Techniques for Debugging Parallel Programs with Flowback Analysis

Jong-Deok Choi
jdchoi@ibm.com

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Barton P. Miller
bart@cs.wisc.edu

Robert H. B. Netzer
netzer@cs.wisc.edu

Computer Sciences Department
University of Wisconsin–Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Abstract

Flowback analysis is a powerful technique for debugging programs. It allows the programmer to examine dynamic dependences in a program's execution history without having to re-execute the program. The goal is to present to the programmer a graphical view of the dynamic program dependences. We are building a system, called PPD, that performs *flowback analysis* while keeping the execution time overhead low. We also extend the semantics of flowback analysis to parallel programs. This paper describes details of the graphs and algorithms needed to implement efficient flowback analysis for parallel programs.

Execution time overhead is kept low by recording only a small amount of trace during a program's execution. We use semantic analysis and a technique called *incremental tracing* to keep the time and space overhead low. As part of the semantic analysis, PPD uses a static program dependence graph structure that reduces the amount of work done at compile time and takes advantage of the dynamic information produced during execution time.

Parallel programs have been accommodated in two ways. First, the flowback dependences can span process boundaries; i.e., the most recent modification to a variable might be traced to a different process than the one that contains the current reference. The static and dynamic program dependence graphs of the individual processes are tied together with synchronization and data dependence information to form complete graphs that represent the entire program. Second, our algorithms will detect potential data race conditions in the access to shared variables. The programmer can be directed to the cause of the race condition.

PPD is currently being implemented for the C programming language on a Sequent Symmetry shared-memory multiprocessor.

Index Items – debugging, parallel program, flowback analysis, incremental tracing, semantic analysis, program dependence graph.

Research supported in part by National Science Foundation grants CCR-8703373 and CCR-8815928, Office of Naval Research Contract N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.

1. INTRODUCTION

Debugging is a major step in developing a program since it is rare that a program initially behaves the way the programmer intends. While most programmers have experience debugging sequential programs and have developed satisfactory debugging strategies, debugging parallel programs has proven more difficult. The *Parallel Program Debugger (PPD)* [31] is a debugging system for parallel programs running on shared-memory multiprocessors (hereafter, called ‘‘multiprocessors’’). PPD efficiently implements a technique called *flowback analysis* [8], which provides information on the data and control flow between events in a program’s execution. PPD provides this information while keeping both the execution-time and debug-time overhead low. By using a method called *incremental tracing*, only a small amount of trace is generated during execution and is supplemented during debugging by detailed information obtained by re-executing only selected parts of the program. PPD is also capable of performing flowback analysis on parallel programs and detecting data races in the interactions between processes. This paper describes the mechanisms used by PPD to efficiently implement flowback analysis for parallel programs. These mechanisms include program dependence graphs and semantic analysis techniques such as interprocedural analysis [4, 13] and data-flow analysis [23].

The goal of PPD is to aid debugging by displaying dynamic program dependences. These dependences should guide the programmer from manifestations of erroneous program behavior (the *failure*) to the corresponding erroneous program state (the *error*) to the cause of the problem (the *bug*). Debugging is a difficult job because the programmer has little guidance in locating bugs. To locate a bug that caused an error, the programmer must reason about the causal relationships between events in the program’s execution. There is usually an interval between when a bug first affects the program behavior and when the programmer notices an error caused by the bug. This interval makes it difficult to precisely locate the bug. The usual method for locating a bug is to execute the program repeatedly, each time placing breakpoints closer to the location of the bug. An easier way to locate a bug is to track the events backward from the error to the point at which the bug caused the error. Flowback analysis tracks events in such a way. The programmer sees, either forward or backward, how information flowed through the program to produce events of interest. Using flowback analysis, the programmer can more easily locate the bugs that led to the observed errors.

Parallel programming offers challenges beyond sequential programming that complicate the problem of debugging. First, it is difficult to order events occurring in parallel programs. The ordering of the events during program execution is crucial for seeing causal relationships between the events (and therefore, the cause of errors). Second, parallel programs are often non-deterministic. Such non-determinism often makes it difficult to re-execute the program for debugging purposes. Third, interactions between cooperating processes in a multiprocessor system are frequent, and these accesses to shared variables can occur without the proper synchronization. PPD not only performs efficient flowback analysis for sequential programs, but also helps address the problems of debugging parallel programs.

In this paper, we address the class of parallel programs that use explicit synchronization primitives (such as semaphores, monitors, or Ada rendezvous) and explicit (and dynamic) process creation. While we are not addressing automatic parallelism, many of our techniques might be extended to such systems. Our current algorithms assume that the underlying machine architecture has a sequentially consistent memory system [29] (as is the case on the Sequent Symmetry). The techniques in this paper are described in terms of the C programming language [24], but they should generalize to other imperative languages. We address a large part of the C language, including primitives for synchronization. We discuss a simple approach to pointer variables but this is a topic that needs further investigation.

This paper is organized as follows. Section 2 presents an overview of the design of PPD. Sections 3 and 4 describe the graph structures and tools used by PPD to perform flowback analysis. Section 3 describes the *static program dependence graph*, built at compile time, which shows potential dependences between events in the program's execution. Section 4 describes the *dynamic program dependence graph*, built at debug time, which shows the actual dependences between events in the execution. Section 4 also describes how dynamic graphs are built by augmenting the static graphs with traces generated during execution and debugging. Section 5 presents the details of incremental tracing. Section 6 describes how flowback analysis is extended to parallel programs and how data races are detected. Section 7 presents some initial performance overhead results.

2. STRUCTURAL AND FUNCTIONAL OVERVIEW

Flowback analysis would be straightforward if we were to trace every event during the execution of a program. However, doing so is expensive in time and space. The user needs traces for only those events that may have led to the detected error. The problem is that there is no way to know what errors will be detected before the execution of the program; either the user has to generate a trace of every event so that the traces will not lack anything important when an error is detected, or the user has to re-execute a modified program that generates the necessary traces after an error is detected. Tracing every event is expensive because of unacceptable overhead, and most often impractical for parallel programs because of the distortions that the debugger would introduce in the interaction pattern between processes. Re-execution is impractical for programs that lack reproducibility, as is often the case with parallel programs.

We use *incremental tracing* to reduce the above difficulties. The main idea of incremental tracing is to generate coarse-grained traces, called the *log*, during program execution. Then, during the interactive portion of the debugging session, we use the coarse traces and other compiler-generated information to incrementally produce the fine-grained traces needed to do flowback analysis. This method transfers execution time costs into compile time and debug time. At compile time, we use semantic analyses, such as interprocedural analysis and data flow analysis, to help reduce the amount of information that needs to be generated during program execution. At debug time, we amortize the cost of generating the fine traces over the interactive debugging session. The traces are generated as the programmer asks about dependences in the program.

We divide debugging into three phases: *preparatory* phase, *execution* phase, and *debugging* phase. There are two major components in our debugging system: the *Compiler/Linker* and the *PPD Controller*. During the preparatory phase, the Compiler/Linker produces the object code and the files to be used in the debugging phase. While the object code is running in the execution phase, it generates a log to be used in the following debugging phase. When the program halts, due to either an error or user intervention, the debugging phase begins. The PPD Controller oversees the debugging phase, responding to the programmer's requests.

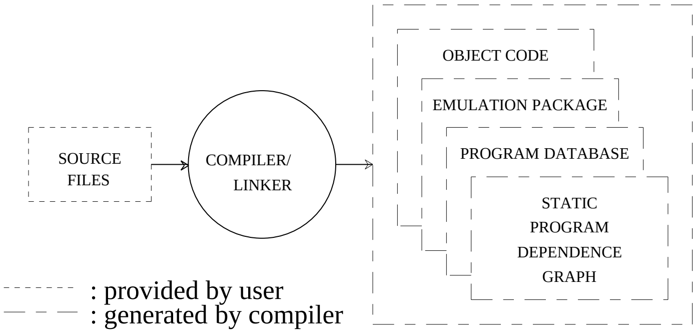


Figure 2.1. The Preparatory Phase

2.1. Preparatory Phase

Figure 2.1 shows the preparatory phase, during which the Compiler/Linker produces, along with the *object code*, the following:

1) the *emulation package* that will generate fine traces during the debugging phase to fill the gap between the information contained in the log generated during execution phase and the information needed to do flowback analysis;

2) the *static program dependence graph* that shows the static (possible) data and control dependences among components of the program; and

3) the *program database* that contains information on the program text such as the places where a variable is defined or used.

2.2. Execution Phase

The object code plays the major role in the execution phase. Figure 2.2 shows the execution phase, during which the object code generates the normal program output and a log that contains dynamic information about program execution. The log is used, along with the emulation package, during the debugging phase to generate fine traces for the flowback analysis. The log entries include *prelogs*, which record the values of the variables that might be read before the next logging point, and *postlogs*, which record the

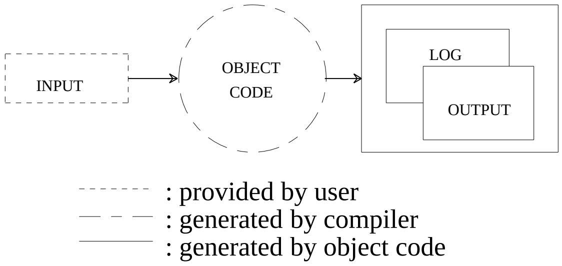


Figure 2.2. The Execution Phase

changes in the program state since the last logging point. The log entries and tracing are described in more detail in Section 5.

2.3. Debugging Phase

The goal of the debugging phase (see Figure 2.3) is to build a graph of the dynamic dependences in a program. The debugging phase assembles information from the previous phases: the static graph and program database generated by the compiler during the preparation phase, and the log generated by the object code during the execution phase. This information is used together with the emulation package to generate the detailed traces needed to build a graph of the dynamic dependences. The PPD Controller oversees the debugging phase. It responds to requests from the programmer, locating the necessary data from the log and static graph, and then executing parts of the emulation package to generate the fine traces.

3. STATIC PROGRAM DEPENDENCE GRAPH

The static program dependence graph (static graph) shows the potential dependences between program components, such as *data dependences* [26] and *branch dependences* (similar to control dependences[16]). The static graph is also the basic building block of the dynamic program dependence graph (dynamic graph).

The static graph is a variation of the program dependence graph introduced by Kuck [25]. Since then, there have been numerous variations that can be categorized into two classes according to their applications. First, the program dependence graph is used as an intermediate program representation for the

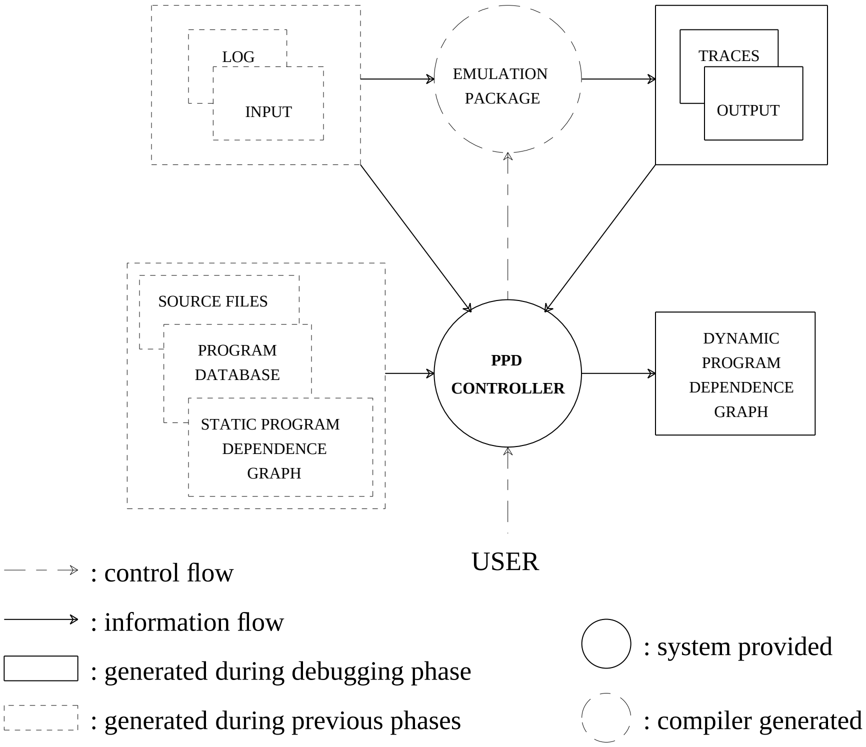


Figure 2.3. The Debugging Phase

purpose of optimizing, vectorizing, and parallelizing transformations of the program [16, 25-27, 36]. The main concern in this class is to decide whether there exist any potential dependences between two sets of statements.

