patterns lead to a stale data access at runtime. Since each potentially stale reference implies a remote memory access instead of a cache hit, we should minimize the number of potentially stale references marked at compile time by utilizing both the temporal and spatial locality in a program as much as possible. Wolf [32] discussed 4 different types of reuses in a loop as shown in Table 2. 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.

### 3.3.1 Upwardly exposed use

An *upwardly-exposed use* of a variable in a block B of statements is a use which is not preceded by any previous definition of the same variable in B. [4]

**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 potentially 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 to the first occurrence of upwardly-exposed uses as an upwardly-exposed use. Subsequent reads will not be marked as upwardly exposed use in our analysis. Upwardly exposed uses of a scalar variable can be precisely detected using

[4] A block B can be either an epoch or a procedure in our analysis.

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. To perform this analysis, we compute the following data sets from DEF and USE sets of each statement.

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

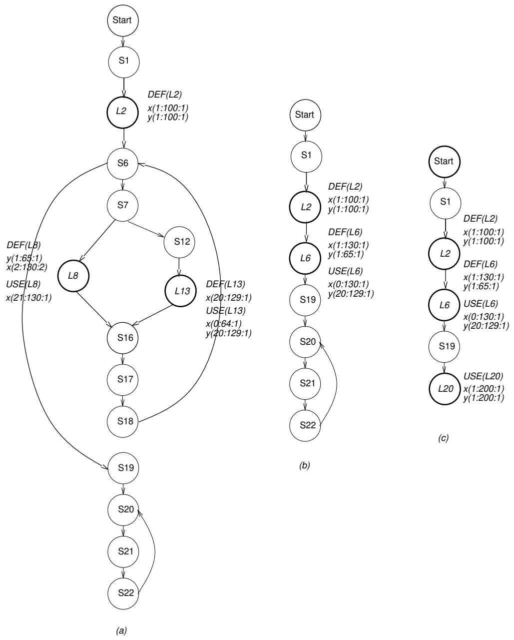



Figure 6: Aggregation of loops during analysis of upwardly-exposed reference. (a) after loop L2, L8 and L13 are summarized, (b) after loop L2 and L6 are summarized, (c) after all the outer loops are summarized. Loop nodes are represented as bold nodes and show the aggregated USE and DEF set for arrays X and Y.

ACCESSED_IN(S) denotes all the scalar variables and subarrays that are referenced unconditionally before reaching the statement S. When more than one edge enters the statement S, we use "$\cap$" as the *confluence operator* to compute ACCESSED_IN(S) since a reference to a variable belongs to ACCESSED_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 is

**Algorithm 3.2: Upwardly-Exposed Use (range)**

**Input**: Epoch flow graph G and DEF(S) and USE(S) for each statement S
**Output**: TARGET(S) for each statement S

```
Procedure initialize
    for each statement S do
        ACCESSED_IN(S) ← φ;
        ACCESSED_OUT(S) ← DEF(S) ∪ USE(S);
        initialize subarray fields of array references in ACCESSED_OUT(S)
           to the corresponding index expression of the references;
        TARGET(S) ← φ;
    end for
End Procedure

Begin
  call initialize;
  while changes to ACCESSED_OUT(S) occur for any statement S do
        for each statement S do
            if ((range == EPOCH) and (there exits an incoming scheduling edge)) then
               call initialize; /* skip the propagation and reinitialize */
            ACCESSED_IN(S) ← ∩_{predecessor P} ACCESSED_OUT(P);
            ACCESSED_OUT(S) ← ACCESSED_OUT(S) ∪ ACCESSED_IN(S);
            for each upwardly-exposed read reference R in S do
                if R is a scalar reference then
                   if !(R ⊆ ACCESSED_IN(S)) then
                      TARGET(S) ← TARGET(S) ∪ R;
                else /* for array references */
                   if ((subarray(R) ⊖ subarray(ACCESSED_IN(S))) != φ) then
                      subarray(R) ← subarray(R) ⊖ subarray(ACCESSED_IN(E))) ;
                      TARGET(S) ← TARGET(S) ⊕ R;
                   endif
                endif
            end for
            if (S == ENDDO statement) then
               for each array reference R in ACCESSED_OUT(S) that are inside the current loop L do
                      R ← ∫_L^L R;
            endif
        end for
  end while
End
```

Figure 7: Marking algorithm for upwardly-exposed uses. The parameter range can be either EPOCH or PROGRAM. EPOCH is used for target reference detection while PROGRAM is used to find read-induced definitions for single-word cache lines as shown in Appendices.

propagated only through forward edges. A bypass edge from the end of a 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 ACCESSED_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(ACCESSED_IN(S)). For this analysis, we need to perform aggregation for array references in the data sets ACCESSED_IN and ACCESSED_OUT during the flow analysis. Initially, all the subarray fields of the array references in ACCESSED_OUT(S) are initialized to the index expressions of the array references. When we finish a loop, we summarize 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. For example, Figure 6 shows the aggregation of the loops during the analysis of the program example shown in Figure 3. Figure 4 shows the target references marked for the program example.

