Program Analysis for Cache Coherence: Beyond Procedural Boundaries *

Lynn Choi
Center for Supercomputing R & D
University of Illinois
Urbana, IL 61801-1351
lchoi@csrd.uiuc.edu
(217)333-0969
(217)244-1351 (fax)

Pen-Chung Yew
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455-0519
yew@cs.umn.edu
(612)625-7387
(612)625-0572 (fax)

Abstract

The presence of procedures and procedure calls introduces side effects, which complicates the analysis of stale reference detection in compiler-directed cache coherence schemes [6, 4, 8]. Previous compiler algorithms use cache invalidation at procedure boundary [5, 7] or inlining [7] to avoid reference marking interprocedurally. However, frequent cache invalidations will result in poor performance since locality cannot be exploited across the procedure boundary. Also, the inlining is often prohibitive due to both its code expansion and increase in compilation time and memory requirements. In this paper, we introduce 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. The result of execution-driven simulations on Perfect benchmarks demonstrates that by avoiding cache invalidations, the intraprocedural algorithm eliminates up to 26.0% of the cache misses for a compiler-directed scheme compared to an existing invalidation-based algorithm [7]. With the full interprocedural analysis, up to 10.8% of additional cache misses can be removed.

Keywords: compiler, interprocedural analysis, data-flow analysis, cache coherence, shared-memory multiprocessors

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

1 Introduction

Procedure calls introduce complications in most global program analysis and optimizations due to side effects and potential aliasing caused by parameter passing. Stale access detection [5, 8] is a compile time analysis technique to identify data references that may violate cache coherence in compile-directed coherence schemes [6, 4, 8]. By identifying these potentially 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, 7] to use conservative approaches, such as cache invalidation or inlining, to avoid reference marking across procedure calls. However, both approaches have their own problems. Frequent invalidations at procedure call boundaries incur cold-start effects. This leads to a poor cache performance especially 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 as well as the increase in compile time since the memory requirement and the complexity of the reference marking algorithm is often nonlinear in terms of the procedure size.

In this paper, we develop both intraprocedural and interprocedural compiler algorithms, both of which can perform stale reference detection without relying on either cache invalidation or inlining. The algorithms are based on a combination of 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 [7]. The interprocedural algorithm performs bottom-up and top-down analysis on the procedure call graph to exploit cache 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. We have implemented both the intraprocedural and interprocedural algorithms on the Polaris parallelizing compiler [11] and demonstrate the performance driven by these algorithms by running execution-driven simulations of two compiler-directed coherence schemes [8, 7] using Perfect benchmarks. The compiler reference marking algorithms developed here are general enough to be applicable to other compiler-directed coherence schemes [4, 6].

1.1 Stale reference sequence

First, let us look at the memory reference pattern leading to a stale data access at runtime. Figure 1(a) shows a sample program and the potentially stale data references marked by a compiler. Such references are denoted as Time-Reads in the figure. We view the execution of a parallel program 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. At each epoch boundary, tasks are scheduled to processors and memory is updated for consistency. In the program example, the read reference to variable X in the epoch 3 can cause a stale data reference at runtime due to the following runtime scenario: (1) a processor P_i reads a variable *X(k)* in epoch 1; (2) another processor P_j ($j \neq i$) writes to *X(k)* in epoch 2;. (3) Processor P_i reads the copy of *X(k)* in P_i's cache in epoch 3. The event (1) will create an initial cache copy of *X(k)* in P_i, and the second write reference will create a new copy of *X(k)* in P_j's cache, making the copy in P_i's cache stale. The following read of *X(k)* by P_i becomes a stale data access. Similarly, a write to *X(k)* in epoch 1 also can create an initial copy of data. Therefore, a sequence of 1) a read or a write , 2) epoch boundaries, 3) a write, 4) epoch boundaries, and 5) the following read can cause the last read to be a stale data access.

With multi-word cache lines, this is not a sufficient condition to cover all the potential stale data accesses at runtime. For example, the read reference to Y in epoch 2 can also cause a stale data reference at runtime. Assume 2-word cache lines and the following runtime scenario. A processor P_i causes a cache miss on a read to Y(1) in epoch 1. On a cache line reload, the cache copies of both Y(1) and Y(2) are brought into P_i's cache. In the same epoch, another processor P_j writes to Y(2), making the copy of Y(1) in P_i 's cache stale. Therefore, if the read of Y(1) is scheduled to the processor P_i in epoch 2, then the read reference to Y(1) by P_i in epoch 2 becomes a stale data access. Note that there is no data dependence in epoch 1. However, there can be implicit dependences due to the false sharing with multi-word cache lines. In general, the sequence of 1) a write 2) followed by epoch boundaries, and 3) the following read can cause the the last read reference to become a stale data access at runtime. This memory reference pattern is called a *stale reference sequence*, and the last read reference is called a *potentially stale* data reference [7]. In the example, the read references to X in epochs 3 and 4 can cause stale data accesses at runtime since the variable has been modified in epoch 2. Note that the first read reference to X in epoch 4 is not marked as potentially stale because the region of array X modified in epoch 2 and the region of X accessed in epoch 4 are distinct.