Second, the program dependence graph is used to extract *slices* from a program. A slice of a program with respect to variable *v* and program point *p* is the set of all the statements that might affect the value of *v* at *p* [38]. Such slices can be used for integrating program variants [21] and for program debugging [16, 35, 37, 38]. One common attribute of the two classes of applications is that they do not use the dynamic information obtained during program execution. However, in PPD, we augment the static graph with the dynamic information obtained during execution and debugging in building the dynamic graph. The dynamic graph in PPD can be viewed as a dynamic slice of the program at an execution point based upon the actual dependences between statements.

Accordingly, the static graph structure in PPD differs in several ways from previous systems. The structure of the static graph is motivated by the following observations. First, the static graph should contain enough information to build the dynamic graph with only a small amount of trace generated at execution time. A small amount of trace means low execution-time overhead. Second, compile-time efficiency should not be compromised to identify dependences that can be easily determined with dynamic information obtained at execution and debugging times. Since the dynamic trace information effectively unrolls all loops, computing *data dependence direction vectors*[39], which are approximate compile-time characterizations of dependences, is unnecessary to show execution-time dependences. Although computing data dependence direction vectors is essential for automatic loop parallelization[3], it is unnecessary because we can reconstruct this information at execution time. Moreover, because we require the *actual* paths of control flow taken at run-time (obtained from the dynamic trace), we need not approximate such information at compile-time. We therefore do not construct a precise static control flow graph. Finally, for each subroutine, we want to identify the sets of variables that might be used or defined by the execution of that subroutine. Such identification will allow us to decide whether to show or skip the execution details of a subroutine when showing the dependences requested by the user.

In this section, we describe a static graph consisting of two layers. The outer layer, called the *branch dependence graph*, shows the branch dependences, and the inner layer, called the *data dependence graph*, shows the data dependences within the blocks of the branch dependence graph. We will discuss the two layers in detail. Interprocedural analysis is used in building the data dependence graph. With separate compilation, interprocedural analysis also allows us to avoid rebuilding the entire static graph from scratch when one or more modules of the program are modified. The separate compilation issue is described in detail in Section 3.5, where we describe how we use interprocedural analysis in building the static graphs.

3.1. Branch Dependence Graph

The outer layer of the static graph is the branch dependence graph. This (static) branch dependence graph, which is always a tree, is developed from syntactic program analysis (i.e., at parse-time). In Section 4.3, we compare this graph with the control dependence graph[16]. The static branch dependence graph consists of nodes called *control blocks* and *branch dependence edges* between these nodes. Figure 3.1 shows an example branch dependence graph. Such a graph is constructed for each subroutine in the

```
          extern int g1, g2, g3;
          extern int A[100], B[100];
          Wolf ( ) {
          int i, j, k, l, a, b;
s1:           if (g1 > 0) {
s2:               a = Add2(g1, g2);          C
              } else {
s3:               g1 = 0;
s4:       F       SubY();
s5:               a = g1;
          L1:                                D
s6:   A           a = (A[i] = b);
s7:       G       b = A[j];
s8:               A[k] = B[l] + a + b;
          L2:
s9:               a = g1;
s10:              g2 = a;
s11:      H       g1 = g2;
s12:              SubX(a);
s13:              g2 = g2 + g3;
              };
s14:          while (a > 1) {
s15:              a = a - 1;                 E
s16:  B           g2 = g2 * a;
              };
          }; /* Wolf */
```

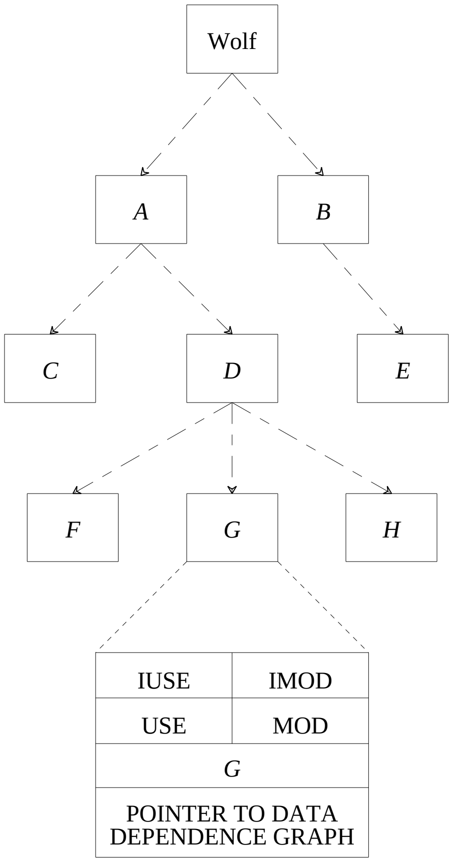


Figure 3.1. A Sample Static Graph

program. A control block is identical to a basic block, except that labels (which are potential targets of branching statements such as **goto**) always delimit the start of a new control block. For example, to handle **switch** statements in C, we also treat a **case** statement as a label, since an implicit branch occurs when a **case** does not end with a **break** and is allowed to fall through to the following **case**. A leaf control block represents a block of statements in which the flow of control always enters at the beginning and exits at the end, and that is devoid of conditional or loop control statements. For programs without labels that are potential targets of branching statements such as **goto**, the branch dependence graph is identical to the abstract syntax tree[2] of the program, with the basic blocks being the leaf nodes of the tree. Thus, the branch dependence graphs can be built at compile-time without control-flow analysis.

Since we do not perform control-flow analysis of the program, we simply assume at compile time that every label will be a target of at least one branching statement. This assumption sometimes results in overly fine-grained basic blocks, such as blocks F, G, and H in Figure 3.1. However, the benefit from not performing control-flow analysis easily offsets the small, additional overhead incurred by such pessimistic assumptions. Branching statements, such as **goto**, can affect the structure of dynamic graphs. In Section 4.3, we describe how the simple structure of the branch dependence graphs, combined with run-time traces, can handle these branching statements.

There are four non-leaf block types needed for C programs. The first type represents conditional statements, such as **if** or **switch** statements. In the absence of **goto**s (including implicit **goto**s, which occur when one **case** of a **switch** statement falls through to the following **case**), only one child of a conditional node in the static graph will execute. Block A in Figure 3.1 is of this type. During execution, either block C or block D will be executed. The second non-leaf block type represents loop control statements such as **while** or **for**. Execution of the descendent blocks may be repeated zero or more times depending upon the loop control statement. Block B in Figure 3.1 is of this second type.

The third and fourth non-leaf block types do not correspond to any statement. The third type acts as *summarizing* block for its descendent blocks and is used when its descendents constitute an *e-block*; an e-block is the unit of incremental tracing during debugging (described in Section 5.1). All the descendants of a summarizing block execute in left-to-right order. Also, the root block of a static graph is a summarizing block, even if we do not construct an e-block out of the subroutine.

The fourth type of non-leaf block is a dummy block. This block exists only as a descendent of a conditional block to group together the blocks (if there are more than one) dependent on the conditional. The dummy block satisfies the condition that only one of the descendents of a conditional block will be executed. All the descendents of a dummy block will also be executed in left-to-right order. Control block D in Figure 3.1 is a dummy block with three descendents. Leaf blocks G and H are defined because of labels ‘‘`L1`’’ and ‘‘`L2`’’; flow of control can potentially enter at these points. (We introduced these labels to show how labels affect the static graph, although there is no **goto** statement in the example program.)

Associated with each control block (except dummy blocks) are four sets of variables — the *IUSE*, *IMOD*, *USE*, and *MOD* sets — and a data dependence graph. The IUSE set is the set of variables that might be referenced before they are defined by a statement in this block; it is the set of upwards-exposed used variables [2] of this block. The IMOD set is the set of variables whose values might be defined by statements in this block. The USE set is the set of variables that might be used before they are defined in this block or any block in a subroutine called from this block (following the transitive closure of calls). The MOD set is similarly defined. While the IUSE and IMOD sets are determined locally by inspecting the statements belonging to a block, the USE and MOD sets can only be determined by interprocedural analysis.† The USE and MOD sets are described in more detail in Section 3.5 on interprocedural analysis.

The branch dependence graph for a subroutine can have several summarizing blocks, one for each e-block in the subroutine. The four sets (the IUSE, IMOD, USE, and MOD sets) for a summarizing block are the unions of the same sets of all the descendents' blocks that constitute the e-block. However they do not contain variables that cannot be accessed outside the corresponding e-block, except for upwards-exposed static variables. For example, those four sets for a subroutine do not contain variables local to the subroutine, although static variables are treated the same way as global variables. The program database [31] contains the scope information of each variable, telling whether a given variable is a global variable, a variable local to a subroutine, a static variable (in C), or a formal parameter of a subroutine. It also tells whether a given global variable of a parallel program is a shared variable. (Sequent C has two additional key words to support parallel programming[1]: **shared** and **private**.) The variables in the IUSE and IMOD sets of the summarizing block are the variables that will be written to the log (described in Section 5) at execution time.

The structure of the branch dependence graph and the four sets of used and defined variables allow for easy identification of the sets of variables that might be used and defined during the execution of an e-block. They also allow for easy identification of which e-blocks might use or modify a given variable. Section 5 discusses how these data structures work together with the log and incremental tracing.

† In previous papers[12, 31], we used different terminology for these sets as follows: IMOD was previously referred to as DEFINED, IUSE as USED, MOD as GDEFINED, and USE as GUSED. We now use terminology from Banning[9].

3.2. Data Dependence Graph

Each control block (except for summarizing and dummy blocks) has a data dependence graph that shows only the dependences between statements belonging to that block. Data dependences between different blocks are resolved at debug-time and appear in the dynamic graph. Figure 3.2 shows a sample control block and its data dependence graph. The (static) data dependence graph has two node types: *singular* and *sub-graph nodes*. The singular node represents an assignment statement, a control predicate in a statement such as an **if** or **switch**, or branch statement such as **goto** or **exit**. For a constant used on the right-hand side of a statement, we create a *constant node*, which is a sub-type of the singular node. The sub-graph node represents the call site of a subroutine and is a way of encapsulating the inside details of such subroutines. There is one static graph for each subroutine. Each node of the data dependence graph is labeled with the statement number and either an identifier or an expression.

The data dependence graph has three edge types: *data dependence*, *flow*, and *linking edges*. The data dependence edge represents a *true dependence* [5, 26]. (A statement S_2 has a true dependence upon another statement S_1, if S_2 uses output of S_1.) A flow edge from n_i to n_j is defined when the event

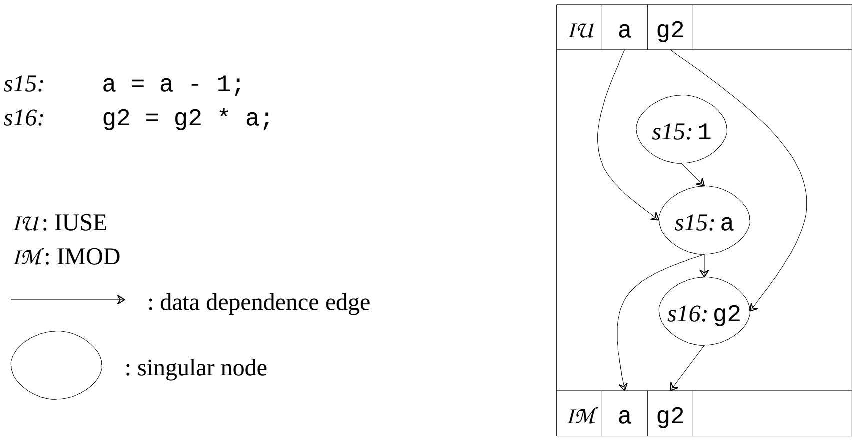


Figure 3.2. Basic Block and Its Data Dependence Graph (Control Block E in Figure 3.1)

represented by n_j immediately follows the event represented by n_i during execution; it shows the control flow of the program. The linking edge helps resolve the dependences that can only be determined at execution time, for example, deciding which array element is actually accessed when the array index is a variable. Linking edges are described in more detail in Section 3.4.

