Interprocedural Array Data-Flow Analysis for Cache Coherence *

Lynn Choi[1] and Pen-Chung Yew[2]

[1] Center for Supercomputing Research and Development, University of Illinois, Urbana, IL 61801-1351, USA
[2] Department of Computer Science, University of Minnesota, Minneapolis, MN 55455-0519, USA

Abstract. *The presence of procedures and procedure calls introduces side effects, which complicate the analysis of stale reference detection in compiler-directed cache coherence schemes [4, 3, 10]. Previous compiler algorithms use the invalidation of an entire cache at procedure boundary [5, 8] or inlining [8] to avoid reference marking interprocedurally. However, frequent cache invalidations will result in poor performance since locality can not be exploited across the procedure boundary. Also, the inlining is often prohibitive due to both its code expansion and increase in its compilation time and memory requirements. In this paper, we introduce an improved intraprocedural and interprocedural algorithms for detecting references to stale data. The intraprocedural algorithm can mark potential stale references without relying on any cache invalidation or inlining at procedure boundaries, thus avoiding unnecessary cache misses for subroutine local data. The interprocedural algorithm performs bottom-up and top-down analysis on the procedure call graph to further exploit locality across procedure boundaries.*

1 Introduction

Procedure calls introduce complications in most global compiler analysis and complicate optimizations due to its side effects and potential aliasing caused by parameter passing. Stale access detection [5, 10] is a compile time analysis to identify potential references to stale data in compiler-directed coherence schemes [4, 3, 10]. By identifying these potential stale references at compile time, cache coherence can be maintained by forcing those references to get up-to-date data directly from the main memory, instead of from the cache. In stale reference detection, procedure boundaries force all previous algorithms [5, 8] to use conservative approaches such as cache invalidation or inlining to avoid reference marking across procedure calls.

In this paper, we develop both intraprocedural and interprocedural compiler algorithms, both of which can perform stale reference detection without relying on either invalidation or inlining. The algorithm is based on a combination of

* This work is supported in part by the National Science Foundation under Grant No. MIP 89-20891, MIP 93-07910.

interval and def-use chain data-flow analysis performed on a modified program control flow graph which represents both parallel program constructs as well as procedure control flow. To obtain more precise array access information, we compute the array region referenced by each array reference. The interprocedural algorithm performs bottom-up and top-down analysis on the procedure call graph to exploit cache locality across procedure boundaries. The first bottom-up *side effect analysis* eliminates side effects by summarizing the access information at each call site. The second 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. The compiler marking algorithms developed here are general enough to be applicable to other compiler-directed coherence schemes [3, 4].

We also propose a new condition for a stale access which identifies the memory reference sequences leading to stale accesses at compile time. The condition considers RAW (read-after-write) and WAW (write-after-write) dependencies caused by multi-word cache lines.

1.1 Parallel execution model

The execution of a parallel program can be 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. The figure 1 shows an example program and its epochs at runtime.

A *task* is a unit of computation that gets scheduled and assigned to a processor at runtime. In a serial epoch, a single task is executed on a single processor. In a parallel epoch, multiple tasks are scheduled and executed on multiple processors concurrently. Therefore, each task boundary corresponds to an epoch boundary where processor scheduling occurs. Synchronizations are assumed at each epoch boundary and at every synchronization points, memory should be made up-to-date by writing back to memory. Barrier synchronizations are used at the end of parallel epochs.

1.2 Memory reference patterns for stale references

Let's first define the ordering of events which leads to a stale reference. The following sequence of events creates a stale access [17]: (1) a read/write to a memory location x by the processor i; (2) a write to x by another processor j ($\neq i$); (3) a read of x by the processor i. The first read or write reference will create an initial cache copy of the data x in processor i, and the second write reference will create a new copy of x in processor j's cache, making the copy in processor i's cache stale. The following read of x by processor i becomes a stale reference.

With multi-word cache lines, there can be implicit dependences due to line aliasing. Let's assume a 2-word cache line containing variables x and y. Assume that processor i's read reference to variable y in epoch a causes a cache miss. An entire cache line is fetched into the cache, creating a copy for each variable. However, in the same epoch a, another processor j ($\neq$ i) writes to the variable x.