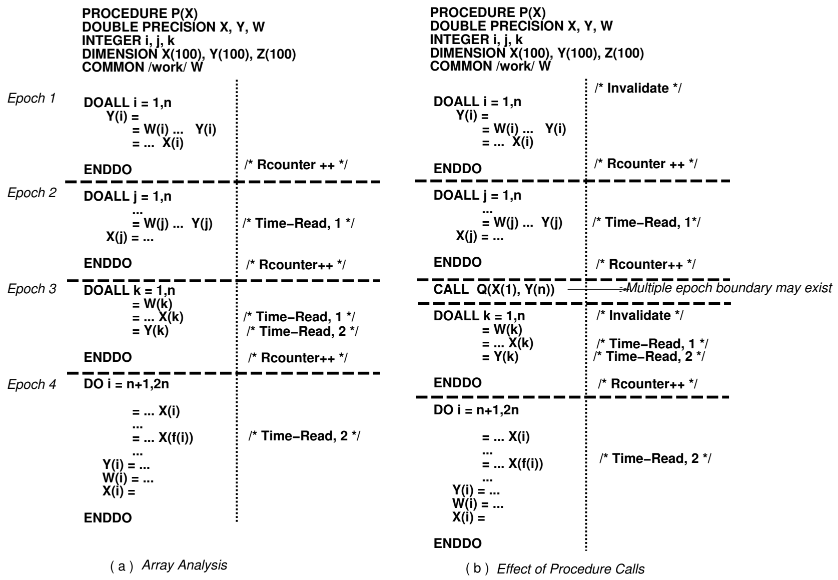

Figure 1: A program example of compiler marking of Time-Reads.

1.2 Coherence mechanism and hardware support

Compiler scheme (CP) 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 done by using a by-pass 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.

However, this pure compiler scheme has several limitations. First, the read reference to X(k) in epoch 3 (see Figure 1) will be marked as potentially stale because the variable X was updated in epoch 2. The scheme cannot determine whether the update in epoch 2 and the read in epoch 3 are issued by the same processor or not, due to unknown runtime scheduling information. Second, the read reference to X(f(i)) in epoch 4 cannot be analyzed precisely at compile time due to the unknown index value. To overcome these limitations, we have proposed a hardware scheme, called the two-phase invalidation scheme that keeps track of the local cache states at runtime [8].

Two-phase invalidation scheme (TPI) In this scheme, each epoch is assigned a unique epoch number. 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 the cache has an n-bit timetag that records the epoch number when the cache copy is created.

In addition to normal read and write operations, a new memory operation, called Time-Read, is used exclusively for a potentially stale reference marked by the compiler. It is similar to the normal 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. At runtime, Time-Read determines whether the cached data is stale or not by comparing the runtime timing information (timetag) of each cache copy with the timing information (offset) provided by the compiler. A Time-Read returns 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.

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, potentially 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.

Let us revisit the program example in figure 1(a). All the potentially stale data references marked by the compiler should be issued as Time-Reads. 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. The staleness of the data is determined at runtime by using this Time-Read operation because it will check whether the cache data is created after the most recent write by checking condition (1). Note that, in this scheme, all the potential stale data references (Time-Reads) will be cache hits if the previous write is performed by the same processor. For example, the Time-Read to X(k) by P_i in epoch 3 will be a cache hit if the write to X(k) in epoch 2 is also issued by P_i, which can be determined by checking condition (1). Also, the second read reference to X in the epoch 4 can be a cache hit if the previous write in epoch 2 is issued by the same processor, or if the memory location is already accessed by the first read reference to X in epoch 4.

Effect of procedure calls Procedure calls complicate the analysis in stale reference detection. To illustrate this, we insert a new procedure CALL statement to the previous program example between the epoch 2 and the epoch 3 as shown in Figure 1(b) Since FORTRAN uses call-by-reference parameter passing mechanism, the parameters X and Y can be modified in the procedure Q. Furthermore, all the global COMMON variables such as variable W in the example can also be modified in the procedure Q. This complicates the analysis in stale ref-