The top of the control block shows the variables in the IUSE set of the block and the bottom of the block shows the variables in the IMOD set of the block. The IUSE set of a block is the set of upwards-exposed used variables of the block. A data dependence edge from the IUSE entry for a variable into a node *N* shows a *dangling* data dependence in this block — meaning that the value of the variable has not been defined in this block before the statement represented by node *N*. A data dependence edge into the IMOD entry for a variable shows the last statement in the block that modifies the variable. All the nodes in a data dependence graph are totally ordered according to the corresponding statements in the control block, because statements in a control block are sequential. This total ordering shows the execution order of events represented by the nodes, and is represented by the flow edges connecting the nodes, so we can say that a node is after or before another node in a control block. Ordering events belonging to different

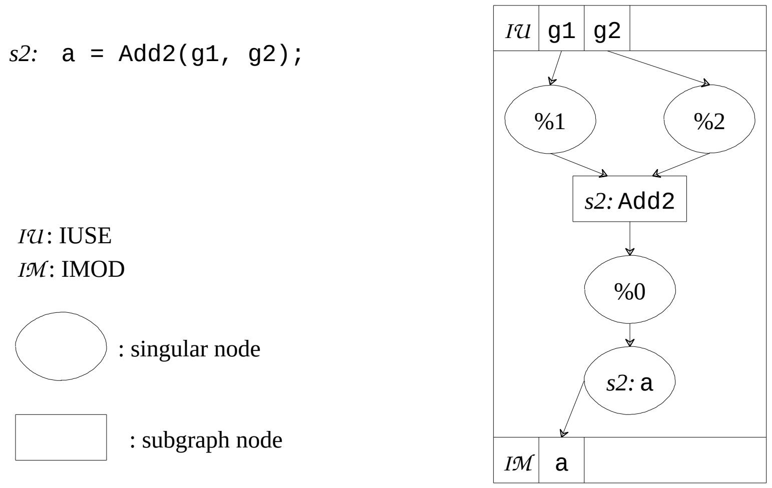


Figure 3.3. Data Dependence Graphs for Parameter Mapping (Control Block C in Figure 3.1)

processes is important in debugging parallel programs, which is described in Section 6. We will not explicitly show the flow edges in the figures in this section.

Inter-block dependences (dependences between two statements belonging to different control blocks) are not resolved at compile time; they are not recorded in the static graph. Inter-block dependences are resolved during debugging and are recorded in the dynamic graph (described in Section 4).

3.3. Parameters to Subroutines

To map between formal parameters and actual parameters of a subroutine call during debugging, we create a *parameter* node (a variant of the singular node) for each actual parameter passed to a subroutine. Each parameter node is labeled with ‘‘%’’ followed by the parameter position (%0 represents a function return value). Figure 3.3 shows the static graph of control block C in Figure 3.1. and shows how actual parameters are mapped to the formal parameters of a called subroutine.

3.4. Arrays and Linking Edges

Array index values are usually unknown at compile time, so it is not possible to identify the array elements that will actually be accessed. Our approach is to supply enough information in the static graph so that array reference dependences can be quickly determined at debug time. We use a new edge type, the *linking edge*, and two variants of the singular node, the *index* and *select* nodes. Index nodes show the indices used in array accesses, and select nodes represent read-accesses of an array. Linking edges represent *potential* data dependences, and are used during debugging to quickly locate the actual dependences.

To represent an assignment to an array element, a singular node is created. Nodes ‘‘*s6:* A’’ and ‘‘*s8:* A’’ in Figure 3.4 are examples of such nodes. As with assignments to scalar variables, this node contains data dependence edges from the nodes representing the variables used in the right-hand side of the assignment. However, for array assignments, a linking edge is then added, from the most recent node in the control block that writes the same array, to the assignment node. If there are no previous writes to the same array in the control block, then a special IUSE set entry is made for the array and a linking edge is added from this entry. Finally, an index node is created for each array index and is labeled with ‘‘%’’ followed by the index position (similar to a parameter node). A data dependence edge is added from each

```
s6:   a = (A[i] = b);
s7:   b = A[j];
s8:   A[k] = B[l] + a + b;
```

IU: IUSE
IM: IMOD

: data dependence edge
: linking edge

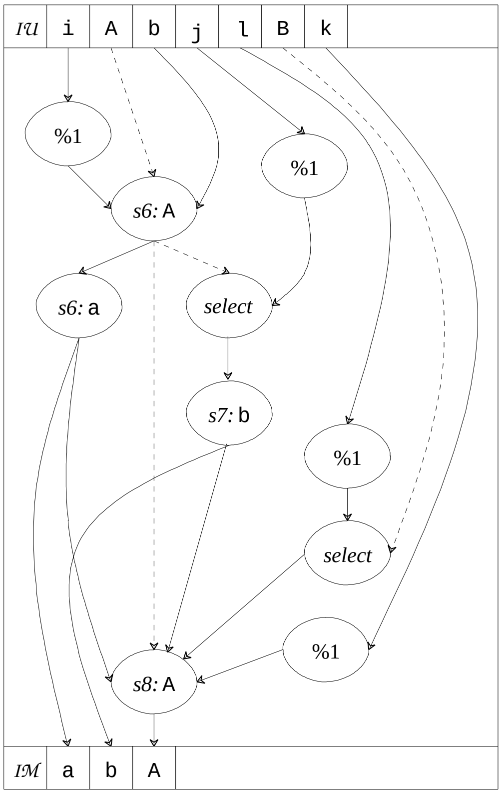


Figure 3.4. Data Dependence Graph with Array and Linking Edges (Control Block G in Figure 3.1)

index node to the assignment node. For example, node ‘‘*s6:* A’’ in Figure 3.4 contains three incoming edges: one data dependence edge for the index value, one data dependence edge for the variable used in the right-hand side of the assignment, and a linking edge from the IUSE set entry for the array being modified (since there were no previous modifications of array ‘‘A’’ in the control block).

Because a definition of an array element is a *preserving* definition[2], which fails to prevent any uses reached by the definition from being upwards-exposed, a use of an array element always creates an entry in the IUSE set of the control block. We also insert an entry if the first reference to the array in the control block is a definition of an element, in anticipation of a subsequent use of the array.

A read from an array element is handled identically except that a select node is created to represent the read. For example, the select node above node ‘‘*s7:* `B`’’ in Figure 3.4 represents the array access ‘‘`A[j]`’’ on the right-hand side of statement *s7*. This select node has an incoming data dependence edge from the index node and an incoming linking edge from node ‘‘*s6:* `A`’’, the most recent modification of array ‘‘`A`’’ in the control block. The above mechanisms are similar to the ideas used for array related dependences in [35].

The actual data dependences for each array read are determined during debugging and are reflected in the dynamic graph. Once the fine traces for the e-block containing an array read are generated, the index values of all array accesses in that e-block will be known. The linking edges are followed backwards, from the select node, until an assignment to the same array location is found. A data dependence edge can then be added in the dynamic graph from this assignment to the select node. If no such assignment is found (the IUSE set entry for the array was reached), then a dangling dependence exists for the array read. The dangling dependence can then be resolved as described in Section 5.

3.5. Interprocedural Analysis and Data Dependence Graph

USE and MOD sets, computed by interprocedural analysis, allow us to identify more precise (potential) dependence information than the worst-case assumption that every global variable in the program is possibly used and defined by each call to a subroutine. In this section, we describe the use of the USE and MOD sets.

Building the data dependence graphs with interprocedural analysis is done in two steps. The first step is done at compile time without interprocedural information, building the *pre-graph* form of the data dependence graphs. The graphs in Figures 3.2–3.4 are all pre-graphs. The second step is done at link time, producing the *post-graphs* by modifying (if necessary) the pre-graphs with interprocedural summary information. When several modules of a program are re-compiled with separate compilation, we need to rebuild only the pre-graphs of the re-compiled modules. Only those post-graphs that contain calls to subroutines whose USE or MOD set has changed need to be built again. Figure 3.5 shows the pre-graph and the post-graph for control block H in Figure 3.1. We outline how to build the pre-graph and the post-graph in this section. Detailed algorithms for building these graphs appear in [11].

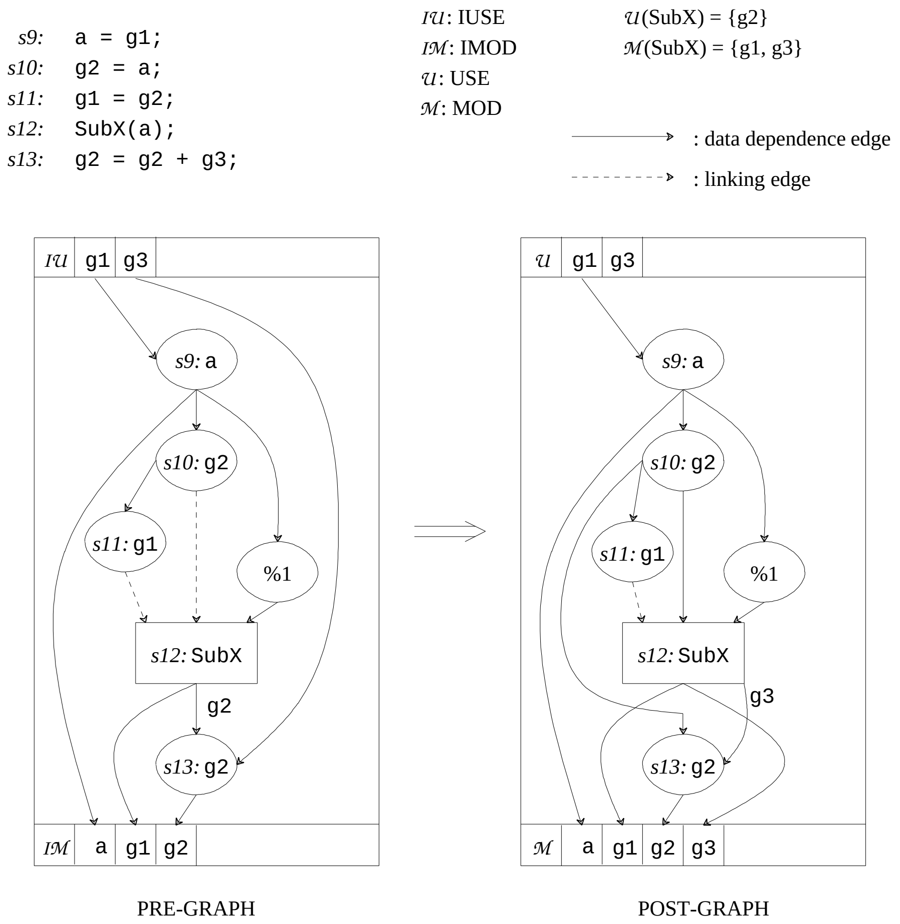


Figure 3.5. Data Dependence Graph
Before and After Interprocedural Analysis
(Control Block H in Figure 3.1)

Our approach (heuristics) to include interprocedural information is as follows. When we meet a subroutine call in building the pre-graph of a control block, we assume that all the global variables written so far in the control block might be written by the subroutine. Then, we create a linking edge for each such global variable out of the most recent node that wrote the variable, and into the sub-graph node representing the subroutine call. We use the linking edges to identify the parts of the pre-graph that might need to be modified to produce the corresponding post-graph. Our approach is certainly more pessimistic than approaches that use the MOD set of a procedure, computed interprocedurally, as the basis of determining what might be modified by a subroutine call [4]. However, our approach is simple to implement and not overly pessimistic in that we do not assume all the global variables but only those that are used or defined in a control block might be modified during the execution of a subroutine called in the control block. We need more experiments with the working prototype under construction before we can evaluate the effectiveness and efficiency of this approach.

When building the post-graph, the interprocedural summary information is reflected in each sub-graph node in the following ways: First, we create a data dependence edge into the sub-graph node for each global variable that is in the USE set of the sub-graph node. Second, we create a data dependence edge out of the sub-graph node for each global variable that is in the MOD set of the sub-graph node. Finally, we create a linking edge into the sub-graph node for each global variable that is in the MOD set but is not in the USE set of the sub-graph node.