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 $\mu$ and $\gamma$ functions and the first argument of the $\alpha$ function. For simplicity, this is not shown in the algorithm in Figure 7.

```
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 8: Code generation using guarded execution for statement S21 in the program example. Bypass-read operations (b-read) are issued for potentially stale data references.

### 3.3.2 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 subarray access information.

For the program example shown in Figure 3, after stale reference detection and target reference detection, our algorithms find X(101:200:1) and Y(101:200:1) as potentially stale for read references in statement S21. Figure 8 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 potentially stale data reference. As shown in Figure 8(b), half of remote memory accesses can be eliminated by checking the array range information available from 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 8(c). A similar optimization can be done for self-temporal reuses.

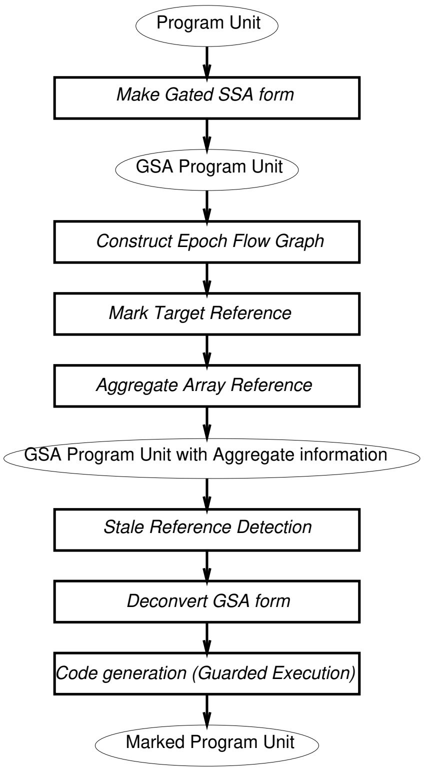



Figure 9: Flowchart for intraprocedural stale reference marking.

### 3.4 Overall intraprocedural algorithm

First, we transform the source program to GSA form. Then, we construct an epoch flow graph, which contains the epoch boundary information as well as the control flow of the program. Given a source program unit and its epoch flow graph G, we mark target references in each epoch for potentially stale references. After marking the target references, the regions of arrays referenced in each epoch are computed. Then, the 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.

## 4 Beyond procedural boundaries

**Complications in the presence of procedure calls** Until now, the algorithm presented assumes program units (i.e. single procedures) and, therefore, single flow graphs. The presence of procedures and procedure calls introduces the following complications to the stale reference detection.

- **side effect** The execution of a procedure can have a side effect on variables at the point from which the procedure is called. These variables include the actual parameters at the call site as well as the global variables visible to both the calling procedure and called procedure.
- **hidden context** Any of the global variables and formal parameters of a procedure could have been previously read or written at the beginning of the procedure.
- **aliases** The third issue is the aliasing caused by the call-by-reference parameter passing mechanism and the EQUIVALENCE statement of FORTRAN.
  1. **static alias** EQUIVALENCE statements cause distinct variables to refer to the same memory location. Since the alias relationship is fixed in any instantiation of the procedure, we call it *static* alias. In our algorithm, all the static aliases are treated as a single variable.
  2. **dynamic alias** The call-by-reference parameter passing mechanism associated with procedure calls in FORTRAN can cause two distinct variables to refer to the same memory location.

**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 an invalidation, we have to consider the definitions of all the global variables and formal parameters before the procedure is called. The dynamic aliases 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.

All of the previous compiler algorithms use cache invalidations at the procedure boundaries. However, both cache invalidation and inlining approaches have their own problems. The obvious drawback of cache invalidations is that it cannot exploit locality across procedure boundaries. The cache invalidation invalidates not only shared data, but also private and read-only shared data unnecessarily. As we demonstrate in section 5, frequent invalidations at procedure call boundaries incur cold-start effects, thus leading to poor cache performance particularly for programs that contain many small procedure calls in their critical path of execution. Inlining can achieve the most precise analysis, but it is often prohibitive due to the excessive growth in code size and the increase in compile time, since the memory requirement and complexity of the reference marking algorithm is often nonlinear in procedure size. In the following, we develop intraprocedural and interprocedural compiler algorithms, both of which can perform stale reference detection without relying on either cache invalidation or inlining.

## 4.1 A modified intraprocedural algorithm to avoid cache invalidations

Another approach that can avoid cache invalidations without relying on interprocedural analysis is employing a more conservative stale reference detection algorithm. This is achieved by considering both side effects and hidden contexts during the stale reference detection. This modified intraprocedural algorithm is presented in Figure 17 of Appendix A.2, and works as follows.

Due to the unknown context information, we assume that all the formal parameters and global variables have been previously modified at the beginning of a procedure. This is accomplished by keeping track of the minimum offset from the beginning of a procedure. For a target reference with no reaching definition inside a procedure, we issue a Time-Read with the minimum offset, implying that the referenced data item can be potentially modified before entering the procedure. This is an improvement over previous algorithms [8, 12] that use cache invalidation at the beginning of a procedure since only global and formal variables are affected by the unknown context information. We propagate definitions through the flow graph and increment their offsets when they cross scheduling edges. Note that for procedure CALL statements, we purposely insert all the actual parameters and global variables in its OUT set, implying that those variables can be modified at the call site. Without any information for the procedure called, we have to assume that an entire array can be modified by the procedure. Note that this lack of information also affects the analysis of the subroutine local variables since the offset can be unnecessarily small if the call site has at least one epoch boundary. This small offset can incur unnecessary cache misses at runtime because the epoch counter will be incremented at runtime by the number of epoch boundaries inside the call site. This will cause the following Time-Reads to miss due to the conservative offset values. As in the algorithm in Figure 5, whenever there is a reaching definition in IN(S) with an offset greater than 0, the target reference is marked as a Time-Read operation.

The advantage of this intraprocedural algorithm is that it can avoid unnecessary side effects of cache invalidations (i.e. the algorithm does not invalidate private and read-only shared data). However, this can only achieve limited cache utilization since it cannot exploit locality across procedure boundaries for shared writable variables. To further exploit locality, we need to look at an entire program rather than one program unit at a time.

## 4.2 Two-pass full interprocedural algorithm

To perform full interprocedural analysis, we use the *procedure call graph*, which is the basic data structure for the interprocedural analysis.

**Definition: procedure call graph** Let a directed multi-graph G = (V, E) represent a call graph where V is a set of procedures, and E is a set of directed edges. An edge from node p to node q exists if procedure p can invoke procedure q.

We extend the procedure call graph to contain the following summary information for each procedure.

- Procedure name, code size (number of lines)