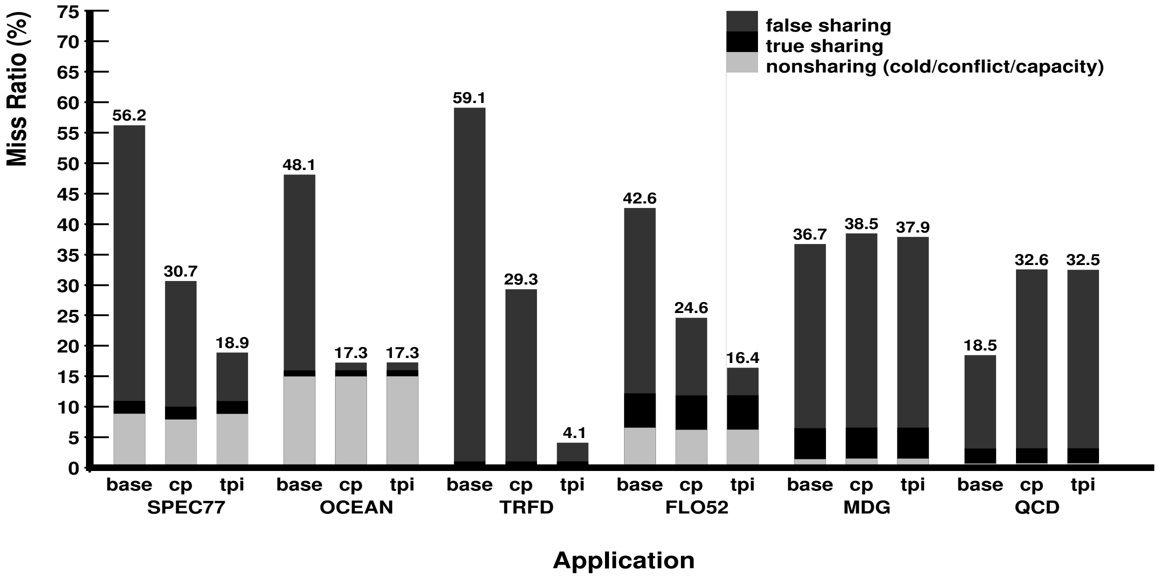

Figure 2: Cache miss rates using an invalidation-based compiler algorithm for the compiler scheme (CP) and the two-phase invalidation scheme (TPI) compared to the BASE underlying architecture.

erence detection since with an intraprocedural algorithm no access information is available at each call site. In addition, this also causes a problem at the beginning of each procedure since any global COMMON variables and formal parameters could have been previously modified before entering the procedure. To avoid such complications caused by procedure calls, previous algorithms [5, 7] 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 subroutine local variables as well as the global COMMON variables and parameters. Frequent invalidations at procedure boundaries can degrade the cache performance substantially since it limits the scope of locality to be exploited within procedural boundaries.

Figure 2 shows the cache miss rates of both the compiler scheme (CP) and the two-phase invalidation scheme (TPI) using an invalidation-based array data-flow algorithm [7] against the underlying machine with no cache coherence support (BASE) which does not cache shared data. The results shown are from our execution-driven simulations on Perfect benchmarks (see section 4 for more details). The miss rates are classified into sharing misses and nonsharing misses. A sharing miss occurs when the address tag matches but either the cache copy has been invalidated or the reference is marked as potentially stale. All the other misses account for nonsharing misses as in a uniprocessor system. The sharing misses are further classified into true and false sharing misses. The true sharing misses are necessary misses to avoid cache coherence, while the false sharing misses are the unnecessary misses caused by conservative compiler decisions.

For the benchmarks MDG and QCD, both compiler-directed coherence schemes (CP and TPI) have significant false sharing misses. Note that they even have higher cache miss rates than the BASE architecture. This is due to the cache invalidations used by the compiler algorithm. The cache invalidation invalidates private data and subroutine local data as well as shared data from the cache. Those private and subroutine local data should not be affected by the interprocedural side effects. In addition, for benchmarks SPEC77 and FLO52, there are still significant false sharing misses which are due to both the invalidations and conservative decisions made by the compiler. This limitation led us to develop more precise program analysis algorithms which can avoid such invalidations.

In section 2, we first briefly describe our program representation methods: epoch flow graph [8], array descriptors, and gated single assignment (GSA). Then, we present an improved intraprocedural stale reference marking algorithm which can detect stale data references in the presence of procedure calls without cache invalidation or inlining. In section 3, we develop a full interprocedural algorithm which performs incremental 2-pass analysis on the procedure call graph to further exploit locality across procedure boundaries. The performance results from our execution-driven simulations follow in section 4. Section 5 concludes the paper.

2 Intraprocedural Algorithm

2.1 Data-Flow Framework

We use the following three techniques as a framework for our array data-flow analysis. A more detailed description can be found in [7].

- **Regular Section Analysis** The data-flow information propagated during the flow analysis are implemented as sets of data descriptors D, which consists of three data fields: name(D), subarray(D), and offset(D). The notion of the subarray we use is an extension to the *regular section* used in [3, 9, 14].

- **Gated Single Assignment (GSA) form** To perform effective array flow analysis, symbolic manipulation of expressions is necessary since the computation of array regions often involves the equality and comparison tests between symbolic expressions. For this purpose, we use the gated single assignment (GSA) form [1] to treat arrays with different access regions as different symbolic variables.

- **Epoch Flow Graph** This is a modified flow graph to represent the epoch boundary information as well as different control flows due to the parallel execution[8]. Figure 3 shows the epoch flow graph for a program example. Bold arcs denote *scheduling edges*, which represent epoch boundaries, while the remaining arcs denote normal control flow edges. A statement which has an incoming scheduling edge is called a *head node*, while a statement which has an outgoing scheduling edge is called a *tail node*. A directed path from each head node to tail node shows the epoch created at runtime.