The write creates a new cache copy in processor j's cache, making the cache copy in the processor i stale. The following reference to the variable x by processor j in epoch b ($> a$) becomes a stale reference. The read-write dependence between the variables in the same cache line creates a false sharing effect. With write-allocate policy, similar situations can occur for a write reference. These causes implicit RAW(read-after-write) or WAR(write-after-read) dependences between the two tasks in the same epoch even for doall types of parallel loops. To handle the line aliasing with multi-word cache lines, the sequence of events (1) a write, (2) one or more epoch boundaries, and (3) a read is sufficient to create a potential stale reference. We call this sequence of events a *stale reference sequence*.

1.3 Coherence mechanism and hardware support

In our cache coherence scheme, each epoch is assigned a unique epoch number which is similar to the version number in previous schemes [3, 6, 7, 14]. The epoch number is stored in an n-bit register in each processor, called *epoch counter* ($R_{counter}$), and is incremented at the end of every epoch by each processor individually. Every word in a cache is associated with an n-bit timetag that records the epoch number when the cache copy is created. The timetag is updated with the value of the current epoch counter during a cache miss or a write operation. On a cache line reload (either due to a read miss or a write-allocate), the timetags for the remaining words of the cache line are assigned the value, $R_{counter}$ - 1, to avoid the line aliasing problem (refer to section 1.2).

In addition to normal read and write operations, one extra memory operation, called Time-Read, is required to support our coherence scheme. A Time-Read is a special cache read operation to check for a potential stale access. A Time-Read is similar to a Read except that it is augmented with an *offset*. The *offset* indicates the number of epoch boundaries between the current epoch and the epoch in which the data was last updated. On a Time-Read, in addition to checking the address tag and the valid bit, the timetag of the cache word is tested to determine a cache hit. If (1) timetag $\geq$ ($R_{counter}$ - offset), which checks whether the cache copy is up-to-date, (2) the address tag matches, and (3) the valid bit is set, then we have a cache hit. For correct coherence enforcement, the compiler needs to generate appropriate memory and cache management operations. First, the compiler should insert the epoch counter increment operation at the end of each epoch. Second, potential stale read references need to be identified and issued as Time-Read operations. For such references, the compiler also needs to calculate their offset values.

Consider the program example in figure 1(a). Since a copy of Y is modified by a processor in epoch 1, any read reference to the variable Y in the following epochs is a potential stale access and should be issued as a Time-Read. The Time-Read operation to Y in epoch 2 will have an offset 1 since the last write occurs in epoch 1 while the Time-Read operations in epoch 3 are assigned an offset 2. Note that the read references to X in epoch 4 are not marked as Time-Read. Treating an array as a single variable, the compiler will mark read references to X in both epoch 3 and epoch 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 in epoch 4 as safe because the array regions accessed in epoch 2 and epoch 4 are distinct, so the array elements addressed by the read references to X in epoch 4 have not been modified. All the references to the variable W are safe because the variable W is read-only. The staleness of the data is determined at runtime by using this Time-Read instruction because it will check whether the cache data is created after the most recent write by checking the condition (1).