- Number/types of formal parameters and global variables
- Number/types of actual parameters for each call site (*call site information*)
- OUT sets of callees at each call site (*summary side effect information*)
- IN sets of callers at the beginning of this procedure (*summary context information*)
- Number of minimum epoch boundaries in a procedure

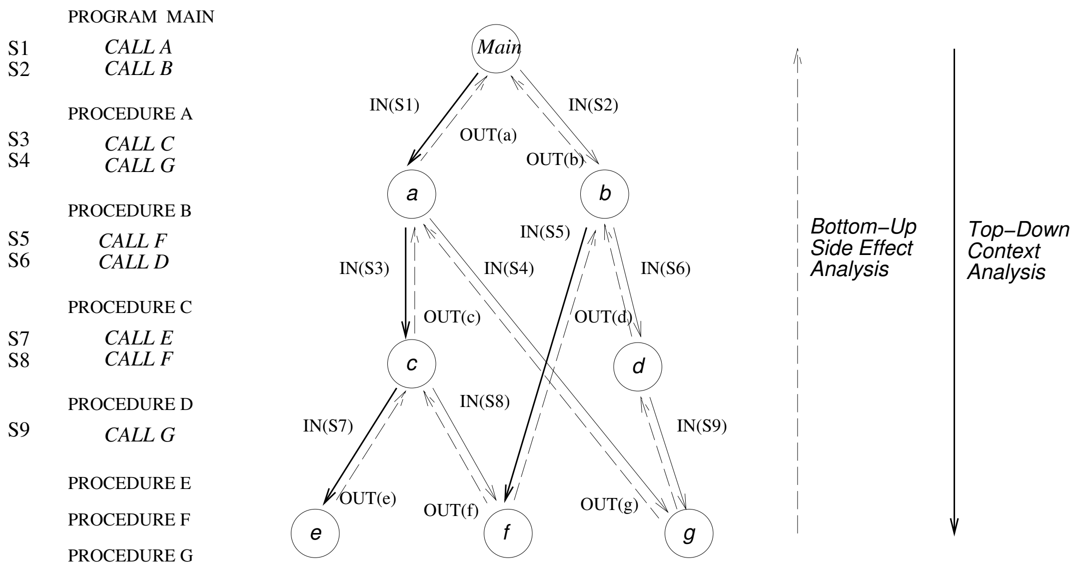



Figure 10: An example of a procedure call graph and the overall structure of the two-pass interprocedural analysis.

### 4.3 Side effect analysis

Figure 10 shows the overall structure of interprocedural analysis. It consists of 2 passes on the procedure call graph: bottom-up *side effect analysis* and top-down *context analysis*. The detailed algorithm is presented in Figure 18 of Appendix A.2.

The side effect analysis combines intraprocedural analysis with a bottom-up scan of the procedure call graph to eliminate the side effects caused by each call site. The side effects of each call site are summarized by the OUT set of the procedure called. We first start at the bottom of the procedure call graph. We apply the intraprocedural algorithm described in Figure 17 of Appendix A.2 for all the leaf procedures. This can be performed without considering the side effects since those procedures do not include procedure calls. After performing the intraprocedural analysis, we can summarize the side effect information and propagate to the procedures in the next level that have call sites to the current procedure. Note that this requires the translation of the summary information from callee's context to caller's context (see section 4.5).

The summary side effect information should contain the following information for each actual parameter and global variable:

- whether the variable is modified
- the number of epoch boundaries from the last write (offset)
- the regions of an array that have been modified (subarray)

Note that all the above information can be represented by the OUT set of the procedure called, which is already computed by intraprocedural analysis. Since a procedure can have multiple return points, the OUT sets at all the return points should be merged. This can be accomplished by taking the conservative union of all the OUT sets and by taking the minimum offset among the OUT sets for each global variable. Since we need the side effect information only for actual parameters and global variables, we eliminate the information for subroutine local variables from the summary information. By using the summary information, we add the translated OUT set from the callee to the OUT set of the CALL statement in a caller.

In addition to the above information, for each call site we should compute the minimum number of epoch boundaries crossed, and add the number to each definition in the OUT set of the corresponding CALL statement. Without this, the offsets for Time-Reads after the procedure call would not reflect the number of epoch boundaries crossed in the call site, which may generate offsets too small to capture the locality across the procedure boundaries.

### 4.4 Context analysis

The side effect analysis summarizes the data-flow information from the descendants of the procedure call graph. However, reference marking using only such an analysis is still conservative because we assumed that all the global variables and formal parameters of each procedure have been modified before entering the procedure, due to the unknown context information for the activation records invoking the current procedure. To eliminate this conservative assumption, we need to perform the second pass, which is the top-down context analysis.

We start from a main program unit. Since the main program unit does not have any context at the beginning, the previous bottom-up analysis already generates a precise result. But, we need to propagate the context information of the main program unit to all its call sites. Generally, for each call site, we need to propagate the context information of the caller to the callee. The context information in our analysis can be represented by the IN set of CALL statement at each call site. This context propagation allows the IN set at the beginning of a procedure to be replaced by the IN sets of its corresponding callers' CALL statements. Since the context information is necessary only for formal parameters and global variables, we propagate the summary context information only for those variables. As opposed to the side effect analysis, we need to translate the IN set information from the caller's context to the callee's context. Note that there can be multiple callers to a procedure. So, we need to merge the context information from multiple call sites. This is achieved by taking the union of all the IN sets from the call sites and by taking the minimum offset among multiple callers for each actual parameter and global variable. This is necessary unless we clone the procedure for each call site (duplicate the code for each case), which in the worst case produces the same effect as inlining.

Using the summary context information, we can refine the conservative reference marking obtained from the previous bottom-up analysis. With the context information, we can compute a more precise (larger) offset for each Time-Read, or we can eliminate the Time-Read completely if there has been no previous write to the variable referenced. To do so, we mark all the Time-Reads issued as a result of hidden context (Time-Reads that do not have a reaching definition inside the procedure and all its call sites) during side effect analysis. Note that for context analysis, we need to update the reference marking results only for these references. The other Time-Reads are already precise with the side effect analysis alone.

Using the bottom-up and top-down analyses, we can avoid redundant computation by performing a minimal number of computations (twice) per program unit. In addition, our top-down analysis updates the reference marking results only for necessary cases, allowing incremental updates. Note that during the top-down pass, we don't have to propagate the data-flow information for each procedure again since the summary context information is enough to refine reference marking. We just need to add the offset of each variable from the summary context information to the offset of Time-Reads for the corresponding variables. For the Time-Reads that do not have definitions in the summary context, we could eliminate them completely since there have been no previous writes to the variables referenced by them.

In addition, this two-pass analysis allows separate compilation. We don't need to load an entire program in memory for the interprocedural analysis. We need to load only a single procedure at a time, as well as the procedure call graph with the summary information. This two-pass interprocedural algorithm allows incremental flow analysis without losing any preciseness. We also can limit the scope of the analysis on a level-by-level basis rather than the entire call graph.

### 4.5 Issues for interprocedural array data-flow analysis

**Naming translation** To propagate data-flow information interprocedurally, we need to consider the naming translation between callers and callees. In interprocedural stale reference detection, both the summary side effect information and the summary context information should be translated into the context of the procedure analyzed. There are two cases when a variable can be renamed across procedure boundaries.

- **parameter translation** All the formal parameters of the side effect information should be translated into the corresponding actual parameters during bottom-up side effect analysis, while all the actual parameters should be translated to their corresponding formal parameters during top-down context analysis. The translation can be complicated for array variables due to possible reshaping. For example, a variable with one dimension can be mapped to a two-dimensional variable in the procedure called. Maintaining subarray information in such a case is difficult since all the subarray information will also need to be reshaped.