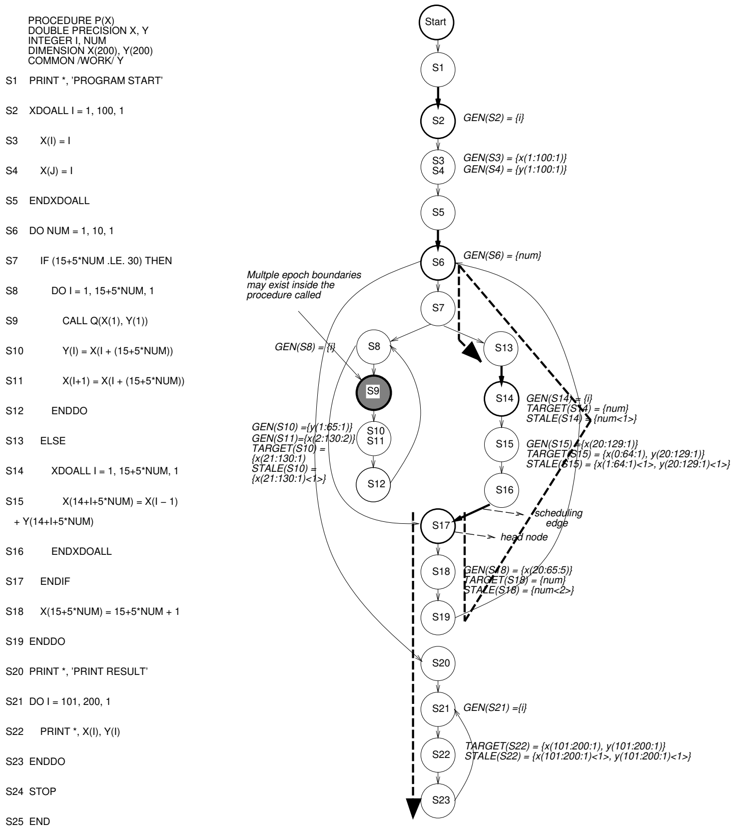

Figure 3: A program example and the corresponding epoch flow graph. The bold-dotted paths show two instances of the epoch where statement S17 can belong at runtime.

Target reference marking Not all the potentially stale references lead to a stale data access at runtime. To minimize the number of those potentially stale references marked at compile time, we utilize both temporal and spatial reuses in a program as much as possible.

To consider the temporal reuses in each task, we mark only the first occurrence of upwardly-exposed uses in each epoch. These references are called *target references*. A detailed array data-flow algorithm to find such target references is shown in [7], and for the following discussion, we assume that those target references are already computed.

2.2 Stale reference detection

The intraprocedural algorithm for stale reference detection is an improved version of the previous algorithm we developed in [7]. We refine the algorithm to eliminate cache invalidations by considering both side effects and hidden contexts at procedural 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 during flow analysis. 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.

Potentially stale reference 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 stale reference sequence (refer section 1.1), and the target reference to v should be marked as potentially 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, $IN(\mathrm{S}) = \bigcup_P OUT(\mathrm{P})$ 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 across epoch boundaries, whose intersection with the target reference has nonempty subarrays, 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 to this intraprocedural stale reference detection algorithm.

- **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 could have been read or written previously at the beginning of a procedure.
- **aliases** The third issue is the aliasing caused by call-by-reference parameter passing mechanism 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. These aliases make 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 is presented in Appendix A.1, and works as follows. Due to the unknown context information, we assume 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 which does not have a 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 [5, 7] which 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. 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.

3 Interprocedural Algorithm

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 cannot exploit locality across procedure boundaries for global variables. To further exploit locality, we need to look at an entire program rather than one program unit at a time. To perform the 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 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)

- 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

3.1 Side effect analysis

Figure 4 shows the overall structure of interprocedural analysis. It consists of 2 passes on the procedure call graph: bottom-up side effect analysis pass and top-down context analysis pass. The detailed algorithm is presented in Figure 12 of Appendix A.2.

```
      PROGRAM MAIN
S1      CALL A
S2      CALL B

      PROCEDURE A
S3      CALL C
S4      CALL G

      PROCEDURE B
S5      CALL F
S6      CALL D

      PROCEDURE C
S7      CALL E
S8      CALL F

      PROCEDURE D
S9      CALL G

      PROCEDURE E
      PROCEDURE F
      PROCEDURE G
```

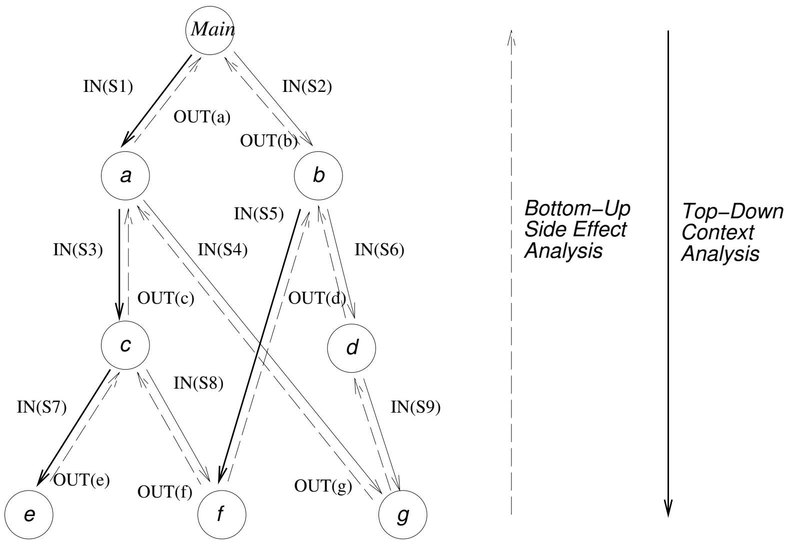

Figure 4: An example of procedure call graph and the overall structure of the two-pass interprocedural analysis.