The linking edge is needed because USE and MOD are sets of variables that might be accessed during the procedure call. For example, if during debugging we discover that ‘‘`SubX`’’ (see Figure 3.5) does not actually modify ‘‘`g1`’’, we need to locate the most recent node before ‘‘`SubX`’’ that modifies (or might modify) ‘‘`g1`’’, which in this example is ‘‘*s11:* `g1`’’. The linking edge from ‘‘*s11:* `g1`’’ to ‘‘*s12:* `SubX`’’ serves this purpose. (Note that the linking edge was similarly used for arrays in the previous subsection.)

Figure 3.5 shows how the post-graph is constructed from the pre-graph and information from interprocedural analysis. First, the linking edge of ‘‘`g2`’’ into the sub-graph node in the pre-graph is changed into a data dependence edge, because ‘‘`g2`’’ is in USE(SubX). Second, the data dependence edge of ‘‘`g2`’’ out of the sub-graph node into the node ‘‘*s13:* `g2`’’ is disconnected from the sub-graph node and reconnected into the node ‘‘*s10:* `g2`’’, because ‘‘`g2`’’ is not in MOD(SubX). The reconnection is done by following the ‘‘`g2`’’ dependences through the sub-graph node. Third, there are two additional edges out of the sub-graph node: one into the MOD entry for ‘‘`g3`’’ and the other into node ‘‘*s13:* `g2`’’. These edges are added because ‘‘`g3`’’ turned out to be in MOD(SubX). Last, the data dependence edge from the IUSE entry for ‘‘`g3`’’ into node ‘‘*s13:* `g2`’’ is deleted. The linking edge out of ‘‘*s11:* `g1`’’ into the sub-graph node is intact because ‘‘`g1`’’ is in MOD(SubX) but is not in USE(SubX).

3.6. Pointers and Parameter Aliases

Pointers and aliases make the semantic analysis of the program difficult. Currently, we do not detect dependences involving pointers at compile time. Instead, we simply trace all uses of pointers in the log and establish such dependences during debugging. This approach will be viable if the dynamic frequency of pointer references is low. For example, tracing a pointer access requires approximately 20 assembly language instructions, and if one out of every ten instructions is a pointer reference [32], the tracing will slow execution by a factor of three. However, we are investigating ways to reduce the potentially large amount of execution-time traces due to pointers and dynamic objects by using a method similar to [22, 30].

Our methods can be extended to handle the special case of aliases resulting from reference parameters in languages like Pascal or FORTRAN. Our approach is to identify, at compile time, potential aliases resulting from reference parameters [9, 10]. In the static data dependence graphs, we link together (with linking edges) all nodes representing writes to variables that are potential aliases. In the prelog for a subroutine containing reference parameters that are potential aliases, the address of each such reference parameter is recorded. Then, during debugging, aliases can be detected by comparing these addresses. Parameters whose addresses are the same are aliases. In addition, a parameter whose address is identical to the address of a global variable is an alias for that variable. Once aliases are known, incremental tracing can be employed, and actual data dependences can be established in the dynamic graph (by following linking edges back, as was done for arrays).

4. DYNAMIC PROGRAM DEPENDENCE GRAPH

The *dynamic program dependence graph* (dynamic graph) is constructed during debugging to show the causal relations between events in a program's execution. This graph shows the *dynamic* data and branch dependences exhibited by the execution. In this section, we describe how the dynamic graph is constructed from the static graphs (generated at compile time) and the fine traces (generated by incremental tracing during debugging), and illustrate its construction with an example.

4.1. Dynamic Program Dependence Graph

A dynamic graph is constructed for each e-block executed during the program's execution, and shows the actual dependences that occurred among events belonging to that e-block. The dynamic graph is constructed by splicing together the data dependence graphs for each control block that was executed in the e-block. Data dependence edges are added between the graphs to show the dynamic data dependences that actually occurred, and branch dependence edges are added to show how control flow was transferred from one control block to another. In addition, *ENTRY* and *EXIT* nodes are added to show the entry and exit points of the e-block.

Singular nodes are augmented with values (when appropriate) indicating the value computed by the statement represented by the node. Sub-graph nodes, which encapsulate the execution details of subroutine call, can be expanded to uncover a nested dynamic graph showing the details of the call.

A flow edge from n_i to n_j is defined when the event represented by n_j immediately follows the event represented by n_i during execution; it shows the control flow of the program. A data dependence edge shows a *true* data dependence between two nodes.

A branch dependence edge from n_i to n_j is defined when the event represented by n_i is the most recent branch statement, such as an **if** or **goto** statement, that caused the program control to flow to n_j in a given execution instance. The branch dependence is concerned about the *actual* program control flow in an execution instance of a program, while *control dependence* in *Program Dependence Graphs (PDG)* [16] is concerned about the *potential* program control flow in a program. Details on branch dependences and their relationship to control dependences are presented in Section 4.3.

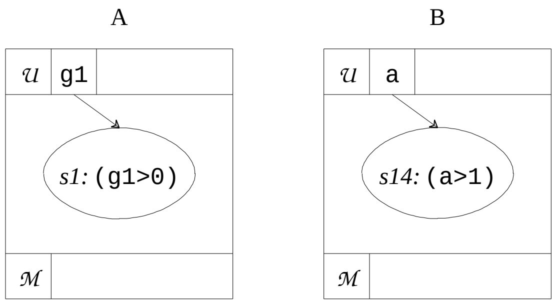
A synchronization edge shows the initiation and termination of synchronization events between processes, such as semaphore operations or sending and receiving messages. Synchronization edges are used in debugging parallel programs and will be described in more detail in Section 6.

4.2. Building the Dynamic Graph

We will use subroutine ‘‘`Wolf`’’ to illustrate how the dynamic graph is built from the static graph and fine traces. The data dependence graphs for the blocks A and B are given in Figure 4.1, and the graphs for the remaining blocks were given back in Figures 3.2–3.5. We assume that, of the choice between blocks C and D, block C is executed. We also assume, for this execution instance, the execution sequence of blocks is A, C, B, E, B, E, B — i.e., we assume the body of the **while** statement (blocks B and E) is executed twice.

```
A:
  s1:  if (g1 > 0)

B:
 s14:  while (a > 1)
```



𝒰: USE
ℳ: MOD

Figure 4.1. Data Dependence Graphs for Control Blocks A and B of Figure 3.1

Figure 4.2 shows the resulting dynamic graph of this execution. (Dotted boxes showing blocks are not part of the dynamic graph.) Notice that for simplicity, parameter nodes for simple variable parameters are replaced in the figure with labeled edges. Also, flow and synchronization edges are not shown. The graph was constructed by combining the data dependence graphs in the order that their control blocks were executed, and by inserting branch and data dependence edges between them. To insert the data dependence edges, we connect each variable in the USE set to the variable in the most recent MOD set that contains the variable. The branch dependence edges are obtained from the branch dependence graph.

The linking edges in the static graphs are the means of representing data dependences unresolved at compile/link time. The linking edges that connect nodes that write to the same array will not be included in the dynamic graphs. Also, a linking edge going into a select node for a read from an array element will be replaced with a data dependence edge coming out of the most recent node for a write to that same array element.

A linking edge coming out of a variable and going into a sub-graph node is deleted or replaced with a data dependence edge, depending on the execution of the sub-graph node. If the variable is actually written by the sub-graph node, we simply delete the linking edge. If it is not written, we delete the linking edge and make the data dependence edges of the variable that are coming out of the sub-graph node bypass the sub-graph node in the dynamic graph. These data dependence edges will be now coming from the node

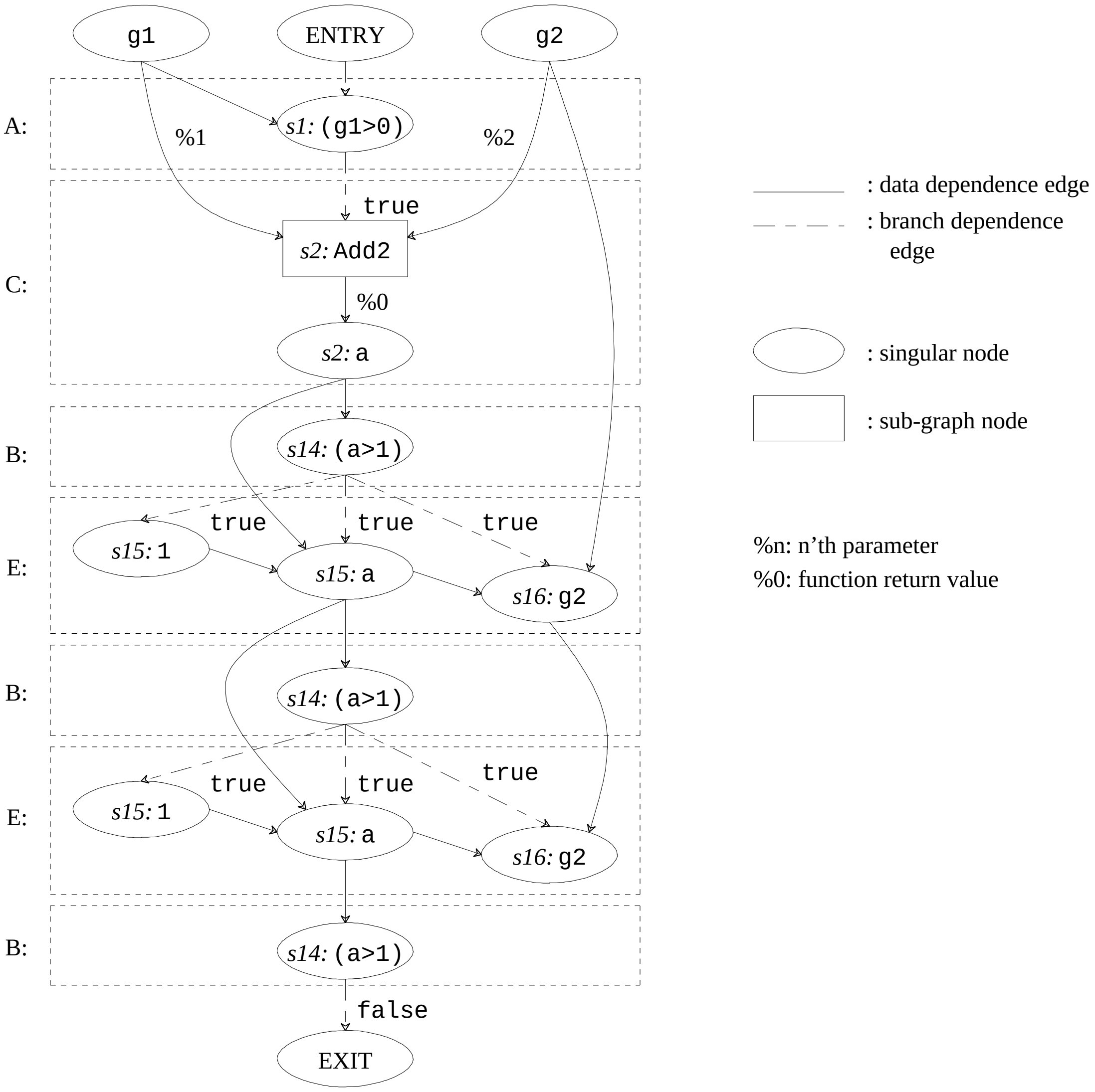


Figure 4.2. Dynamic Graph for An Instance of Subroutine ''`Wolf`''

from which the deleted linking edge originally came (note that if the variable is read in the sub-graph node before it is written, there would have never been a linking edge; it would be a data dependence edge). In addition, as more is learned as the debugging session proceeds about which variables are actually read and written inside sub-graph nodes, data dependence edges may have to be re-routed to keep the dynamic graph up-to-date. For example, if it is discovered that the execution represented by a sub-graph node did not actually modify a variable that is in its MOD set, then the data dependence edge for that variable would be re-routed around the sub-graph node.