- **COMMON block translation** A variable in a COMMON block can have different names across procedures sharing the same COMMON block. In addition to possible array reshaping, the translation should consider that a single array variable in a COMMON block declaration can be mapped to several variables (either scalar or array) in the same COMMON block declarations of other procedures.

A simple renaming can be performed both between actual and formal parameters and between COMMON block variables, by looking up the corresponding location in the parameter list or in the COMMON block declaration. However, this renaming can be complicated for array variables due to the potential reshaping and the difference in COMMON block declarations. To take advantage of full array data-flow information, all the subarray information also should be reshaped across procedure boundaries. However, we found that both array reshaping and mismatched COMMON block declarations rarely occur (from 0 to 2 cases per benchmark) for the benchmarks tested. Thus, we use a conservative scalar analysis approach when array reshaping is found. For example, if a procedure A with a COMMON block declaration of a single array variable calls a procedure B that maps the same COMMON block to three different variables, then we assume that the COMMON variable in the caller will be modified at the call site if the OUT set of the procedure B includes any of the three COMMON variables. In addition, we treat all the array variables containing dimensional reshaping as scalar variables.

**GSA form and array data-flow analysis** Since we represent subarray information in GSA form, we need to construct the GSA form carefully. Usually, GSA construction is performed on a per-procedure basis. In this case, to consider all the side effects at the call sites, all the global variables (actual parameters and COMMON variables) are assumed to be modified at each call site. However, this may lead to imprecise analysis since those global variables might be read-only inside the procedure called. For a more precise array analysis, we construct GSA interprocedurally using side effect (MAYMOD) information with additional flow analysis.

# 5 Experimentation

We have implemented all the compiler algorithms in the Polaris parallelizing compiler. In this section, we demonstrate how different compiler algorithms affect the performance. Perfect Club benchmark suites [4] are chosen as our target applications. They are first parallelized by the Polaris compiler. In the parallelized codes, the parallelism is expressed in terms of DOALL loops. Then, we process the parallelized source codes using both scalar and array flow analysis versions of the algorithms given in sections 3 and 4.2. Execution-driven simulations [28] are used to verify the compiler algorithm and to evaluate the performance of compiler-directed coherence schemes. All the simulations assume a 16-processor, distributed shared-memory architecture with each processor containing an on-chip 64-KB direct-mapped cache with 4-word cache lines. The detailed description of our experimentation methodology and simulations are described in the companion paper [14].

**Compiler algorithms** We use three different compiler algorithms to generate memory operations for the software cache-bypass scheme (*SC*) and the two-phase invalidation scheme (*TPI*).

1. *Invalidation-based intraprocedural algorithm* (**ALG1**) This algorithm performs stale reference detection on a per-procedure basis [12]. To avoid the complications caused by unknown side effects, cache invalidation operations are inserted after each call site and at the beginning of a procedure.

2. *A simple interprocedural algorithm with no cache invalidation* (**ALG2**) Instead of the modified intraprocedural algorithm in Figure 17 of Appendix A.2, we use a more sophisticated algorithm. Using the MAY-MOD information used in GSA construction, we assume that only the variables in MAY-MOD, rather than all the actual parameters and global COMMON variables, are modified at each call site. For hidden context, we assume that all formals and COMMON variables are modified at the beginning of a procedure, as in the intraprocedural algorithm. The algorithm no longer uses cache invalidations.

3. *A full interprocedural program algorithm* (**ALG3**) This algorithm performs the full interprocedural flow analysis by propagating data-flow information across procedures (see section 4.2.).

**Benchmarks and static reference statistics** Table 3 shows the number of potentially stale data references (Time-Reads) marked at compile time using the 3 compiler algorithms. For comparison, we have implemented both array and scalar data-flow analysis versions of each algorithm and show their corresponding results. For both OCEAN and SPEC77, the results of scalar implementations are shown only for algorithms ALG2 and ALG3. The result of the array data-flow algorithms are not available because of its extensive memory usage.

In our compiler implementation, all the variables are treated as shared variables. Therefore, the number of the potentially stale data references are over-estimated in Table 3. However, during simulation, we issue normal memory read operations for all private read references.

After considering both side effects and hidden contexts, the simple interprocedural algorithm ALG2 increases the number of potentially stale data references substantially. On average, an additional 8.71% (array) and 20.15% (scalar) of data references are marked as potentially stale compared to ALG1. The full interprocedural algorithm ALG3 eliminates on average 33.8% (array) and 45.1% (scalar) of the potentially stale data references marked by ALG2. For ALG3, the figure also illustrates the number of Time-Reads removed (the difference in the number of Time-Reads between the end of top-down analysis and the end of bottom-up analysis) and the number of offsets incremented by the context analysis. Note that the impact of the more precise array data-flow analysis is greater in ALG2 and ALG3 than in ALG1 since the interprocedural analysis exposes more references to the array analysis for optimizations.

**Dynamic reference statistics** Table 4 shows the dynamic reference counts of Time-Reads generated during our simulations. Note that the percentage of Time-Reads vary significantly depending on the application used, ranging from 1.96% (QCD) to 39.0% (SPEC77). With ALG1, an average of 3.34% of the memory references are marked as Time-Reads. Note that

| Program | Size (lines) | ALG1 | ALG2 | ALG3 | | |
|---|---|---|---|---|---|---|
| | | | | end of bottom-up | end of top-down | offset incremented |
| FLO52 | 3182 | 5.08% (6.31%) | 16.3% (19.9%) | 11.4% (20.3%) | 8.78% (14.4%) | 1.1% (3.3) |
| MDG | 1523 | 6.76% (10.0%) | 16.2% (22.3%) | 15.0% (23.8%) | 11.2% (16.4%) | 1.2% (2.2) |
| QCD | 3505 | 7.44% (13.5%) | 16.6% (28.8%) | 12.4% (29.3%) | 10.1% (23.2) | 0.4% (1.6%) |
| SPEC77 | 5346 | 4.96% (9.02%) | 9.64% (16.5%) | ** (17.1%) | ** (13.9%) | ** (2.0%) |
| OCEAN | 3101 | 7.71% (8.91%) | 16.6% (18.6%) | ** (18.7%) | ** (13.7%) | ** (3.8%) |