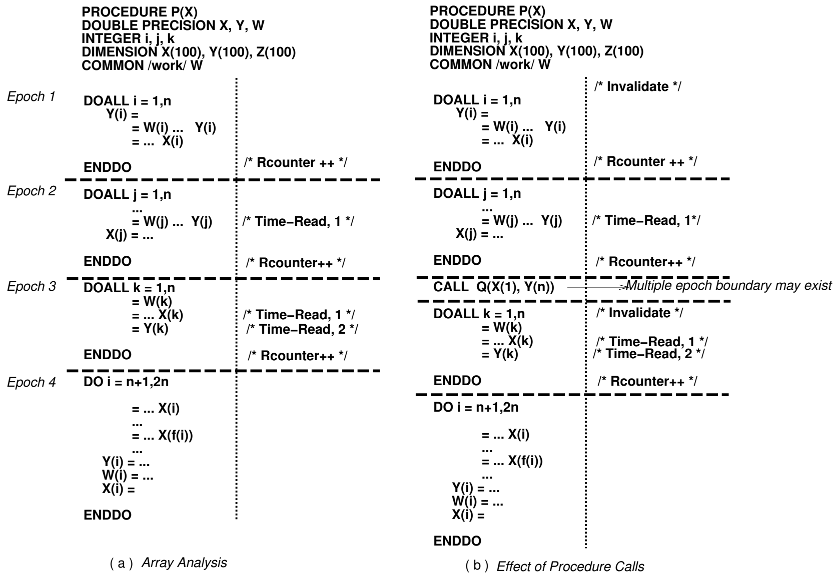

Fig. 1. A program example of compiler marking of Time-Reads.

Effect of procedure calls Procedure calls complicate the analysis of stale reference detection. Let's look at the program example in figure 1(b) where a procedure CALL statement is inserted between the epoch 2 and the epoch 3. Since FORTRAN uses call-by-reference parameter passing mechanism, the parameters X and Y can be modified in the procedure Q. Moreover, all the global COMMON variables such as W in the example can also be modified in the procedure Q. This complicates the analysis of stale reference detection since an intraprocedural algorithm does not have the access information from the procedures called. In addition, this also causes another problem at the beginning of procedure since any global COMMON variables and formal parameters could have been previously modified before entering the procedure. To avoid the complications caused by procedure calls, previous algorithms [5, 8] use cache invalidation both at the beginning of a procedure and after each call site. Since the algorithms assume a clean cache at procedure boundaries, their analysis can guarantee the correctness of reference marking. However, note that such invalidations cause unnecessary cache misses for global COMMON variables and parameters as well as for subroutine local variables. Frequent invalidations at procedure boundaries will degrade cache performance significantly because it limits the scope of locality within procedural boundaries. From our execution-driven simulations on Perfect benchmarks [8], the performance degradation could be significant especially if a program contains many small procedures.

In the following, we first describe our program representation methods: epoch flow graph [10], array descriptors, and GSA in section 2. Then, we present an intraprocedural stale reference marking algorithm which can detect stale data references without cache invalidation or inlining in the presence of procedure calls. In section 3, we develop an interprocedural algorithm which performs incremental 2-pass analysis on the procedure call graph to further exploit locality across procedure boundaries. Section 4 concludes the paper.

2 Intraprocedural Algorithm

2.1 Basics

We consider the problem of identifying potential stale references in a data-flow framework. A data-flow analysis examines the flow of values through a program by propagating information along the paths of a control flow graph. The data-flow information collected during the analysis is based on a modified version of the reaching definition algorithm in a conventional data-flow analysis. A write to a variable v is called a *definition* of the variable v while a read is called a *use* of the variable. An *upwardly-exposed use* of a variable in a block of statements, B, is a read reference which is not preceded by any previous definition of the same variable in B. [3] The first occurrence of the *upwardly-exposed uses* of a variable in an epoch is called a *target reference*. Note that only the target reference can be a potential stale access. The following read accesses in the epoch will not be stale because the cache copy will have been made up-to-date by the target reference if it was stale. The rest of the memory references can thus be marked as regular reads or writes.

From the stale reference sequence in section 1.2, we now need to look for the following def-use chains to determine a potential stale data reference: (1) a definition, (2) one or more epoch boundaries, and finally (3) a target reference.

Data structure for array references The data sets propagated during the flow analysis are implemented as sets of data descriptors. For each memory reference, we associate a data descriptor D, which contains the following fields.

- **name(D)** the variable name
- **subarray(D)** the region of the variable being referenced

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

– **offset(D)** the offset for the Time-Read operation

The subarray field represents the region of the array that are accessed by the array reference. The notion of subarray we use is an extension to the *regular section* used in [2, 12]. A subarray consists of a subscripted variable and one or more ranges for each index in the subscript expression. A range includes expressions for the lower bound, upper bound and stride. The offset field keeps track of the number of epoch boundaries crossed for each definition propagated.