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 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 section 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 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 (see section 3.3).

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 regions of an 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 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, we should compute the minimum number of epoch boundaries crossed for each call site, 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 too small offsets 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, 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 before entering the procedure, due to the unknown context information for the activation records that invoke 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 only propagate the summary context information 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 the 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 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 does not have a reaching definition inside the procedure and all its call sites) during side effect analysis. Note that for context analysis, we only need to update the reference marking results 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 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, 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 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 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 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.

3.3 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 the 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 the bottom-up side effect analysis, while all the actual parameters should be translated to their corresponding formal parameters during the 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 case is difficult since all the subarray information 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 the array reshaping and the mismatched COMMON block declarations occur rarely (either none or up 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 which 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 with dimensional reshaping as scalar variables.

GSA form and array data-flow analysis Since we represent the subarray information in GSA form, we need to be careful when we construct the GSA form. Usually, the 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 the GSA interprocedurally using a side effect (MAYMOD) information with additional flow analysis.

4 Experimentation

4.1 Experimentation Methodology

We use six programs from the Perfect Club benchmark suite [2] as our target benchmarks. The Perfect benchmarks are first parallelized by the Polaris compiler. In the parallelized codes, the parallelism is expressed in terms of DOALL loops. Then, we mark the Time-Read operations and insert epoch counter operations in the parallelized source codes according to the compiler algorithms in sections 2 and 3 which are implemented on the Polaris compiler. After the compiler marking, we instrument the benchmarks to generate simulation events, which include global and local memory accesses, parallel loop setup and scheduling operations, and synchronization operations.

Simulation Execution-driven simulations [12] are used to verify the compiler algorithm and to evaluate the performance of our proposed coherence scheme. All the simulations assume a 16-processor, physically distributed shared-memory multiprocessor similar to Cray T3D. Each processor has a 64-KB direct-mapped lock-up free data cache with 4-word cache line. LRU replacement policy is used for cache line replacement. Static cyclic scheduling is used for distributing loop iterations to the processors.

The memory events are generated by a source level instrumentation. Thus, we do not consider back-end compiler code generation and optimizations for our simulations. Code optimizations usually decrease the frequency of private references. This is because private references are eliminated by register allocation and other optimizations, while the need for shared data consistency prevents existing compilers from optimizing shared data [13]. However, code generation steps such as address calculation will increase the frequency of private references due to spill codes. Since these back-end compiler issues usually affect only private references, we ignore them in our simulations as we are interested in comparing different coherence schemes for shared data references.

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

1. *Invalidation-based intraprocedural algorithm* (**ALG1**) The algorithm performs stale reference detection per-procedure basis [7]. 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 intraprocedural algorithm in section 2, we use a more sophisticated algorithm. Using the MAY-MOD information used in the GSA construction, we assume that only the variables in the MAY-MOD are modified at each call site rather than all the actual parameters and global COMMON variables. 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**) It performs the full interprocedural flow analysis by propagating data-flow information across procedures (see section 3.).

Benchmarks and Static Reference Statistics Figure 5 shows the number of potential 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 only shown for the algorithms ALG2 and ALG3. The result of the array data-flow algorithms are not available due to its extensive memory usage.

In our compiler implementation, all the variables are treated as shared variables. Therefore, the number of the potential stale data references are over-estimated in the Figure 5. 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 potential stale data references substantially. On average, an additional 8.71% (array) and 20.15% (scalar) of data references are marked as potentially stale

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)
QCD2	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%)

Figure 5: The static reference counts of Time-Reads generated by different compiler algorithms. The data in parenthesis represents the result of a scalar version of each algorithm. For ALG3, the average figure shows the averaged statistics only from 3 benchmarks FLO52, MDG, and QCD2.

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 the offsets incremented by the context analysis. Note that the impact of the more precise array data-flow analysis are bigger in the interprocedural algorithms (ALG2 and ALG3) rather than intraprocedural algorithm (ALG1) since the interprocedural analysis expose more references for the optimization by the array analysis.

Dynamic Reference Statistics Figure 6 shows the dynamic reference counts of Time-Reads generated during our simulations. Note that the percentage of the Time-Reads vary significantly depending on the applications, ranging from 1.96% (QCD2) to 39.0% (SPEC77). With ALG1, an average of 3.34% of the memory references are marked as Time-Reads. Note that this number is misleading and only represents the reference count per-procedure basis since ALG1 uses invalidations at procedural boundaries. By considering the interprocedural side effects and contexts, ALG2 increases the percentage of the Time-Reads to 14.6%. With the full interprocedural analysis, ALG3 decreases it to 6.7%. The numbers shown in the parenthesis represents the percentage of the Time-Reads generated by the corresponding scalar implementation. On average, the 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.

Program	Size (lines)	% Read	Shared (total refs)	% Timeread		
				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%)
QCD2	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%)

Figure 6: Dynamic memory reference statistics for Perfect benchmarks. The data in parenthesis shows the result of scalar algorithms.