| Avg. | 3122 | 6.39% (9.55%) | 15.1% (29.7%) | 12.7% (24.5%) | 10.0% (18.0%) | 0.9% (2.4%) |

Table 3: The static reference counts of Time-Reads generated by different compiler algorithms. The data in parentheses represent the results of a scalar version of each algorithm. For ALG3, the average figure shows the averaged statistics only from 3 benchmarks: FLO52, MDG, and QCD.

| Program | Size (lines) | % Read | Shared (total refs) | % Time-read | | |
|---|---|---|---|---|---|---|
| | | | | ALG1 | ALG2 | ALG3 |
| SPEC77 | 5346 | 79.3% | 55.3% (5.0*$10^6$) | 3.85% (16.0%) | 23.2% (46.2%) | ** (39.0%) |
| OCEAN | 3101 | 72.6% | 34.9% (3.2*$10^6$) | 0.1% (0.1%) | 4.17% (3.9%) | ** (2.1%) |
| FLO52 | 3182 | 72.9% | 39.4% (2.8*$10^6$) | 10.9% (13.1%) | 16.1% (20.3%) | 12.2% (17.3%) |
| MDG | 1523 | 75.0% | 36.5% (1.8*$10^6$) | 1.71% (1.73%) | 19.5% (**) | 5.94% (5.7%) |
| QCD | 3505 | 78.4% | 18.3% (1.2*$10^6$) | 0.14% (0.57%) | 10.2% (11.0%) | 1.96% (3.0%) |
| Average | 3331 | 75.6% | 36.9% | 3.34% (6.3%) | 14.6% (20.4%) | 6.7% (13.42%) |

Table 4: Dynamic memory reference statistics for the Perfect Club benchmarks. The data in parentheses show the results of scalar algorithms.

this number is misleading because it represents the reference count on a per-procedure basis, since ALG1 uses invalidations at procedural boundaries. By considering the interprocedural side effects and contexts, ALG2 increases the percentage of Time-Reads to 14.6%. With full interprocedural analysis, ALG3 decreases it to 6.7%. The numbers shown in the parentheses represent the percentage of Time-Reads generated by the corresponding scalar implementation. On average, array data-flow analysis could eliminate 51.7%, 23.5%, and 50.0% of Time-Reads marked by the scalar implementation for ALG1, ALG2, and ALG3 respectively.

### 5.1 Impact of compiler algorithms

Figure 11 shows how the different compiler algorithms affect the miss rates of both the software cache-bypass scheme (*SC*) and the hardware scheme (*TPI*) against the underlying BASE architecture, which uses remote memory accesses exclusively for all shared memory references.[5]

As can be seen in the figure, compared to ALG1, ALG2 can eliminate a significant number of

[5]The results of array data-flow analysis are shown for benchmarks FLO52, MDG, and QCD, while the results of scalar versions of algorithms are shown for SPEC77 and OCEAN. This is because SPEC77 and OCEAN require excessive memory using the interprocedural array data-flow algorithm.

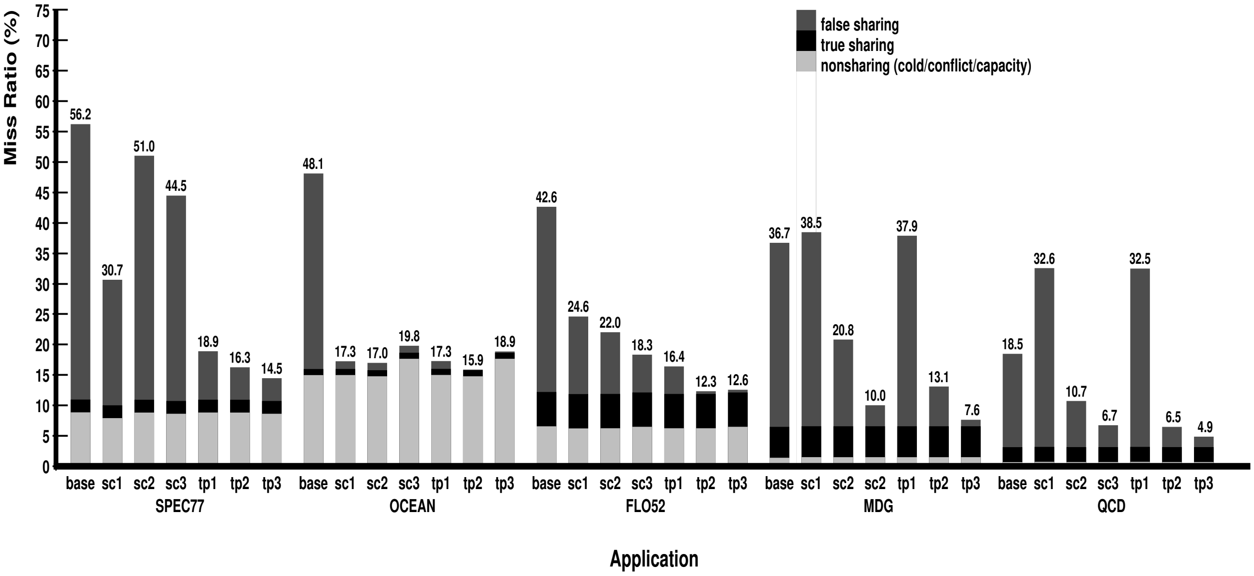



Figure 11: Miss rates using different compiler algorithms for the SC and the TPI scheme compared to an underlying BASE architecture. SC1, SC2, and SC3 represent the software cache-bypass scheme using algorithms ALG1, ALG2 and ALG3 respectively, while TP1, TP2 and TP3 show the two-phase invalidation scheme using ALG1, ALG2 and ALG3 respectively.

cache misses for both *TPI* and *SC* except SPEC77. This is primarily a result of avoiding cache invalidations at procedure boundaries. The most significant improvement in cache utilization is in benchmarks MDG and QCD. In these benchmarks there are several small procedures in the critical path of execution. The invalidations used by ALG1 not only invalidate shared data but also sweep out private data from the cache unnecessarily. For MDG, ALG2 could eliminate 17.7% and 24.8% of cache misses in *SC* and *TPI* schemes respectively. With full interprocedural analysis (ALG3), an additional 10.8% and 5.5% of cache misses are eliminated as compared to ALG2. Similar to MDG, ALG2 eliminates 21.9% and 24.0% of unnecessary cache misses of the *SC* and *TPI* schemes in QCD. And, using the full interprocedural algorithm for *SC* scheme eliminates up to 4.0% of additional cache misses in QCD.

A similar but rather modest trend is observed in other benchmarks. In FLO52, ALG2 achieves a modest improvement over ALG1 for both *SC* and *TPI* schemes, eliminating cache misses by 2.6% and 4.1% respectively. Also, using the full interprocedural algorithm for *SC* scheme eliminates 1.9% of additional cache misses. In OCEAN, most of the misses are dominated by non-sharing misses, and both compiler-directed coherence schemes achieve comparable miss rates regardless of the compiler algorithms used.

In SPEC77, an interesting situation is observed. The simple interprocedural analysis (ALG2) substantially increases the cache misses by 20.3% for *SC* scheme as compared to the invalidation-based algorithm (ALG1). In this benchmark, the negative effects of the conservative stale reference marking used in ALG2 are more significant than the gain from avoiding cache invalidations. Note that this is true only for the software cache-bypass scheme. In SPEC77, the procedures

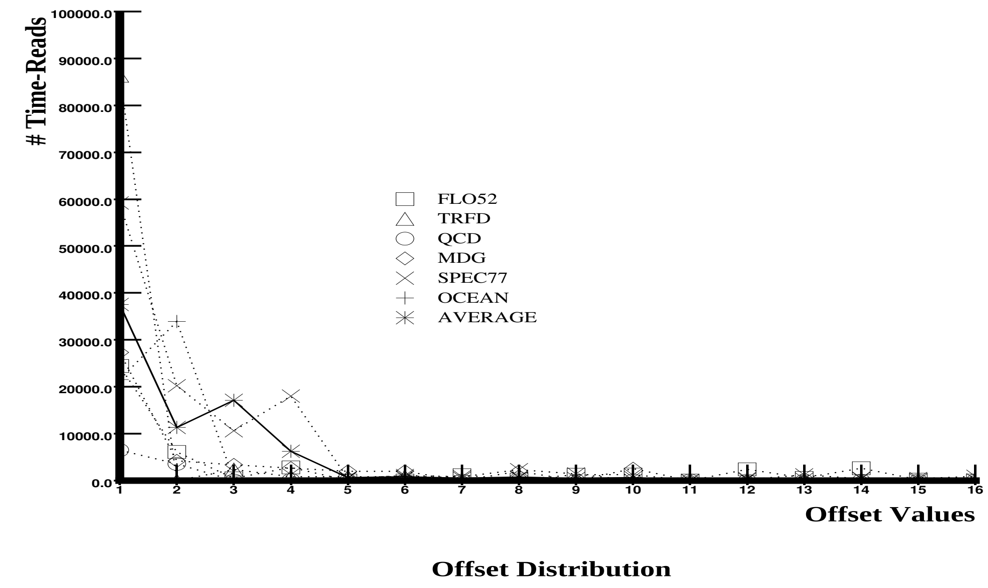


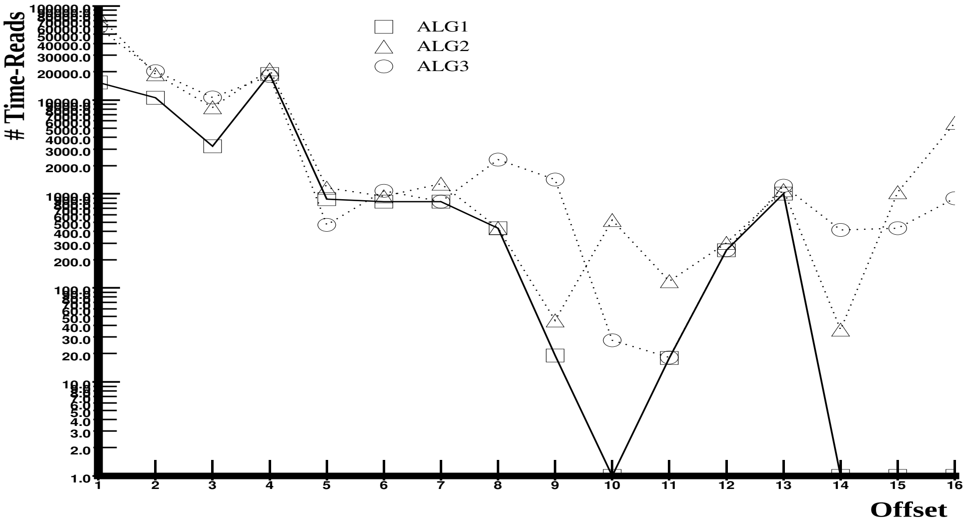

Figure 12: Distribution of offsets for Time-Read. The bold line shows the averaged offset distribution of all benchmarks.

are reasonably large, making the impact of the cold start effects of the cache invalidations negligible. However, the conservative marking strategy employed by ALG2 increases the number of potentially stale references substantially (23.2% (ALG2) compared to 3.85% (ALG1) for array analysis, and 46.2% (ALG2) compared to 16.0% (ALG1) for scalar analysis; refer to Table 4). This leads to a significant number of unnecessary cache misses for the software cache-bypass scheme. For the *TPI* scheme, the conservative marking can be overcome by keeping track of the runtime cache states. In other words, more Time-Reads marked by ALG2 turn out to be cache hits since *TPI* can determine the staleness of cached data at runtime more precisely using the timetag information.

## 5.2 Impact of compiler algorithms on offset distribution

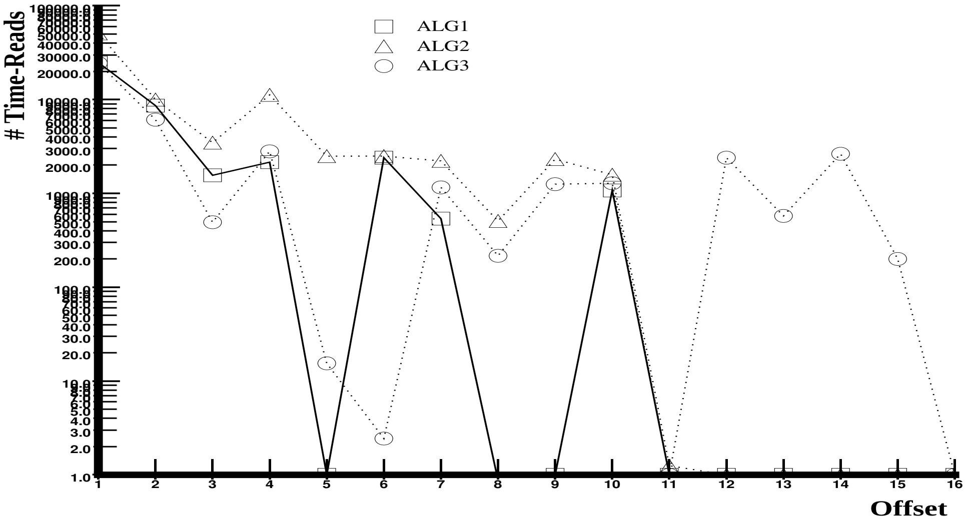
Figure 12 shows the distribution of offsets of Time-Reads for the five benchmarks with the full interprocedural algorithm (ALG3). As shown in the figure, most offsets are small, implying that the distance between the inter-epoch reuses in the source program are small. In fact, more than 90% of offsets are within 8 epoch distances. The largest offset value is 37, and this occurred in SPEC77. This implies that a timetag size as small as 3 or 4 bits should be sufficient to capture most of the inter-epoch locality.

Figures 13 and 14 show the distribution of offsets for Time-Reads in benchmarks SPEC77 and FLO52 using three different compiler algorithms. To demonstrate the distribution of large offsets more clearly, the y axis is shown in log scale. By using interprocedural algorithms (ALG2 and ALG3), larger offsets are generated for the Time-Read operations for both benchmarks. The value of the offset determines how far the two-phase invalidation scheme can exploit locality across epoch boundaries. Compared with ALG2, ALG3 generates fewer Time-Reads with small




**Offset Distribution for SPEC77**

Figure 13: Distribution of offsets for Time-Read in SPEC77 using three compiler algorithms. The y axis is given in log scale to show the distribution of large offsets more clearly. The bold line shows the averaged offset distribution of all benchmarks.




**Offset Distribution for FLO52**

Figure 14: Distribution of offsets for Time-Read in FLO52 using three compiler algorithms.

offsets and more Time-Reads with large offsets. Interestingly, ALG2 sometimes generates more Time-Reads with large offsets, but this is due to conservative marking strategy. These additional Time-Reads are unnecessary and can be eliminated by more precise interprocedural analysis in ALG3.