Gated single assignment (GSA) form GSA is an extension of static single assignment (SSA) form. Like SSA, GSA automatically renames the variable, so we can determine whether the two array references are referring to the same element of X by comparing their index expressions. Similarly, by representing the subarray fields in the GSA form, we can perform subarray operations involving symbolic loop bounds. In addition, GSA allows a backward demand-driven symbolic analysis to compute values and conditions across the confluence points [16]. Figure 2 shows the example program and its GSA form which will be used throughout the discussion in this section.

Because the epoch boundary information is essential in identifying potential stale references, we include parallel constructs in our control flow graph, called the *epoch flow graph* [10].

Epoch flow graph Let the directed graph G = (V, E) represent a control flow graph where V is a set of statements, and E is a set of directed edges, representing the control flow between nodes in V. We define the *epoch flow graph* G' = (V ∪ 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 divide the edges into two types. The edge into the beginning of a parallel loop and the edge exiting a parallel loop are called *scheduling edges*. The 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 statement which has an incoming *scheduling* edge. The start node is a special head node which does not have any incoming *scheduling* edge.
– *Tail Node*: A statement which has 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 the *epoch level from H*. There is an one-to-one relationship between a head node and an epoch level. Multiple tail nodes can exist in L. A directed path from H to each tail node T is called an *epoch from H to T*.

```
         PROCEDURE P(x)
         DOUBLE PRECISION x, y
         INTEGER*4 i, num
         DIMENSION x(200), y(200)
         COMMON /work/ y

S1       PRINT *, 'PROGRAM START'
CSRD$ PARALLEL (i)
S2       DO i = 1, 100, 1


S3        x(i) = i
S4        y(i) = i
S5       ENDDO
S6       DO num = 1, 10, 1



S7        IF (15+5*num.LE.30) THEN
S8         DO i = 1, 15+5*num, 1


S9          CALL Q(x(1), y(1))
S10          y(i) = x(i+(15+5*num))

S11         x(i+i) = x(i+(15+5*num))

S12        ENDDO
S13       ELSE
CSRD$ PARALLEL (i)
S14        DO i = 1, 15+5*num, 1

S15         x(14+i+5*num) =
              x((-1)+i)+y(14+i+5*num)

S16        ENDDO
S17       ENDIF



S18       x(15+5*num) = (15+5*num)+1

S19      ENDDO
S20      PRINT *, 'PRINT RESULT'
S21      DO i = 101, 200, 1
S22       PRINT *, x(i), y(i)
S23      ENDDO
S24      STOP
S25      END
```

```
      PROCEDURE P(x)
      DOUBLE PRECISION x, y
      INTEGER*4 i, num
      DIMENSION x(200), y(200)
      COMMON /work/ y

      PRINT *, 'PROGRAM START'
CSRD$ PARALLEL (i)
      DO i1 = 1, 100, 1
       y1 = <MU>(y0, y2)
       x1 = <MU>(x0, x2)
       x2(i1) = <ALPHA>(x1, i1)
       y2(i1) = <ALPHA>(y1, i1)
      ENDDO
      DO num1 = 1, 10, 1
       y3 = <MU>(y1, y6)
       x3 = <MU>(x1, x7)
       i2 = <MU>(i1, i4)
       IF (15+5*num1.LE.30) THEN
        DO i3 = 1, 15+5*num1, 1
         y4 = <MU>(y3, y5)
         x4 = <MU>(x3, x5)
         CALL Q(x4(1), y4(1))
         y5(i3) = <ALPHA>(y4,
           x4(i3+(15+5*num1)))
         x5(i3+i3) = <ALPHA>(x4,
           x4(i3+(15+5*num1)))
        ENDDO
       ELSE
CSRD$ PARALLEL (i)
        DO i5 = 1, 15+5*num1, 1
         x8 = <MU>(x3, x9)
         x9(14+i5+5*num1) =
           <ALPHA>(x8, x8((-1)+i5)
           +*y3(14+i5+5*num1))
        ENDDO
       ENDIF
       y6 = <GA>(15+5*num1.LE.30, y4, y3)
       x6 = <GA>(15+5*num1.LE.30, x4, x8)
       i4 = <GA>(15+5*num1.LE.30, i3, i5)
       x7(15+5*num1) = <ALPHA>(x6,
         (15+5*num1)+1)
      ENDDO
      PRINT *, 'PRINT RESULT'
      DO i6 = 101, 200, 1
       PRINT *, x3(i6), y3(i6)
      ENDDO
      STOP
      END
```