4.2 Miss Rates

Figure 7 shows how the different compiler algorithms affect the miss rates of both the compiler scheme (*CP*) and the hardware scheme (*TPI*) against the underlying BASE architecture.

As can be seen in the figure, ALG2 eliminates significant number of cache misses for both *TPI* and the *CP* compared to ALG1. This is mainly due to the result of avoiding cache invalidations at procedure boundaries. In particular, the improvement of the cache utilization is largest in the benchmarks MDG and QCD2, which suffered most (see Figure 2 in section 1.2) from the cold-start effects of the cache invalidations used in ALG1. In these benchmarks, there are several small procedures in the critical path of the execution, and the invalidations not only invalidate shared data but also sweep out private data from the cache unnecessarily. For MDG, the elimination of cache misses by ALG2 accounts for 17.7% and 24.8% for the *CP* scheme and the *TPI* scheme respectively. With the full interprocedural analysis (ALG3), an additional 10.8% and 5.5% of cache misses are eliminated compared to ALG2. Similar to MDG, ALG2 eliminates 21.9% and 24.0% of unnecessary cache misses of the *CP* and the *TPI* scheme in QCD2. The full interprocedural algorithm eliminates an additional cache misses up to 4.0% for the *CP* scheme in QCD2. This demonstrates that to achieve a good cache performance in the compiler-directed schemes, the elimination of cache invalidations is critical for the compiler marking strategy.

A similar but rather modest trend is observed in other benchmarks. In FLO52, ALG2 achieves a modest improvement against ALG1 for both the *CP* and the *TPI* scheme, eliminating cache misses by 2.6% and 4.1% respectively. Also, the full interprocedural algorithm achieves the elimination of additional cache misses by 1.9% for the *CP* scheme. In OCEAN, most of the misses are dominated by non-sharing misses and both the compiler scheme and the two-phase

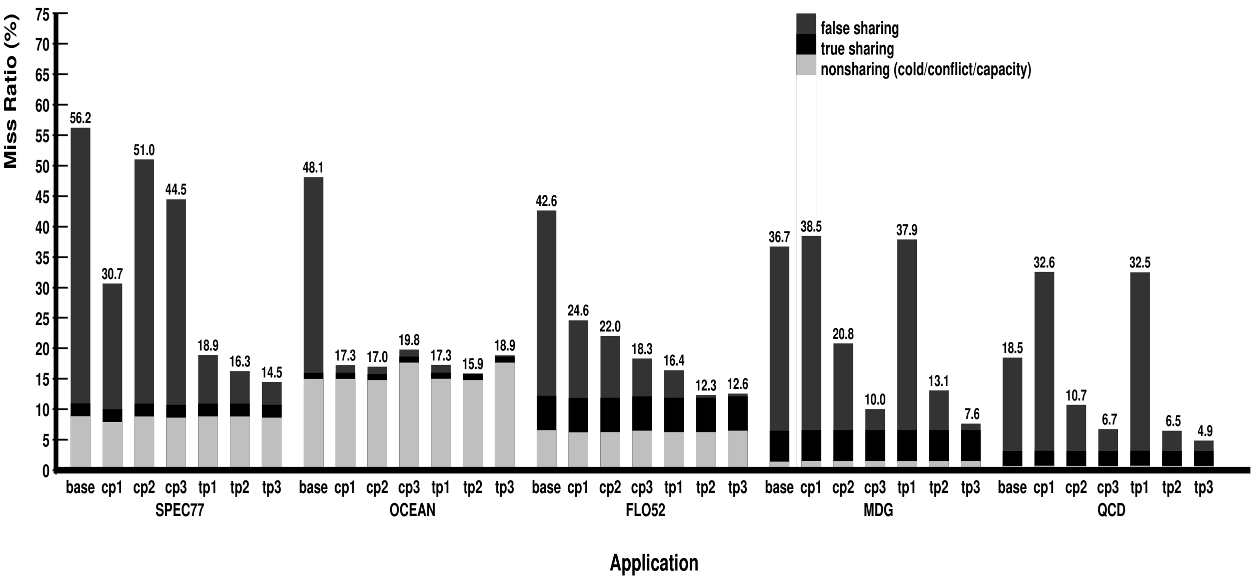

Figure 7: Miss rates using different compiler algorithms for the CP and the TPI scheme compared to an underlying BASE architecture. CP1, CP2, and CP3 represents the compiler scheme using the algorithm ALG1, ALG2 and ALG3 respectively while TP1, TP2 and TP3 shows the two-phase invalidation scheme using ALG1, ALG2 and ALG3 respectively.

invalidation scheme achieves comparable miss rates regardless of the compiler algorithms used.

In SPEC77, an interesting situation is observed. The simple interprocedural analysis (ALG2) increases the cache misses substantially by 20.3% for the *CP* scheme 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 by avoiding cache invalidations. Note that this is only true for the compiler scheme. In SPEC77, the procedures are reasonably large and so the impact of the cold start effects of the cache invalidations are 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 Figure 6). This leads to a significant number of unnecessary cache misses for the compiler scheme. For the *TPI* scheme, the conservative marking can be overcome by keeping track of cache states. In other words, more number of Time-Reads marked by ALG2 turns out to be still cache hits since *TPI* can determine the staleness of cached data at runtime more precisely using the timetag information.