# 6 Discussion

## 6.1 Previous work

There have been several studies on compiler algorithms for stale reference detection [8, 15]. Among them, only Cheong and Veidenbaum included a compiler implementation study using Parafrase 1 [8] . They pioneered a combination of scalar data-flow analysis and graph algorithms to find potentially stale references. In [15], we proposed a simpler algorithm that eliminates the graph construction phase of [8], but it may overestimate potentially 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 [19]. Feautrier [18] gave an algorithm to calculate them exactly. Pugh [29] developed some exact techniques that are substantially faster than Feautrier's. Our implementation is based on regular section analysis[5], which is less accurate but allows large programs to be analyzed efficiently.

## 6.2 Code generation for other compiler-directed schemes

In Lifespan strategy [7], memory-reads should be issued for all the potentially stale references. In addition, we need to compute the N-bit vector for all the read and write references. The bit vector is used to invalidate cache copies created by each read or write reference before a new version is created. Using another pass of def-def or use-def chain analysis, the N-bit vector can be computed by using the offset field similar to the two-phase invalidation scheme.

It is also straightforward to apply our algorithm for the fast selective invalidation scheme [9]. All the potentially stale references marked are issued as memory-reads, while cache invalidate operations should be inserted at every epoch boundary.

Overall, the stale reference detection algorithm can be used for any hardware or software coherence techniques. Hardware directory protocols can take advantage of this compile time analysis to eliminate unnecessary coherence transactions for safe memory references. The same applies to other compiler-directed coherence techniques.

# 7 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 remains a challenge. Hardware directories can be used to maintain coherence but require significant storage overhead and complicated hardware design for their directory and cache controllers. In this paper, we develop and implement compiler algorithms that maintain cache coherence without requiring such expensive hardware. The algorithms eliminate unnecessary remote memory accesses by detecting potentially stale references at compile time. These algorithms 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 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 form for demand-driven symbolic analysis.

Our algorithm is based on a full interprocedural analysis. It performs intraprocedural analysis according to the bottom-up and the top-down scan of the procedure call graph. The bottom-up side effect analysis replaces each call site with summary side effect information from its descendants, while the top-down context analysis propagates the context of predecessors to each procedure. This eliminates cache invalidations used by previous algorithms [8], and allows the locality of programs to be preserved across procedure boundaries.

We have implemented these algorithms in the Polaris parallelizing compiler [27], and measured the performance driven by the new compiler algorithms by running execution-driven simulations of five Perfect benchmarks. Our results show that, for most benchmarks, the compiler algorithms can improve cache utilization significantly by caching remote shared data. However, cache invalidations at procedure boundaries can sometimes increase cache misses substantially because they invalidate both private and shared read-only data unnecessarily. This suggests that it is important to exploit data locality across procedural boundaries. By avoiding cache invalidations, a simple modified intraprocedural algorithm eliminates up to 26.0% of the cache misses for a compiler-directed scheme, compared to an existing invalidation-based algorithm [12]. With the full interprocedural analysis, up to 10.8% of additional cache misses can be removed.

**Acknowledgments** 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 Kuck and Associates, Inc. for his numerous help in the development of execution-driven simulations. We also wish to thank Polaris group members Peng Tu, Jay Hoeflinger, and Prof. David Padua for their guidance on the implementation of array data-flow analysis in the Polaris compiler. Finally, we thank Lawrence Rauchwerger and IBM for providing RS6000 clusters for simulations.

# References

[1] S. V. Adve, V. S. 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 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 SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271, June 1990.

[4] M. Berry and others. 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] Y.-C. Chen and A. Veidenbaum. Comparison and Analysis of Software and Directory Coherence Schemes. *Proceedings Supercomputing'91*, pages 818–829, November 1991.

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

[8] 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.

[9] H. Cheong and A. Veidenbaum. A Cache Coherence Scheme with Fast Selective Invalidation. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988.

[10] H. Cheong and A. Veidenbaum. A Version Control Approach To Cache Coherence. *Proceedings of the 1989 ACM/SIGARCH International Conference on Supercomputing*, June 1989.

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

[12] Lynn Choi and Pen-Chung Yew. Eliminating Stale Data References through Array Data-Flow Analysis. *To appear in IEEE International Parallel Processing Symposium*, April. 1996.

[13] Lynn Choi and Pen-Chung Yew. Interprocedural Array Data-Flow Analysis for Cache Coherence. *Proceedings of the Eighth International Workshop on Languages and Compilers for Parallel Computing '95, also avaiable as CSRD Technical Report No. 1427*, Aug. 1995.

[14] Lynn Choi and Pen-Chung Yew. Hardware and Compiler-Directed Cache Coherence in Large-Scale Multiprocessors: Design Considerations and Performance Study. *submitted to IEEE Transactions on Parallel and Distributed Systems*, Feb. 1996.

[15] Lynn Choi and Pen-Chung Yew. A Compiler-Directed Cache Coherence Scheme with Improved Intertask Locality. *Proceedings of the ACM/IEEE Supercomputing'94*, pages 773–782, Nov. 1994.

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

[17] E. Darnell and K. Kennedy. Cache Coherence Using Local Knowledge. *Proceedings of the Supercomputing '93*, pages 720–729, November 1993.

[18] Paul Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20(1), Feb. 1991.

[19] Elana D. Granston and Alexander V. Veidenbaum. Detecting Redundant Accesses to Array Data. *Proceedings of the Supercomputing '91*, Nov. 1991.

[20] Cray Research Inc. Cray T3D System Architecture Overview. Mar. 1993.

[21] J. Kuskin, D. Ofelt, M. Heinrich, and J. Heinlein et al. The Stanford FLASH Multiprocessor. *Proceedings of The 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 18-21, 1994.

[22] 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.

[23] A. Louri and H. Sung. A Compiler Directed Cache Coherence Scheme with Fast and Parallel Explicit Invalidation. *Proceedings of the 1992 International Conference on Parallel Processing*, I, Architecture:2–9, August 1992.

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

[25] 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.

[26] F. Mounes-Toussi and D. Lilja. The Potential of Compile-Time Analysis to Adapt the Cache Coherence Enforcement Strategy to the Data Sharing Characteristics. *IEEE Transactions on Parallel and Distributed Systems*, 1995.

[27] D. A. Padua, R. Eigenmann, J. Hoeflinger, P. Peterson, P. Tu, S. Weatherford, and K. Faign. Polaris: A New-Generation Parallelizing Compiler for MPPs. In *CSRD Rept. No. 1306*. Univ. of Illinois at Urbana-Champaign., June, 1993.