For example in the post-graph of Figure 3.5, if the execution of ‘‘`SubX`’’ actually wrote ‘‘`g1`’’, the linking edge coming out of node ‘‘*s11:* `g1`’’ and going into the sub-graph node would be deleted in the dynamic graph. If the execution of ‘‘`SubX`’’ did not write ‘‘`g1`’’, the linking edge would be replaced with a data dependence edge that bypasses the sub-graph node and goes into the MOD entry for ‘‘`g1`’’. The data dependence edge coming out of ‘‘*s12:* `SubX`’’ and going into the MOD entry for ‘‘`g1`’’ would also be deleted in this case.

4.3. Dynamic Branch Dependence Graph

The dynamic branch dependence graph provides information about the actual control flow taken during execution. The graph contains one node for each execution instance of a control block, and *branch*

```
if (C0) {
    if (C1) {
          if (C2) {
                                        B1
                a = b;
                goto L1;
          } else {
                if (c3) {
                     b = c;             B2
                     goto L1;
                else {
                   L1:                  B3
                   c = d;
                }
                d = e;                  B4
          }
    } else {
          a = c;                        B5
    };
    d = c;                              B6
};
```

Figure 4.3. A Sample Program Segment with goto Statements

dependence edges that connect the nodes. Intuitively, each block contains an incoming branch dependence edge from the most recent branch statement (either conditional or unconditional) that caused control flow to reach the block. We will use the example program segment in Figure 4.3 to illustrate how to construct the dynamic branch dependence graph. We first give an intuitive description of how dynamic branch dependence edges are constructed given static branch dependence graphs and trace information. We then provide formal description of the mechanism. Finally, we compare branch dependences to control dependences used in the PDG by Ferrante, et al [16].

There are two cases when we add branch dependence edges, reflecting the two ways program control can flow from one basic block (*source block*) to another basic block (*target block*). First, the source block can contain a (conditional or unconditional) branch statement that transfers control to the target. In this case, the target block is reached only because of the branch statement, and a branch dependence edge is constructed from the source to the target block. Second, the source block can contain no branch statements and control passes through the source block into the target. In this case, the branch dependence edge into the target is constructed from the block containing the most recent branch statement†.

Figure 4.4A shows the static and dynamic branch dependence graphs for the program segment in Figure 4.3 in which execution sequence of blocks is C0, C1, C2, B1, B3, B4, B6. Note that B3 has *dynamic* branch dependence edge from the **goto** statement of B1, while B3 has a *static* branch dependence edge from C3. B4 also has a dynamic branch dependence edge from this **goto** statement, but a static branch dependence edge from C2. These dynamic branch dependence edges show that blocks B3 and B4 were reached because of the **goto** in B1; they were *not* reached because of the conditionals in which they are nested (C3 was bypassed altogether, and C2 evaluated to true). However, B6 has a dynamic branch dependence edge from C0, because B6 is a child of C0, which evaluated to true, causing B6 to be reached.

We now formally describe how to identify dynamic branch dependence edges with static branch dependence graphs and dynamic traces.

† This is actually a slight oversimplification. If the target is one child of a conditional node, and the most recent branch is a **goto**, then a branch dependence edge is constructed from the block containing this **goto** only if the **goto** caused control flow to either bypass the conditional or jump into the conditional from outside.

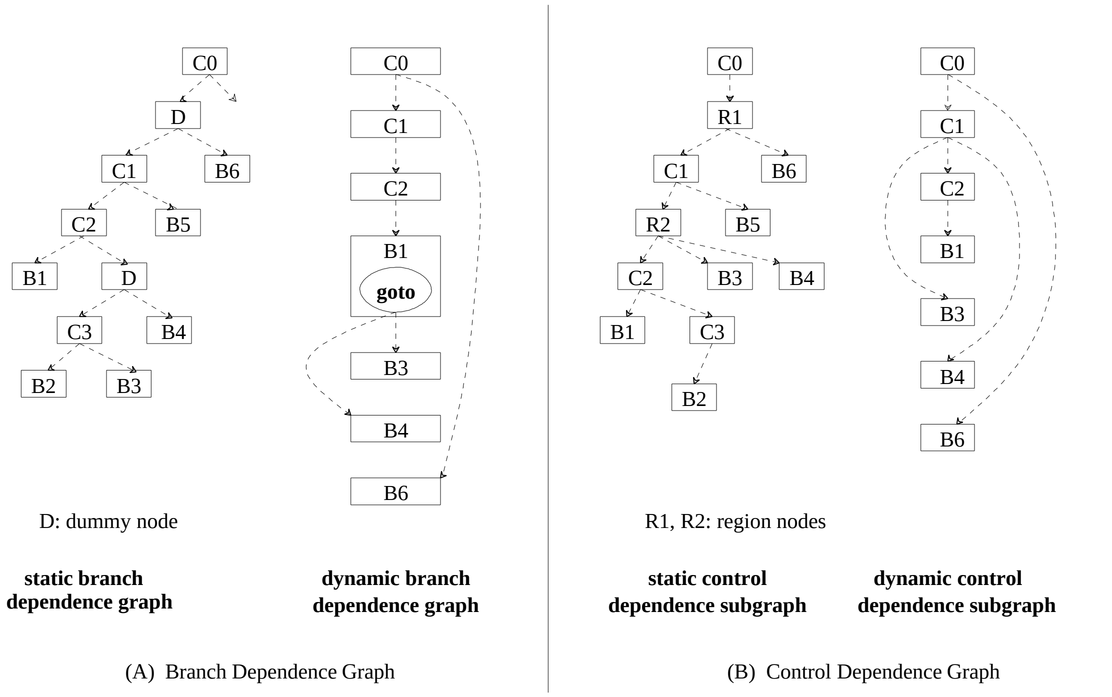


Figure 4.4. Branch and Control Dependence Graphs

Definition 4.1

The *conditional stream* of a program execution is the sequence of control block instances representing the conditional statements executed, in the order they executed. □

The instances of conditional blocks C0, C1, and C2 in Figure 4.4A belong to the conditional stream, although C3 does not, since its execution was bypassed.

Definition 4.2

For a control block instance C_i in the conditional stream, the *dynamic children* of C_i are the instances of those executed control blocks that are reachable from C_i in the static branch dependence graph, by following the edge from C_i corresponding to the branch that was actually taken, and without passing through another conditional node. □

For example, C0 has two dynamic children: C1 and B6; C1 has one dynamic child C2, which has one dynamic child B1. Now, we formally describe how to construct the dynamic branch dependences.

Definition 4.3

For each block B in the dynamic branch dependence graph, an incoming dynamic branch dependence edge is constructed from block S if

(1) S contains a (conditional or unconditional) branch for which B was the target, or

(2) B is a dynamic child of S, and S is the most recent ancestor of B in the static branch dependence graph that is a conditional node, or

(3) neither (1) nor (2) hold and S contains the unconditional branch most recently executed before B.

If none of the above conditions are true (e.g., because B is nested within no conditional statement), then no incoming edge is constructed for B. □

Recall that for programs without either explicit or implicit **goto** statements (such as a break-less **case** statement falling through to the following **case**), our notion of branch dependence is identical to *control dependence* used by Ferrante, et al [16]. However, for programs containing **goto** statements, branch dependences are different than control dependences. A major objective of flowback analysis is to show the flow of a particular execution instance and not to speculate on possible control flows in the execution. A control dependence from block S to block T means that the value of the conditional expression at S determines, in all cases, whether control flow will reach T. In contrast, static branch dependences are designed so that a dynamic branch dependence from block S to block T shows how control flow actually reached T. Figure 4.4B shows the (static) control dependence sub-graph for their PDG (of the program segment in Figure 4.3) and the corresponding (possible) dynamic control dependence sub-graph.† Note that B3 is control dependent on C1 (and not C3), meaning that the value computed by C1 determines whether B3 is reached, and B3 will be reached regardless of the value computed by C3. In contrast, B3 is branch dependent on B1, meaning that B3 was reached because of the **goto** in B1. This branch dependence shows how

† They do not actually build a dynamic graph.

control flow actually reached B3 even though C3 was bypassed. The control dependence only shows that the execution of B3 depends on C1 (and not C3); it does not show *how* C3 was bypassed in this particular execution, allowing control to reach B3. However, the dynamic control dependence sub-graph in Figure 4.4B combined with that in Figure 4.4A, might be informative to show the possible behavior of the program in some other execution instances.

5. INCREMENTAL TRACING

We use incremental tracing to reduce the execution overhead associated with flowback analysis. In incremental tracing, we divide the program into blocks, called emulation blocks (e-blocks), and generate coarse execution-time traces (logs) based on these blocks. For parallel programs, there is one log file for each process created during the execution. During the interactive portion of the debugging session, we use these traces and other compiler-generated information to incrementally produce the fine-grained traces needed to do flowback analysis. In this section, we first describe the compile time issues associated with dividing the program into e-blocks. We also describe the debugging time issues associated with how to quickly locate the coarse traces generated by a particular execution instance of an e-block. Accesses to large arrays pose a special problem in controlling execution overhead, since generating traces that contain the entire contents of an array could substantially slow a program's execution. Section 5.5 addresses this issue and presents heuristics to deal with the problem. Section 7 discusses the effectiveness of these heuristics.

5.1. Emulation Blocks and Logs

As described in Section 2, the traces generated during program execution include prelogs and postlogs. The object code generated by the compiler/linker during the preparation phase contains code to generate the prelogs and postlogs. By using semantic analysis, we divide the program into numerous segments of code called e-blocks. Each e-block starts with code to generate a prelog and ends with code to generate a postlog. The IUSE and IMOD sets of an e-block correspond to its prelog and postlog. An e-block is also the unit of incremental tracing during debugging. As will be described in more detail later in this section, a subroutine is a good example of an e-block.

The i'th prelog and the corresponding postlog generated by an e-block during program execution are called *prelog(i)* and *postlog(i)*, respectively. The time interval between a prelog and its matching postlog is called a *log interval* and is denoted as I_i for the log interval between prelog(i) and postlog(i). Programs usually contain loops, so a given e-block in a program may have several corresponding log intervals during execution. Figure 5.1 shows example log intervals.

Prelog(i) consists of the values of the variables belonging to the IUSE set (of the e-block that generated the prelog) at the beginning of I_i, and postlog(i) consists of the values of the variables belonging to the IMOD set (of the same e-block) at the end of I_i. Each log entry also carries the e-block identifier that generated the log entry. To reproduce the same program behavior for log interval I_i during the debugging phase, we use the program code for the e-block that generated prelog(i) and postlog(i), the log entries generated during I_i, and the same input as originally fed to the program during that log interval.

Log intervals nest when one subroutine calls another. For example, in Figure 5.1 we assume that log interval I_3 corresponds to the execution of a subroutine named Sub3. We also assume that I_4 corresponds to the execution of a subroutine named Sub4, that is called from within Sub3. Prelog(3) and postlog(3) are made at the start and end of I_3, respectively; prelog(4) and postlog(4) are made at the start and end of I_4, respectively. In this case, we say log interval I_4 is nested inside log interval I_3. When we need to generate fine traces at debugging time for log interval I_3, we can use postlog(4) to avoid generating fine traces for I_4; we update the program state with postlog(4) when the call to Sub4 is reached, and skip over the execution of Sub4. Details on the fine trace generation and debugging time activities are given in [31].

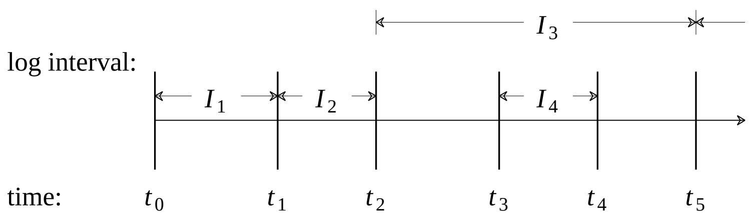


Figure 5.1. Log Intervals

5.2. Tradeoffs for Constructing E-blocks

In this section, we describe how to divide the program into e-blocks. The only condition for several consecutive lines of code to form an e-block is that there is a single entry point. Whenever control is transferred from one e-block to another, the control must be transferred to the entry point of the second e-block, where the prelog is made. The postlog is made at the exit point where the control is transferred out of an e-block. One natural candidate for constructing an e-block is the subroutine, since the entry and the exit points are well defined. (Actually, an e-block could be any node of the control dependence graph in PDG [16], since the entry and exit points of each node in PDG are well defined.)