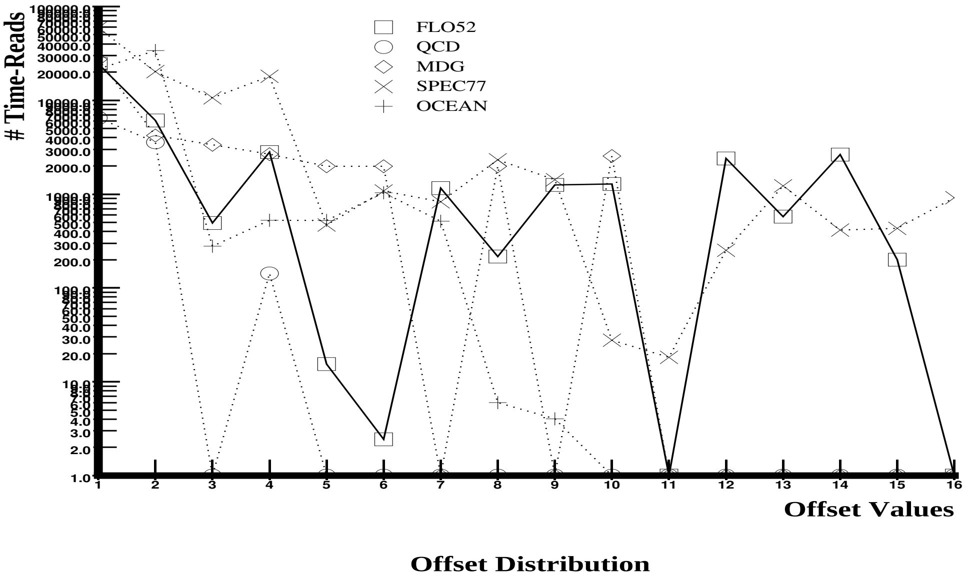

Figure 8: Distribution of offsets for Time-Read. The y axis is shown in log scale to show the distribution of large offsets more clearly. The bold line shows the averaged offset distribution of all benchmarks.

4.3 Distribution of offsets

Figure 8 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 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 might be sufficient to capture most of the inter-epoch locality.

Figures 9 and 10 show the distribution of offsets for Time-Reads in the benchmarks SPEC77 and FLO52 using three different compiler algorithms. By using the interprocedural algorithms (ALG2 and ALG3), more 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 to ALG2, ALG3 generates less Time-Reads for small offsets and more Time-Reads for large offsets. Interestingly, ALG2 sometimes generates more Time-Reads for large offsets, but this is due to conservative marking strategy. These additional Time-Reads are unnecessary and eliminated by more precise interprocedural analysis in ALG3.

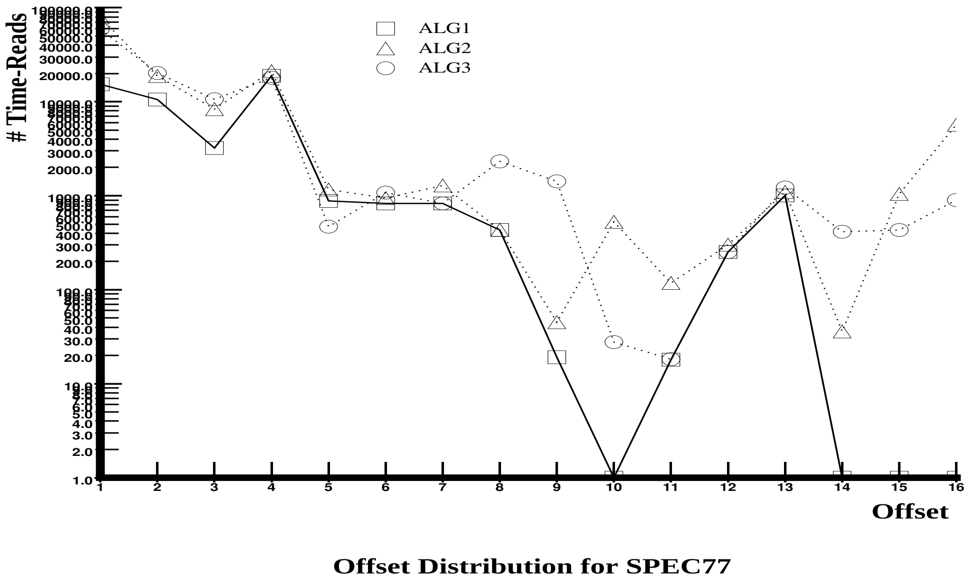

Figure 9: Distribution of offsets for Time-Read in SPEC77 using three compiler algorithms.