Fig. 2. A program example and its GSA form

Figure 3 shows the control flow graph for the program example in figure 2 and its corresponding epoch flow graph. Note that the notion of an epoch is *dynamic* in that a section of code can belong to different epochs at runtime depending on the branch taken. For example, in figure 3, the statement S17 can belong to the epoch from S17 to S23 (The epoch consists of nodes S17, S18, S6, S19, S20, S21, S22 and S23) or the epoch from S6 to S13 (The epoch consists of nodes S6, S7, S8, S9, S10, S11, S8, S16, S17, S18, S6, S7, S12, and S13 if S9 does not contain any parallel loop.).

2.2 Overall algorithm

First, we transform the source program to a GSA form, which is used to refine symbolic analysis involving array references. Then, we construct the *epoch flow graph*, which contains the epoch boundary information as well as the control flows of the program. Given a source program unit and its epoch flow graph G, we mark target references in each epoch for potential stale references. After marking the

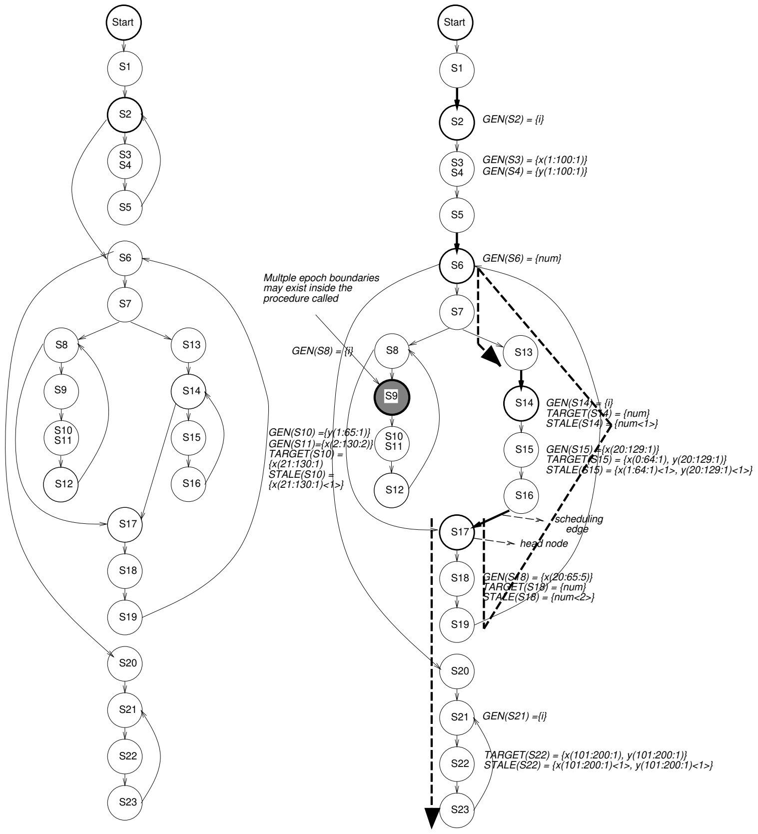

Fig. 3. The control flow graph and the corresponding epoch flow graph for the program example. A bold arc denotes a scheduling edge while a normal arc denotes a flow edge. Head nodes are represented as bold nodes. Call sites are denoted as shaded nodes. The bold-dotted paths show two instances of the epoch where statement S17 can belong at runtime. The epoch flow graph also shows the set of definitions GEN(S), TARGET(S), and STALE(S) computed for each statement according to the intraprocedural algorithm.

target reference, regions of array that are referenced in each epoch are computed. Then, a stale reference detection algorithm determines the existence of stale memory reference sequences and compute offsets for the target references.