The size of e-blocks is crucial to the performance of the system during the execution and debugging phases. In general, if we make the size of the e-blocks large in favor of the execution phase, the debugging phase performance will suffer. On the other hand, if we make the size of the e-blocks small in favor of the debugging phase, execution phase performance will suffer. While the number of logging points should be small enough so as not to introduce unacceptable performance degradation during the execution phase, it should also be large enough so as not to introduce unacceptable time delay in generating fine traces during the debugging phase. Consider, for example, the case in which the size of a subroutine is very large. Though the size of a subroutine has no direct relationship to the time needed to execute it, we can act conservatively to construct several e-blocks out of such a large subroutine.

Loop constructs, even though small in size, may require long execution time and thus introduce unacceptable time delay in generating fine traces. Currently, the PPD compiler constructs one e-block from each loop. However, the compiler constructs only one e-block from the outermost of multiply nested loops. Defining e-blocks for loops allows the debugging phase to proceed without excessive time spent in re-executing the loops. Still, if the user is interested in the execution details inside such loops, we can re-execute the e-blocks corresponding to the loops.

Three elements can affect the program behavior of an e-block: the initial state as recorded by the prelog, the code of the e-block, and input statements in the e-block. We need to accommodate input statements in an e-block to make the behavior of the e-block during debugging the same as that during execution. We can make each input statement an e-block, whose IMOD set consists of the variables affected by the input statement.

5.3. Log Optimization

Small and frequently called subroutines can be a problem. If we make an e-block out of each small subroutine, the amount of logging done during the execution phase may be large enough to introduce unacceptable performance degradation. To avoid this problem, it may be better not to make e-blocks out of subroutines that do not contain subroutine calls (i.e., subroutines that correspond to leaf nodes in the call graph). If an e-block is not formed from such a subroutine, then the subroutine itself does not perform any logging. Instead, the e-blocks that call this subroutine (its parent e-blocks in the call graph) perform its logging. However, a subroutine that either contains a loop or contains accesses to a static variable (in the C language) is not eligible for such optimization. This process can be applied recursively to the parent e-blocks, and continue any number of levels up the call graph (as specified by the user) until an e-block is reached that is ineligible for optimization.

5.4. Locating Log Intervals for Incremental Tracing

When the debugging phase starts, we generate fine debugging time traces for the last log interval — the log interval that contained the last statement executed. (The last log interval usually lacks the postlog when the execution halted due to an error or user intervention.) This allows the initial dynamic graph to be constructed. From then on, there are three cases when we need to generate fine traces for a new log interval: (1) when the user wants to know the details of the dependences of a parameter passed from a calling subroutine, (2) when the user wants to know the details of a *hidden* dependence edge — a dependence edge that either terminates into or comes out of a sub-graph node, or (3) when the user wants to know the details of a dangling dependence (a dependence for a variable that is read in an e-block before it is written).

When the user wants to know the detailed dependence of a parameter, we can easily locate the log interval needed to generate fine traces; the log intervals are nested as in Figure 5.2, and the caller's log interval is the one enclosing the current log interval. When the user wants to know the detailed dependence of a function return value, we can also easily locate the needed log interval; the callee's log interval is one of those log intervals nested in the current log interval, and log intervals at the same nesting level are generated in the execution order of the called subroutines.

When the user wants to know the details of a hidden or dangling dependence, we need to identify the log interval needed to generate fine traces to show the details. To facilitate identifying such log intervals, we obtain the IMOD set of each e-block at compile time and keep it as part of the program database [31]. We also keep in the program database, for each variable that might be accessed by more than one e-block, the list of e-blocks that contain the variable in their IMOD sets. We call the list the *e-block table*. The e-block table in Figure 5.2 shows the list of e-blocks for three variables: ‘‘g1’’, ‘‘g2’’, and ‘‘g3’’.

Figure 5.2 also shows an example log file. Log entries generated by the same e-block form a linked list; each postlog has two pointers: one pointing to its corresponding prelog, and the other pointing to the most recent postlog made by the same e-block. *E-pointers* is an array of pointers to the last log entry made by each e-block and is updated during program execution.

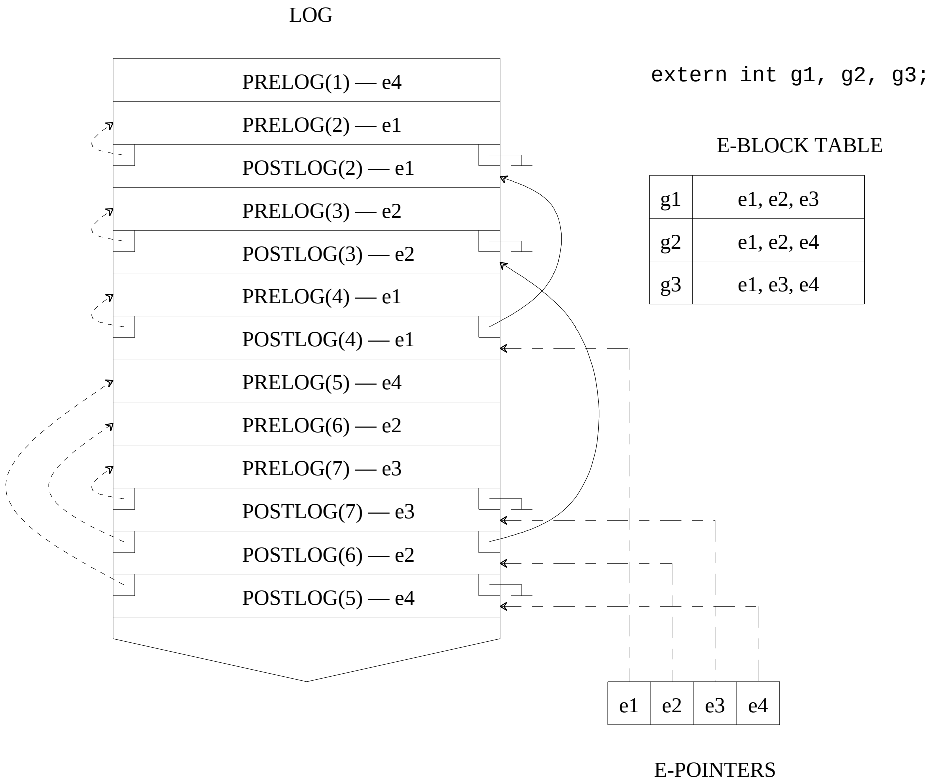


Figure 5.2. LOG with back pointer for each e-block type

To locate the most recent log interval that contains a modification to a variable, we first retrieve the list of e-blocks that contain the variable in their IMOD sets. The list of e-blocks is stored in the e-block table. We then locate either the most recent postlog produced by any of these e-blocks in the case of hidden dependence, or the most recent prelog in the case of dangling dependence. We finally generate the fine traces by using the emulation package for that e-block and the log entries for that log interval. This process may need to be repeated if the e-block did not actually modify the variable or if the last modification of the variable in the e-block occurred before a nested e-block that also potentially modifies the variable [11].

When we construct more than one e-block out of a subroutine because of debugging time efficiency considerations, we sometimes need to locate an e-block that might write a local variable. Unlike global variables, a variable local to a subroutine has an instance in each execution instance of the subroutine and we should not use log entries generated by different execution instances of the subroutine for the detailed dependence of a local variable.

5.5. Arrays and the Log

For an e-block with array accesses, it is not possible to compute IUSE and IMOD sets that contain only those array elements that are actually accessed in the e-block. One approach is to generate a log entry for the entire array even if only a few array elements are accessed. A second approach is to simply trace every array access. However, both approaches can potentially generate large amount of traces during execution.

Our solution to this problem is as follows. We distinguish two types of array accesses: *systematic accesses* and *random accesses.* We say there is a systematic access to an array if the array is accessed in a loop and the array index has a possibly transitive data dependence on the loop control variable. With a systematic access, we regard the entire array as accessed and generate a log entry (as usual) for the entire array. We regard all the other types of accesses to arrays as random accesses and generate a special log entry for the array index and the accessed value (read or updated value) of the array element at the time the access is made.

6. PARALLEL PROGRAMS AND FLOWBACK ANALYSIS

The discussion so far has described mechanisms to efficiently implement flowback analysis for sequential programs. In this section, we discuss the mechanisms for extending flowback analysis to parallel programs. For parallel programs, data dependences may exist across process boundaries. Locating such data dependences involves constructing an abstraction of the dynamic graph that contains the events belonging to all processes, and then ordering the events in this graph. With additional logging of shared variables, the incremental tracing scheme described in Section 5 can then be used to establish dependences between processes. In addition, potential data races in the program execution can be detected.

6.1. Parallel Dynamic Graph and Ordering Concurrent Events

To apply flowback analysis to parallel programs, we construct an abstraction of the dynamic graph, called the *parallel dynamic graph*, that contains the events belonging to all processes in the program execu-

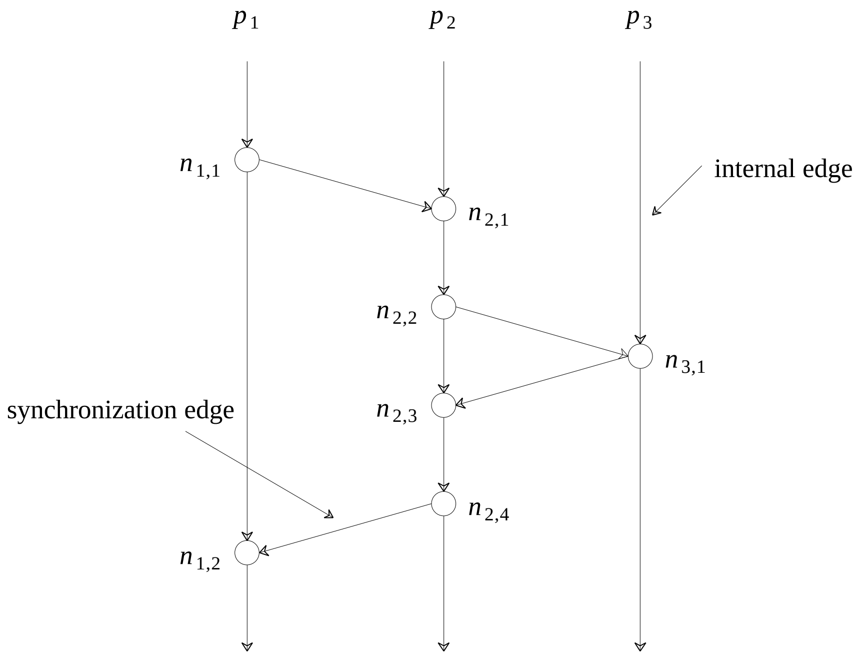


Figure 6.1. An Example Parallel Dynamic Graph

tion. To this graph we add edges that allow us to determine the order in which these events executed. From this ordering, data dependences can be established across process boundaries, and data races can be detected. We now describe how to construct the parallel dynamic graph and what run-time information must be recorded to do so. We then show how this graph orders the events belonging to different processes, and how this ordering allows intra-process data dependences, and data races, to be detected.

6.1.1. Parallel Dynamic Graph

The *parallel dynamic program dependence graph* (or *parallel dynamic graph*) is an abstraction of the dynamic graph that shows the interactions between processes while hiding the detailed dependences of local events. This graph contains only one node type, the *synchronization node*, and two edge types, the *synchronization edge* and *internal edge* (Figure 6.1 shows an example of a parallel dynamic graph). A synchronization node is constructed for each synchronization operation in the program execution. A synchronization edge from one node to another indicates that the first synchronization operation executed before the second. An internal edge abstracts out all events (belonging to the same process) that executed between the synchronization operations connected by the edge. For example, in Figure 6.1, all the events of process p_1 that executed before event $n_{1,1}$ also executed before all those events of process p_2 that executed after event $n_{2,1}$. The synchronization edge between $n_{1,1}$ and $n_{2,1}$ can be viewed as a generalized flow edge that spans the two processes.

We now describe how to construct synchronization edges for programs that use semaphores. Other synchronization primitives (such as messages, rendezvous, etc.) can also be handled [11]. In general, we construct a synchronization edge between two nodes if we can identify the temporal ordering between them. We say that the *source node* of an edge is the node connected to the tail of the edge, and the *sink node* of an edge is the node connected to the head of the edge.