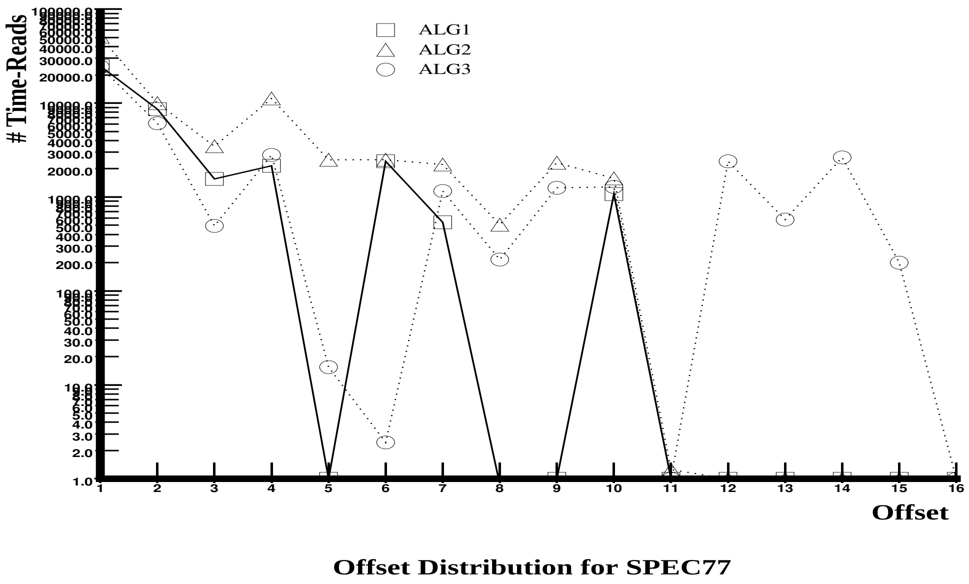

Figure 10: Distribution of offsets for Time-Read in FLO52 using three compiler algorithms.

5 Conclusion

We proposed 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 limit existing compiler-directed cache coherence schemes [3, 6, 4, 8, 10] to exploit locality only within procedure boundaries. Previous algorithms use cache invalidation [5, 7] or selective inlining [7] 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 it is often prohibitive due to potential code size expansion as well as the compile time increase.

By considering both side effects and hidden contexts at procedure boundaries, we propose an improved compiler algorithm which avoids cache invalidations 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 proposed an interprocedural algorithm which can fully 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. 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, and thus minimize the compilation time.

We have implemented these algorithms on the Polaris parallelizing compiler [11], and demonstrated the performance driven by the new compiler algorithms by running execution-driven simulations of five Perfect benchmarks. The results show that by avoiding cache invalidations, the intraprocedural algorithm eliminates up to 26.0% of the cache misses for a compiler-directed scheme compared to an existing invalidation-based algorithm [7]. With the full interprocedural analysis, up to 10.8% of additional cache misses can be removed.

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] M. Berry and others. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall, 1989.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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, November 1994.

[9] Paul Havlak. Interprocedural Symbolic Analysis. Technical report, Rice University, Dept. of Computer Science, May 1994. Ph.D. Thesis.

[10] 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.

[11] 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.

[12] 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, November 1993.

[13] J. Torrellas, M. S. Lam, and J. L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, C-43 No.6:651–663, June 1994.

[14] 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.

A Appendix

A.1 Intraprocedural algorithm

A.2 Interprocedural algorithm

Algorithm A.1 *Intraprocedural Stale Reference Detection*

```
Input:      Epoch flow graph G and GEN(S) and TARGET(S) for each statement S
Output:     STALE(S) for each statement S
Begin
  for each statement S do
     IN(S) = φ;
     OUT(S) = GEN(S)_0;
     /* assume all the actuals and globals may be modified at each call site */
     if S is CALL statement
       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 the globals and formals modified in 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
               increment the offsets 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 u1 of variable v in TARGET(S) do
             for each definition d1 of variable v with its offset > 0 in IN(S) do
                if subarray(u) ≠ φ then
                  STALE(S) = STALE(S) ∪ {u} where
                  offset(d) = minimum among all the definition d1 of variable v in IN(S) and
                  subarray(u) = subarray(u1) ∩ subarray(d1);
                              if u1 is an array reference
                else if u is FORMAL or GLOBAL
                /* assume conservatively all the formals and globals have been modified before entering the procedure */
                     STALE(S) = STALE(S) ∪ {u1} where
                     offset(d) = min_offset;
                end if
             end for
          end for
          OUT(S) = OUT(S) ∪ IN(S);
       end for
  end while
End
```

Figure 11: 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.2: *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 TimeRead issued due to hidden context;
        for each return node R in P do
          OUT(P) = OUT(P) ∪ OUT(R);
          for each formal parameter or global COMMON variable in P do
            take the minimum offset among the definitions for the same variable;
          end for
        end for
      end for
      eliminate the subroutine local variables in OUT(P);
    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 IN(Q) from Q's context to P's context;
          IN(P) = IN(P) ∪ IN(Q);
          take the minimum offset among the definitions for the same variable;
        end for /* inner for */
        perform the intraprocedural algorithm in Figure 9 with the following modification;
        for each CALL statement C in P to Q do
          context(C,Q) = IN(C) by eliminate the subroutine locals of P from IN(C);
        end for
        for each TimeRead marked in the first pass do
          if there is a definition d that covers
            add offset(d) to the offset of TimeRead;
          else if there is no overlapping definition d
              replace TimeRead with regular Read;
          endif
        end for
      end for
    end for
    decrement the level by 1;
  end for
End
```

Figure 12: 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.