If a stale reference sequence is found for a target reference, the reference is marked as a Time-Read operation. Finally, we transform the program in its GSA form back to the original program with the reference marking information.

Target reference marking For our analysis, we are interested in finding only the first occurrence of the upwardly-exposed uses. To simplify the discussion, we refer only the first occurrence of upwardly-exposed uses as an upwardly-exposed use. Therefore, a read following a read will not be marked as an upwardly exposed use in our analysis. The detection of upwardly-exposed uses of a scalar variable can be done precisely using the variable names. However, for array variables, a reference to an array may not access the same region as the one accessed by an earlier reference to the same variable. Hence, both references may have some upwardly-exposed uses in portion 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. We use the algorithm proposed in [8] to mark the target references.

2.3 Stale reference detection

The intraprocedural algorithm for stale reference detection is an improved version of the previous algorithm we developed in [8]. We refine the algorithm both to accommodate multi-word cache blocks as well as to eliminate cache invalidation at procedure boundaries.

Each definition of a variable v, denoted as d^v_{offset}, is associated with an *offset*. Initially, an offset 0 is assigned to each definition, and is incremented when the definition crosses an epoch boundary. The offset represents the number of epoch boundaries crossed. Multiple definitions of a variable with the same offset are considered as a single definition. Therefore, multiple writes in an epoch are considered as a single definition. Because of the processor reassignments at each epoch boundary, a definition in an epoch can reach any following epoch without being *killed*. Therefore, the notion of *kill* used in the traditional reaching definition algorithm is no longer necessary.

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 definition d of variable v in statement S2 to S1 including at least one scheduling edge.

When a definition reaches a target reference across at least one scheduling edge, there should exist the sequence of events from (1) to (3), and the target reference to v should be marked as potential stale and be issued as a Time-Read operation. Since we increment the offset of a definition when we propagate the definition across epoch boundaries, the offset of the definition shows the number of epoch boundaries crossed since the write, and used as an offset for the Time-Read operation.

For each statement S, we define the following sets.

- $GEN(S)_{offset}$ is the set of definitions generated by S. Since all the definitions created in S are assigned the same offset, the offset can be denoted collectively for S as $GEN(S)_{offset}$.
- *IN(S)* is a set of definitions reaching the beginning of S, IN'(S) = $\bigcup_P$ 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) $\cup$ $GEN(S)_0$.

- *STALE(S)* is a subset of TARGET(S) where there exists a definition reaching at the beginning of S across epoch boundaries, STALE(S) = {use u of the variable v: $u \in$ TARGET(S) and $\exists$ a definition d of the variable v in IN(S) where offset(d) > 0 and (subarray(u) $\cap$ subarray(d)) $\neq \phi$ }.

Complications in the presence of procedure calls The presence of procedures and procedure calls introduces the following complications in this intraprocedural algorithm for 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. The variables include the actual parameters given 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 could have been read or written previously at the beginning of the procedure.
- **aliases** The third issue is the static and dynamic aliases caused by call-by-reference parameter passing and EQUIVALENCE statement of FORTRAN.
  1. **static alias** EQUIVALENCE statement causes 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. The aliases makes our analysis no longer valid even if we invalidate the cache at procedure boundaries. However, ANSI FORTRAN does not allow programs written with aliases. Therefore, we do not consider the dynamic aliases in the following algorithm. If aliases do occur in a source program, then we use selective inlining if possible, or treat all the aliased variables as a single variable.

The intraprocedural algorithm works as follows. [4] Because of the unknown context information, we assume all the formal parameters and global variables have been previous modified at the beginning of procedure. This is accomplished by keeping track of minimum offset from the beginning of a procedure. For a target reference which does not have a reaching definition inside a procedure, we issue a Time-Read with the minimum offset, implying the data item referenced can be potentially modified before entering the procedure. This is an improvement over previous algorithms [5, 8] which use cache invalidation at the procedure beginning 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 the 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 having any information for the procedure called, we have

[4] The more detail algorithm is presented in [9].