Semaphore operations, such as P and V, are used in controlling accesses to shared resources by either acquiring resources (through a P operation) or releasing resources (through a V operation). We construct a synchronization edge from the node representing each V operation to the node representing some P operation on the same semaphore. Each V operation, which releases resources, is paired with the P operation that acquires those released resources.

There are two cases to be considered. The first case is where the second process tried to acquire the resources before the first process released them; the second process thus blocked on the P operation until the V operation of the first process. The second case is where the first process released the resources before the second process tried to acquire them; the second process did not block on the P operation in this case. In both cases, we define a source node for the V operation and a sink node for the corresponding P operation. The operations on a semaphore variable are serialized by the system that actually implements semaphore operations, and identifying a pair of related semaphore operations is done by matching the *n'th* V operation to the *(n+i)'th* P operation on the same semaphore variable, where $i(\geq 0)$ is the initial value of the semaphore variable.

Additional logging is necessary to record the information required to determine this semaphore pairing. Each semaphore operation generates a log entry (for the process it belongs to) containing a counter indicating how many operations on the given semaphore have previously been issued. The semaphore operations can easily be paired and the synchronization edges constructed from these log entries.

6.1.2. Ordering Events

In the parallel dynamic graph, each internal edge represents the set of events bounded by the surrounding synchronization operations. The order in which two events executed can be determined if there is a path between the two internal edges that represent those events (if no such path exists, then the actual execution order cannot always be determined). We partially order the nodes and edges of the parallel dynamic graph by defining the *happened-before* relation [28], $\rightarrow$, as follows:

1) For any two nodes n_1 and n_2 of the parallel dynamic graph, $n_1 \rightarrow n_2$ is true if n_2 is reachable from n_1 by following any sequence of internal and synchronization edges.

2) For two edges e_1 and e_2, $e_1 \rightarrow e_2$ is true if $n_1 \rightarrow n_2$ is true where n_1 is the sink node of the edge e_1, and n_2 is the source node of the edge e_2.

There are several approaches to ordering events in a parallel program execution [11, 15, 17, 18, 28, 33]. Although the ordering between two events can be determined by searching for a path in the graph, a more efficient representation of the happened-before relation can be constructed that allows the order between any two events to be determined in constant time. Such a representation is constructed by scanning the graph and computing, for each node, vectors that show the earliest (or latest) nodes in all processes that happened before (or after) that node [11].

6.1.3. Data Races

Once the events in the execution of a parallel program have been ordered, flowback analysis can be performed. Dependences that span process boundaries can be successfully located when the execution is *data-race* free. Once these dependences are located, the incremental tracing scheme described in Section 5 can be extended to re-execute e-blocks belonging to different processes, allowing the dynamic graph to be constructed. We now show how to determine when the execution contains data races. In the subsequent subsections, we show how to locate dependences that span processes and how to extend incremental tracing to parallel programs.

When the user requests to see the dependence for a read of a shared variable, ‘‘SV’’, we must locate the event that assigned the value to ‘‘SV’’ that was read. Locating this event involves finding all events that wrote ‘‘SV’’ and determining their order relative to the read event. However, when two events both access ‘‘SV’’ and are unordered by the happened-before relation, not enough information is available to determine which access occurred first. If at least one of the accesses is a write, then a potential *data race* is said to exist. A data race is usually a program bug, and exists when two events both access a common shared variable (that at least one modifies) and either did execute concurrently or had the potential of doing so.

Definition 6.1

Two edges e_1 and e_2 are *simultaneous edges* if $\neg (e_1 \rightarrow e_2) \wedge \neg (e_2 \rightarrow e_1)$.

Definition 6.2

$READ\_SET(e_i)$ is the set of the shared variables read in edge e_i. $WRITE\_SET(e_i)$ is the set of the shared-variables written in edge e_i.

Definition 6.3

We say two simultaneous edges e_1 and e_2 are *data-race free* if all the following conditions are true:

a) WRITE_SET(e_1) $\cap$ WRITE_SET(e_2) = $\emptyset$.

b) WRITE_SET(e_1) $\cap$ READ_SET(e_2) = $\emptyset$.

c) READ_SET(e_1) $\cap$ WRITE_SET(e_2) = $\emptyset$.

Definition 6.4

A program execution is said to be *data-race free* if all pairs of simultaneous edges in the execution are data-race free.

To determine when the program execution is data-race free, additional tracing must be performed to record the shared variables that are read and written by the events represented by each internal edge. For this purpose we maintain bit-vectors (representing basic blocks) during execution, and set a bit every time execution enters a basic block[6]. The size of these bit-vectors is computed at compile-time (by inspecting the simplified static graph, described in the next subsection). From the run-time trace of these bit-vectors, the sets of scalar shared variables that were read and written can be determined. To determine which shared array elements are accessed, we trace each array access. Our mechanism for logging randomly accessed arrays (described in Section 5.5), which must be employed anyway for flowback analysis, will provide the necessary information. However, for systematic accesses, we must additionally trace each shared array access. Since only the access type (either read or write) must be recorded, and not the value, optimizations can be performed to reduce the associated execution-time overhead. For example, instead of writing a trace record for each access, regions of the array that were accessed can sometimes be summarized[7] by a single record, resulting in a trace whose length is proportional to a small fraction of the number of array elements accessed.

The execution can be analyzed for the presence of data races in one of two ways. Either the entire execution can be checked for data races at one time, or data races can be detected only when the user follows back dependences. In either case, we can only detect when a program execution is data-race *free*. When an execution is not data-race free, a set of *potential* data races (between edges that are not data-race free) is reported. Only potential data races are reported because when two edges are simultaneous, it does not necessarily mean that all the events comprising the edges executed concurrently or had the potential of doing so. Rather, it means that the program's *explicit* synchronization did not prevent the events from executing concurrently; accidental synchronization (through the use of shared variables) can still prevent them from executing concurrently. However, this approach always detects data races when they exist and only reports a data race when at least one occurs [33, 34].

This data race detection scheme is similar to other methods [6, 14, 19, 33, 34], with the exception of pairing the P and V operations.

6.1.4. Data Dependences for Parallel Programs

When the user requests to see a dependence for a read of a shared variable, ‘‘SV’’, we must locate the most recent modification to that variable. This dependence is located by finding the event that assigned the value to ‘‘SV’’ that was read. This event is the one that wrote ‘‘SV’’ that is most recently ordered before the read by the happened before relation. To locate this write event, the latest edge in each process that happened before the edge containing the read is located. These edges give a boundary beyond which all events either executed concurrently with or after the read event. Each process in the parallel graph is then scanned backwards from this boundary to find an edge that modified ‘‘SV’’. The ordering of all such write events is examined to determine which one executed last. A data dependence can then be drawn from this last event to the read event. A unique write event is guaranteed to be found if no data races involving ‘‘SV’’ exist (unless, of course, ‘‘SV’’ was uninitialized).

Figure 6.2 shows an example of a parallel graph in which a shared variable ‘‘SV’’ is read by process p_3 and modified by processes p_1 and p_2. To establish the data dependence edge for ‘‘SV’’, the most recent modification of ‘‘SV’’ that occurred before the read must be located. If events belonging to edges $e_{2,1}$ (the edge emanating from node $n_{2,1}$) and $e_{1,0}$ (the topmost edge of process p_1) are the only modifications of ‘‘SV’’ then a data dependence is established between the event in $e_{2,1}$ that modified ‘‘SV’’ and the event in $e_{3,1}$ that read ‘‘SV’’ If, for example, there exists another event that modified ‘‘SV’’ in any of the edges $e_{1,1}$, $e_{1,2}$, $e_{2,2}$, $e_{2,3}$, or $e_{2,4}$ (i.e., edges simultaneous to $e_{3,1}$), then we cannot tell which event actually modified ‘‘SV’’ last, and a data race is reported to the user.

6.2. Incremental Tracing For Parallel Programs

Our implementation of incremental tracing described in Section 5 relied on the reproducibility of the debugged program. We now discuss applying incremental tracing to shared-memory parallel programs that lack reproducibility. Our solution uses a graph called the *simplified static graph*, which is a subset of the static graph that abstracts out everything except the synchronization operations between processes. From this graph, we determine what additional logging is required to support incremental tracing.

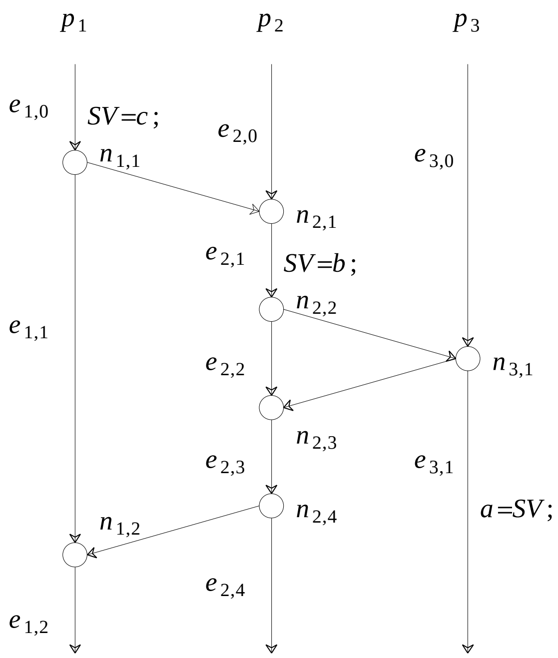


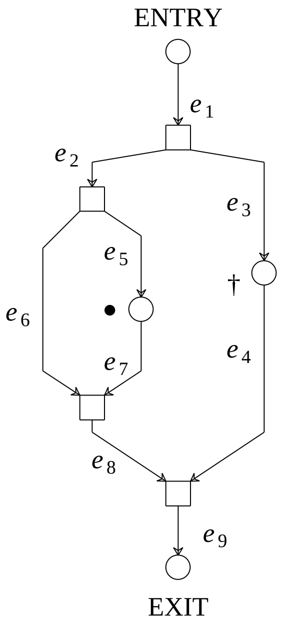
Figure 6.2. Dependences That Span Processes

6.2.1. Simplified Static Graph

To motivate the construction of the simplified static graph, consider the example shown in Figure 6.3, which contains a subroutine that accesses a global variable named ''SV''. The subroutine also constitutes an e-block. The statement indicated by the arrow is the first statement that accesses the variable ''SV'' in this subroutine. In the case of a sequential program, we construct a prelog that saves the value of ''SV'' at the beginning of the subroutine. The value of ''SV'' will not be changed until it is first accessed in the statement indicated by the arrow. Hence, one prelog and one postlog is sufficient to obtain reproducible behavior when re-executing parts of sequential programs during debugging.

However, now consider the case of a parallel program. If ''SV'' is a shared variable, we cannot guarantee that the value of ''SV'' saved in the prelog at the beginning of the subroutine will be the same as when ''SV'' is first read; other processes may have changed the value of ''SV'' between these two moments. Re-execution of this e-block may therefore perform a different computation than was originally performed during execution. In general, more run-time information must be recorded to ensure

```
SubB ()
{
  int a, b, c, p, q;
  if (p==...)
    if (q==...) {
         ⋮
    } else { /* q */
         ⋮
      SubC();
    } /* q */
  else { /* p */
     SV=a+b+SV;
     SubA();
       ⋮
  } /* p */
} /* SubB */
```



□ : branching node

○ : non-branching node

† This node corresponds to the subroutine call of `SubA`

• This node corresponds to the subroutine call of `SubC`

Figure 6.3. A Subroutine and Its Simplified Static Graph

reproducibility of parallel programs. Such additional information is used to restore the program state for read-accessed shared variables. The simplified static graph allows us to determine which shared variables must be recorded and where in the program they should be logged. In our examples, we only consider semaphore operations; however, this approach can be generalized to other synchronization primitives.

The simplified static graph is a subset of the static graph that contains only flow edges and nodes that represent either possible control transfers (such as **if** or **case** statements) or semaphore operations (Figure 6.3 also shows the simplified static graph for subroutine *SubB*). Any sub-graph node representing a subroutine that may perform a semaphore operation during its execution (or during the execution of any subroutine that may be transitively called by it) is treated as a semaphore operation. The simplified static graph therefore contains only *branching* nodes, which represent possible control transfers, and *non-branching* nodes, which represent possible semaphore operations.