[28] D. K. Poulsen and P.-C. Yew. Execution-Driven Tools for Parallel Simulation of Parallel Architectures and Applications. *Proceedings of the Supercomputing 93*, pages 860–869, Nov. 1993.

[29] William Pugh and David Wonnacott. An Evaluation of Exact Methods for Analysis of Value-based Array Data Dependences. *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, August 1993.

[30] Peng Tu. Automatic Array Privatization and Demand-Driven Symbolic Analysis. Technical report, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, 1995. Ph.D. Thesis.

[31] A. V. Veidenbaum. A Compiler-Assisted Cache Coherence Solution for Multiprocessors. *Proceedings of the 1986 International Conference on Parallel Processing*, pages 1029–1035, August 1986.

[32] Michael E. Wolf. Improving Locality and Parallelism in Nested Loops. Technical report, Stanford University, Dept. of Computer Science, August 1992. Ph.D. Thesis.


# A Appendix

## A.1 Stale reference detection for single-word cache lines

For single-word cache lines, there are no implicit RAW and WAR dependences among the cache lines in different processors. Therefore, as we discussed in section 2.1, it is sufficient to find 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 ends with (5) a read. The sequence of memory reference events and its corresponding def-use chain are shown in Figure 15. The first read in the sequence is called a *read-induced definition* in our analysis, and is treated as a special case of write because it will generate an initial version of the data in a cache through its cache miss. The definition created by a write is called a *write-induced definition* to distinguish it from a read-induced definition. Unless specified otherwise, in the following discussion a definition refers to both the read-induced definition and the write-induced definition.

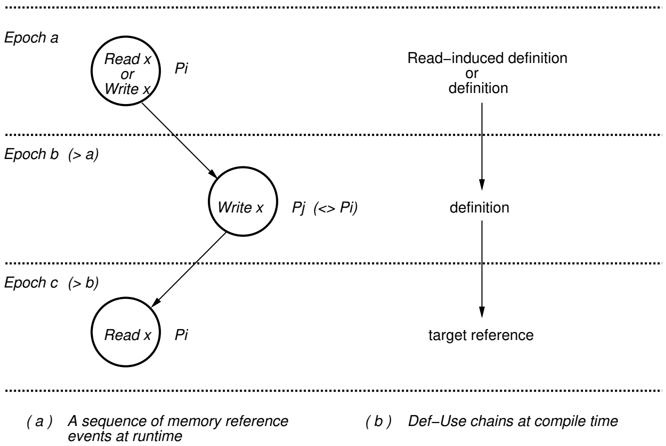



Figure 15: Stale access conditions.

**Definition : read-induced definition** The first occurrence of the *upwardly-exposed uses* of a variable in a procedure is called a *read-induced definition.* We use the same algorithm as one used in the target reference detection in Figure 7. The algorithm is performed on the entire program unit (using PROGRAM as its range parameter) instead of just within each epoch. Note that for a partially upwardly-exposed reference R, we mark only the region of the array that is actually upwardly exposed as a read-induced definition instead of the entire region referenced by R.

**Definition : redefining definition** A definition $d_1$ of a variable $v$ in statement S1 is called a *redefining definition* if it is a write-induced definition, and if there exists a directed path from another definition (either a write-induced definition or a read-induced definition) of $v$ in S2 to S1 in the epoch flow graph that includes at least one scheduling edge.

**Definition: potentially stale** A target 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 redefining definition $d$ of variable $v$ in statement S2 to S1 including at least one scheduling edge.

For each statement S, we define the following sets.

- $DEF(S)_{offset}$ is a set of definitions in S.
- $READ\_DEF(S)_{offset}$ is a set of read-induced definitions in S.
- *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, $OUT(S) = IN(S) \oplus DEF(S)_0 \oplus READ\_DEF(S)_0$.
- *RD(S)* is a set of redefining definitions in S where there exists a definition reaching the beginning of S across epoch boundaries, RD(S) = {definition $d$ of the variable $v$: $d \in$

DEF(S) and $\exists$ a definition *d1* of variable $v$ in IN(S) where offset($d$) $> 0$ and (subarray($d$) $\otimes$ subarray($d1$)) $\neq \phi$ }.

- *$IN'(S)$* is a set of redefining 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 redefining definitions reaching the end of S, $OUT'$(S) = $IN'(S) \oplus RD(S)$.
- *STALE(S)* is a set of uses in S where there exists a redefining 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 redefining definition $d$ of the variable $v$ in $IN'$(S) where offset($d$) $> 0$ and (subarray($u$) $\otimes$ subarray($d$)) $\neq \phi$ }.

The redefining definition represents the sequence of events from (a) to (c). When a redefining definition reaches a read reference across at least one scheduling edge, there should exist a sequence of events from (a) to (e). To refine stale reference marking for an array reference, we take the intersection of the subarrays of a reaching redefining definition and the read reference. When there exists a redefining definition whose intersection with the read reference has nonempty subarrays, the read reference is marked as potentially stale. The detailed algorithm is shown in Figure 16 of Appendix A.2.

The previous scalar algorithm [15] may overestimate the potential stale references by summarizing the flow information from multiple flow paths. By defining the stale data reference sequence as def-def and def-use chains, we are able to eliminate all conservative estimations of the previous algorithm resulting from control dependences.

## A.2 Algorithms

**Algorithm A.1:** *Stale Reference Detection for Single-Word Cache Lines*

**Input**: DEF(S), READ_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;
     OUT(S) ← DEF(S)_0 ⊕ READ_DEF(S)_0;
     RD(S) ← φ;
     IN'(S) ← φ;
     OUT'(S) ← φ;
     STALE(S) ← φ;
 end for
 while changes to OUT(S) or OUT'(S) occur for any statement S do
        for each statement S do
            for each incoming edge p and its corresponding predecessor P of S do
                if p is a scheduling edge then
                  increment the offset of all the definitions in OUT(P) and OUT'(P);
                IN(S) ← IN(S) ⊕ OUT(P);
                IN'(S) ← IN'(S) ⊕ OUT'(P);
            end for
            for each definition d1 of variable v in DEF(S) do
                create a new data descriptor new1 where
                    subarray(new1) ← φ
                    offset(new1) ← 0
                for each definition d of variable v with offset(d) > 0 in IN(S) do
                    if (subarray(d1) ⊗ subarray(d)) ≠ φ then
                      subarray(new1) ← subarray(new1) ⊕ (subarray(d1) ⊗ subarray(d));
                    end if
                end for
                for each descriptor old1 of variable v in RD(S) do
                    subarray(old1) ← subarray(old1) ⊕ subarray(new1);
                end for
                delete the data descriptor new;
            end for
            for each use u of variable v in USE(S) do
                create a new data descriptor new2 where
                    subarray(new2) ← φ
                    offset(new2) ← 0
                for each definition d2 of variable v with offset(d2) > 0 in IN'(S) do
                    if (subarray(u) ⊗ subarray(d)) ≠ φ then
                      subarray(new2) ← subarray(new2) ⊕ (subarray(u) ⊗ subarray(d2));
                      if (offset(d2) < offset(new2)) then
                        offset(new2) ← offset(d2); /* find the minimum offset */
                      end if
                    end if
                end for
                for each descriptor old2 of variable v in STALE(S) do
                    subarray(old2) ← subarray(old2) ⊕ subarray(new2);
                    if (offset(new2) < offset(old2)) then
                      offset(old2) ← offset(new2);
                end for
                delete the data descriptor new2;
            end for
            OUT(S) ← OUT(S) ⊕ IN(S);
            OUT'(S) ← OUT'(S) ⊕ IN'(S) ⊕ RD(S);
        end for
 end while
End
```