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 too small if the procedure call has at least one parallel epoch inside. This small offset can result in unnecessary cache misses at runtime. This is because the epoch counter will be incremented at runtime by the number of epochs inside the procedure, which will cause Time-Reads to miss due to the conservative offset values. For each target reference, 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. To refine stale reference marking for an array reference, we take the intersection of subarrays of a reaching definition 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. Figure 3 shows the result of reference marking for the program example in figure 2.

3 Interprocedural Analysis

Until now, the algorithm presented assumes program units, i.e. single procedures, and therefore single flow graphs. The obvious drawback of such an algorithm is that it can not exploit locality across procedure boundaries. To further exploit locality, we need to look at an entire program rather than a program unit at a time. To perform the interprocedural analysis, we use a *procedure call graph*, which is the basic data structure for the interprocedural analysis.

Definition: procedure call graph Let a directed multigraph 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)
- Numbers/types of formal parameters and global variables
- Numbers/types of actual parameters for each call site (call site information)
- Summary side effect information (OUT set of callee for each call site)
- Summary context information (IN set of caller at the beginning of procedure)
- Number of minimum epoch boundaries in a procedure

3.1 Side effect analysis

Figure 4 shows the overall structure of interprocedural analysis. It consists of 2 passes on the procedure call graph: the bottom-up side effect analysis pass and the top-down context analysis pass. [5]

[5] A complete algorithm is presented in [9].

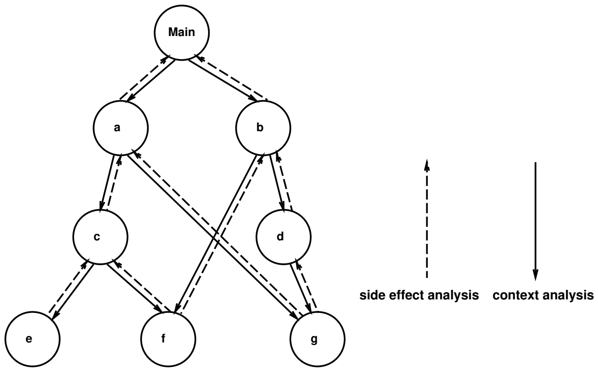

Fig. 4. An example of procedure call graph and the overall structure of the two-pass interprocedural analysis.

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 effect of each call site is summarized by the OUT set computed from the called procedure. We first start at the bottom of the procedure call graph. We apply the intraprocedural algorithm described in section 2 for all the leaf procedures. This can be performed without considering side effects since those procedures do not include procedure calls. After performing the intraprocedural analysis, we summarize the information and attach the summary side effect information to the procedures which 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. The summary side effect information should contain the following information for each actual parameter and global variable.

- whether the variable is modified or not
- the number of epoch boundaries from the last write (offset)
- the region of array that have been modified (subarray)

Note that all the above information can be represented by OUT set of the procedure called, which is already computed by the intraprocedural analysis. Since a procedure can have multiple return points, the summary information requires merging the OUT sets of all the return points. 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 actual parameter. 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, the minimum number of epoch boundaries crossed during the execution of the callee is necessary. This is because we need to consider the number of epoch boundaries crossed inside a call site to compute the offset for Time-Reads. To do so, after each call site, we add the number of epoch boundaries crossed in the call site to each definition in the OUT set of the CALL statement. Without this, the offsets for Time-Reads will not reflect the effect of the call site, which may generate offsets too small to capture the locality across the procedure boundaries.

3.2 Context analysis

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

We start from a main procedure. Since the main procedure does not have any context at the beginning, the previous bottom-up analysis already generates the precise result for the main program unit. But, we still 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 the callee to be replaced by the IN set of the CALL statement. Since the context information is necessary only for formal parameters and global variables of the callee, we only propagate the summary context information for those variables. In opposite to the side effect analysis, at this time we need to translate the IN set information from caller's context to 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 (different code for each case), which, in the worst case, produces the same effect as the inlining.

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

Using the bottom-up and top-down analyses, we can eliminate many redundant computation by performing minimal number of computation (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, for each procedure, we don't have to propagate the data-flow information again since the summary context information is enough to refine reference marking. We only need to add the offset of each variable in the summary context information at the beginning of a procedure to the Time-Reads for those variables. For the Time-Reads marked which do not have definitions in the summary context, we could eliminate the Time-Reads completely since there have been no previous update to the variable referenced by those Time-Reads.

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 only need to load a 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 Conclusion

We propose improved compiler algorithms for detecting stale data references in the presence of procedures calls. Procedure calls can introduce side effects at a call site and hidden context at the beginning of a procedure, which can limit compiler-directed coherence schemes [2, 4, 3, 10, 13, 14] to exploit locality only within procedure boundaries. Previous algorithms use cache invalidation [5, 8] or selective inlining [8] to solve the problem. However, invalidation at procedure boundaries incur significant performance penalty especially if a program contains many small procedures in its critical path. Inlining allows the most precise analysis but often prohibitive due to its potential code size expansion as well as the compile time increase. By detecting target references, we propose an improved compiler algorithm which avoids the invalidation at procedure boundaries. We selectively issue Time-Reads only for procedure parameters and global COMMON variables which are shared by multiple procedures. By doing so, the procedure boundary will not affect the access of subroutine local data. We also propose an interprocedural algorithm which can exploit locality across procedure boundaries. It performs interprocedural analysis according to a bottom-up and a top-down order 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, generating the context information for its intraprocedural analysis. This two-pass algorithm eliminates redundant data-flow computation for each program unit by doing minimal number of computation per program unit. In addition, the top-down pass updates the reference marking result of the side effect analysis incrementally, minimizing the compilation time. We are currently implementing these algorithms on the Polaris parallelizing compiler [15].

References

1. 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.
2. 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.
3. Hoichi Cheong. Life Span Strategy - A Compiler-Based Approach to Cache Coherence. *Proceedings of the 1992 International Conference on Supercomputing*, July 1992.
4. Hoichi Cheong and Alex Veidenbaum. A Cache Coherence Scheme with Fast Selective Invalidation. *Proceedings of The 15th Annual International Symposium on Computer Architecture*, page 299, June 1988.
5. Hoichi Cheong and Alexander V. 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.
6. Hoichi Cheong and Alexander V. Veidenbaum. A Version Control Approach To Cache Coherence. *Proceedings of 1989 ACM/SIGARCH International Conference on Supercomputing*, June 1989.
7. T. Chiueh. A Generational Approach to Software-Controlled Multiprocessor Cache Coherence. *Proceedings 1993 International Conference on Parallel Processing*, 1993.
8. Lynn Choi and Pen-Chung Yew. Eliminating Stale Data References through Array Data-Flow Analysis. *CSRD Technical Report No. 1425*, April. 1995.
9. Lynn Choi and Pen-Chung Yew. Interprocedural Array Data-Flow Analysis for Cache Coherence. *CSRD Technical Report No. 1427*, May. 1995.
10. Lynn Choi and Pen-Chung Yew. A Compiler-Directed Cache Coherence Scheme with Improved Intertask Locality. *Proceedings of the Supercomputing'94*, November 1994.
11. Mary W. Hall. Managing Interprocedural Optimization. Technical report, Rice University, Dept. of Computer Science, April 1991. Ph.D. Thesis.
12. Paul Havlak. Interprocedural Symbolic Analysis. Technical report, Rice University, Dept. of Computer Science, May 1994. Ph.D. Thesis.
13. 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:I–2–I–9, August 1992.
14. S. L. Min and J.-L. Baer. A Timestamp-based Cache Coherence Scheme. *1989 International Conference on Parallel Processing*, I:23–32, 1989.
15. 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.
16. P. Tu and D. Padua. Gated SSA Based Demand-Driven Symbolic Analysis. *CSRD Technical Report No. 1336*, Feb. 1994.
17. 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.

This article was processed using the LaTeX macro package with LLNCS style