Figure 16: Stale reference detection algorithm for single-word cache lines. The USE(S) used in the algorithm can be replaced by TARGET(S) after the target reference detection.

**Algorithm A.2** *Stale Reference Detection2*

**Input**: Epoch flow graph G and 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;
       OUT(S) = DEF(S)_0;
       /* assume all actual parameters and global variables may be modified at each call site */
       if S is CALL statement then
          OUT(S) = OUT(S) + {all the actual parameters and global variables}
       STALE(S) = φ;
  end for
  /* count the minimum offset from the beginning */
  min_offset = 1; /* assume conservatively all global variables and
                     formal parameters are modified in the previous epoch */
  while changes to STALE(S) occur for any statement S do
         for each statement S do
              for each incoming edge p and its corresponding predecessor P of S do
                   if p is a scheduling edge then
                      increment the offset of all the definitions in OUT(P) and OUT'(P);
                      increment the min_offset;
                   IN(S) = IN(S) ⊕ OUT(P);
              end for
              for each use u of variable v in USE(S) do
                   create a new data descriptor new where
                       subarray(new) ← φ
                       offset(new) ← ∞
                   for each definition d of variable v with offset(d) > 0 in IN(S) do
                        if (subarray(u) ⊗ subarray(d)) ≠ φ then
                           subarray(new) ← subarray(new) ⊕ (subarray(u) ⊗ subarray(d));
                           if (offset(d) < offset(new)) then
                              offset(new) ← offset(d); /* find the minimum offset */
                           end if
                        end if
                   end for
                   for each descriptor old of variable v in STALE(S) do
                        subarray(old) ← subarray(old) ⊕ subarray(new);
                        if (offset(new) < offset(old)) then
                           offset(old) ← offset(new);
                   end for
                   delete the data descriptor new;
                   if u is FORMAL or GLOBAL then
                   /* assume conservatively all formal variables and global variables
                       have been modified before entering the procedure */
                       create a new data descriptor side where
                       STALE(S) = STALE(S) ⊕ {side} where
                       offset(side) = min_offset;
                   end if
              end for
              OUT(S) = OUT(S) ⊕ IN(S);
         end for
  end while
End
```

Figure 17: A modified intraprocedural 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.

```
Algorithm A.3: Interprocedural Stale Reference Detection
Input:         Procedure call graph G and TARGET(S) for each statement S
Output:        STALE(S) for each statement S
Begin
  for bottom-up search of G do /* side effect analysis */
    for each leaf node in the current level of G do
      OUT(P) = φ; /* OUT(P) denotes summary side effect for P */
      for each procedure P do
        perform the intraprocedural algorithm in Figure 9 with the following modification;
        for each CALL statement C in P to Q do
          translate OUT(Q) from Q's context to P's context;
          add min_offset(Q) to all the definitions in OUT(C) which have no side effects in Q;
        end for
          mark Time-Read issued due to hidden context;
        for each return node R in P do
          /* side(R) denotes the side effect of P at R */
          side(R) = OUT(R) - {definitions of the local variables of P};
          OUT(P) = OUT(P) ⊕ side(R) where
            choose the minimum offset among the definitions for the same variable in OUT(P) ;
        end for
      end for
    end for
    increment the level by 1;
  end for
  for top-down search of G do /* context analysis */
    for each top level node in G do
      IN(P) = φ; /* IN(P) denotes summary context for P */
      for each procedure P do /* merge the context */
        for each caller Q do
          translate context(Q,P) from Q's context to P's context;
          IN(P) = IN(P) ⊕ context(Q,P) where
            choose the minimum offset among the definitions for the same variable in IN(P);
        end for
        perform the intraprocedural algorithm in Figure 9 with the following modification;
        for each CALL statement C in P to Q do
          /* context(P,Q) denotes summary context from P to Q */
          context(P,Q) = context (P,Q) ⊕ IN(C) - {definitions of local variables of P};
        end for
        for each Time-Read R marked in the first pass do
          if there is a definition d in IN(P) where (subarray(d) ⊕ subarray(R)) ≠ φ then
            add offset(d) to the offset of Time-Read;
          else if there is no overlapping definition d in IN(P) then
                replace the Time-Read with a regular Read;
          endif
        end for
      end for
    end for
    decrement the level by 1;
  end for
End
```

Figure 18: 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.