6.2.2. Synchronization Units and Additional Logging

To generate the additional logging for shared variables, the simplified static graph is partitioned into *synchronization units*, which identify which shared variables to record and where in the program they should be logged.

Definition 6.5

A *synchronization unit* consists of all the edges that are reachable from a given non-branching node in the simplified static graph without passing through another non-branching node.

The sets $\{e_1, e_2, e_3, e_5, e_6, e_8, e_9\}$, $\{e_4, e_9\}$, and $\{e_7, e_8, e_9\}$ in Figure 6.3 each constitute a synchronization unit.

The object code generates an additional prelog at the beginning of each synchronization unit for those shared variables that are potentially read-accessed inside the synchronization unit. There is no corresponding postlog generated for the write-accessed shared variables at the end of a synchronization unit, as the regular logs generated at the beginning and end of the e-block contain the values of both shared and non-shared variables. The additional prelog of the read-accessed shared variables is used to ensure repeatable re-execution of the events in the synchronization unit. As long as there were no data races during execution, the additional prelog will suffice for ensuring repeatable execution behavior during debugging.

7. PERFORMANCE MEASUREMENTS

This section presents measurements of the overhead caused by PPD on execution time of application programs. We compare the execution time of the object code generated by the PPD compiler with that generated by the Sequent Symmetry C Compiler. We also present measurements of execution-time trace size. There is a trade-off between the amount of trace generated during execution time and the amount generated during debug time. The trade-off is based on selecting the size and location of e-blocks. Our current heuristics for making this selection are quite simple, so the performance numbers give only an initial indication of the cost of using PPD.

We present measurement results of five test programs: SORT, MATRIX, SH_PATH_1, SH_PATH_2, and CLASS. SORT sorts a vector of 100 integers using an Insertion Sort algorithm, whose time complexity is $O(n^2)$. MATRIX multiplies two square matrices of integers into a third matrix. The size of each matrix, for our tests, is 100 by 100. MATRIX uses a subroutine in multiplying two scalar elements of the two matrices. The subroutine does not contain a loop or accesses to a static variable, making that subroutine a target of log optimization (see Section 5.3). SH_PATH_1 computes the shortest paths from a city to 99 other cities using an algorithm described by Horowitz and Sahni [20]. SH_PATH_2 is the same as SH_PATH_1 except that it computes the shortest paths from all of the 100 cities to all the other cities. CLASS is a program that emulates course registration for students, such as registering for courses, and dropping from courses. CLASS also can run as an interactive program.

7.1. Execution Time

The goal of the PPD design is to minimize execution-time overhead without unduly burdening the other phases of program execution. Figure 7.1 shows the execution-time overhead of the tested programs. Execution-time overhead ranges $0-330\%$ for object code that is not log-optimized. and $0-75\%$ for object code that is log-optimized. MATRIX has the largest performance improvement from log optimization. The execution-time overhead of MATRIX is reduced from 330.7% to 7.9%. MATRIX has a subroutine that is called one million (100 by 100 by 100) times by another subroutine. Without log optimization, each call to this subroutine generates a prelog-postlog pair, resulting in a large execution-time overhead (due to the one million prelog-postlog pairs). However, this subroutine does not have a loop or accesses to static variables; with log optimization, this subroutine becomes a non-eblock subroutine and the caller becomes the parent e-block. The non-eblock subroutine does not generate log entries, yielding a much smaller execution time. Accordingly, log optimization also causes MATRIX to have a large reduction in the size of execution-time traces.

Log optimization might actually produce a higher execution-time overhead if the non-eblock subroutine is never invoked due to conditional statements in the program; parent e-blocks of these non-eblock subroutines may generate additional log information for the non-eblock subroutines that are never invoked. However, we expect that such cases of losing by log optimization should be rare.

We also see that copying the contents of an entire array (for a log entry) at the beginning or at the end of a loop is inexpensive in terms of execution time overhead if most of the array elements are actually

		Sequent Compiler	PPD compiler w/o log optimization (overhead in %)		PPD compiler w/ log optimization (overhead in %)	
SORT	CPU	5.5	5.7	(3.6%)	5.7	(3.6%)
	Elapsed	5.6	6.1	(8.9%)	6.1	(8.9%)
MATRIX	CPU	12.7	52.5	(313.4%)	13.4	(5.5%)
	Elapsed	12.7	54.7	(330.7%)	13.7	(7.9%)
SH_PATH_1	CPU	1.1	1.8	(63.6%)	1.8	(63.6%)
	Elapsed	1.3	2.2	(69.2%)	2.2	(69.2%)
SH_PATH_2	CPU	107.0	105.5	(- 2.4%)	105.5	(- 2.4%)
	Elapsed	107.0	107.3	(2.8%)	107.3	(2.8%)
CLASS	CPU	0.3	0.4	(33.3%)	0.4	(33.3%)
	Elapsed	0.4	0.7	(75.0%)	0.7	(75.0%)

Figure 7.1. Test Program Execution Time Measurements (time in seconds)

accessed in the loop. Such is the case with program SH_PATH_2. However, if only a fraction of the array elements are accessed in a loop, dumping out an entire array can be expensive, as seen in test program SH_PATH_1. One possible way to reduce this overhead is to generate a smaller log entry containing only the particular row (or other part) of the matrix that is actually accessed by employing techniques for succinctly summarizing data accesses in arrays [7].

Array logging can also cause some interesting performance anomalies. Notice that test program SH_PATH_2 shows a slight improvement in CPU time (the sum of user and system time) with the code generated by the PPD compiler. The PPD compiler generates logging code immediately before the loop that accesses a large array; the logging code accesses the entire array. This extra access seems to affect the paging behavior (possibly at the architecture level) of the program, resulting in less execution time. We are currently investigating this anomaly.

	without log optimization	with log optimization
SORT	18209	18209
MATRIX	40120221	120217
SH_PATH_1	825517	825517
SH_PATH_2	417129	417129
CLASS	104508	104892

Figure 7.2. Execution-Time Trace Size Measurements (sizes in bytes)

Program CLASS can also run as an interactive program. While there is a 33% increase in CPU time and a 75% increase in elapsed time when CLASS ran using an input file, there was no noticeable difference in the response times when CLASS ran interactively.

7.2. Execution-Time Trace Size

Figure 7.2 shows the sizes of execution-time traces (log) generated by the test programs. As described before, program MATRIX has a substantial decrease in trace size from log optimization. Program CLASS has a slight increase in trace size from log optimization because of the reason described previously.

7.3. Trade-Off between Run Time and Debug Time

As described in Section 3, there is a trade-off between efficiency during execution and response time during debugging. If we construct an e-block in favor of the execution phase, debugging phase performance will suffer. On the other hand, if we construct an e-block in favor of debugging phase, execution phase performance will suffer.

Figure 7.3 shows the re-execution times and debug-time trace sizes of various e-blocks of the tested programs. The e-block from SORT consists of a singly-nested loop that sorts the list of numbers once. The e-block of MATRIX is made of a triply nested loop. By constructing a single e-block out of the triply nested loop of MATRIX, we were able to reduce the execution phase overhead, but with a large debug-time overhead: 166 seconds in re-execution time and about 58 Mbytes of debug-time trace. For a comparison, the execution time of MATRIX itself is about 13 seconds, and execution-time trace size is 0.12 Mbytes, with log optimization. The e-block of SH_PATH_1 in Figure 7.3 is constructed out of a singly nested loop that computes the shortest paths from a city to 99 other cities, while the e-block of SH_PATH_2 is constructed out of a doubly nested loop that computes the shortest paths from 100 cities to all the other cities. The e-block of SH_PATH_1 took about 5 seconds to execute with 1.3 Mbytes of trace, while the e-block of SH_PATH_2 terminated because the file system was full. At that time the e-block of SH_PATH_2 lasted more than 7 minutes with more than 100 Mbytes of trace. These two results suggest that it might sometimes be better to construct more than one e-block out of a nested loop. One alternative might be to generate more than one prelog-postlog pair for an e-block with long execution time (such as an

	Execution Time			Debug-Time Trace Size
	original	Re-execution		
	CPU	CPU	ELAPSED	
e-block 1 (SORT)	< 0.1	0.1	1.7	0.37 Mbytes
e-block 2 (MATRIX)	8.6	160.5	165.5	57.76 Mbytes
e-block 3 (SH_PATH_1)	0.1	3.8	4.8	1.24 Mbytes
e-block 4 (SH_PATH_2)	10.5	> 364.8	> 422.8	> 117.79 Mbytes

Figure 7.3. Re-execution Times and Trace Sizes
(time in seconds)

e-block made out of a nested loop). In this case, the decision whether to generate another prelog-postlog pair during the execution of an e-block could be made dynamically at execution time.

7.4. Summary of Measurements

In this section, we have provided performance measurements of the various parts of PPD. The measurements show increases in the execution time vary significantly (0%–86%) among the test programs. However, larger increases in the execution time come from test programs that access only part of arrays in loops. One possible way to reduce this overhead is to employ techniques for succinctly summarizing data accesses in arrays [7]. With a more sophisticated dependence analysis for such complex objects, we expect a reduction in the execution-time overhead.

Execution-time trace sizes are generally small (less than 1 Mbyte in all cases). However, the measurements show that we need more experiments and research to better balance between the trace size during execution and the response time during debugging.

The test programs used in the performance measurements of PPD are in general small in size. However, we think the results obtained with these program will scale up proportionally well with programs of large size. In general, the performance measurements of PPD described in this section have demonstrated the feasibility of the ideas and directions proposed in our approach for debugging parallel programs.

8. CONCLUSION

Debugging parallel programs with flowback analysis has several advantages. First, dependences can be followed backwards, allowing the programmer to directly see causal relationships. In parallel programs, the ordering of events allows dependences to be followed that span process boundaries. Focusing the programmer on the cause of the errors allows parts of the execution irrelevant to debugging to easily be ignored. Flowback analysis should therefore scale well to large parallel programs. Second, repeated execution of the program is not required. The overhead associated with repeatedly re-executing long-running (and possibly non-deterministic) programs is avoided. Finally, data race detection allows us to deal with one of the more difficult synchronization errors encountered in parallel programs.

The graphs and algorithms presented in this paper provide the foundation for the construction of the system that will perform efficient flowback analysis for parallel programs. Several ideas make efficient flowback analysis possible. The use of semantic analysis allows us to identify at compile time only those variables that are necessary to trace at execution time. The incremental generation of the detailed traces at debugging time further amortizes the cost of tracing over the interactive debugging session. The fragmented static graph structure used in PPD is easily built and is tailored to be the building block of the dynamic graph. With the inclusion of synchronization dependences, these graph structures generalize nicely to parallel programs.

There are several issues that must still be addressed in the PPD design. The most immediate issue is the handling of pointers and dynamic data structures. The methods described in Section 3 form a starting point, and we are currently working on this problem. The user interface design is another area that must be investigated. A graphical representation of program dependences can offer quick access to complex structures. But as the body of displayed information increases, these displays can quickly overwhelm the viewer. A careful trade-off between graphical and textual information using multiple views and supporting information will be necessary to provide an intuitive interface.

We believe that PPD can be a platform for more than interactive debugging. Currently, the decision about which variable's dependences to examine is made by the programmer. Flowback analysis could be integrated with a more automated decision making process. This might be a verification system based on formal specifications or an expert system based on debugging knowledge.

Many of the design decisions and heuristics in PPD must be evaluated in practice. A working prototype is under construction to test our decisions on real programs. These tests will allow us to evaluate overall effectiveness and to tune the algorithms for such things as selecting e-block sizes and handling large arrays. An initial implementation of PPD (including all of the facilities described in this paper) is running, using the C programming language, on a Sequent Symmetry shared-memory multiprocessor.

ACKNOWLEDGEMENTS

We wish to thank Ron Cytron for his valuable suggestions and perseverance through several drafts of this paper, and to thank the anonymous referees for their helpful comments and suggestions. We also wish to thank Fran Allen and Michael Burke for their support.

9. REFERENCES

[1] *Guide to Parallel Programming on Sequent Computer Systems,* Sequent Computer Systems, Inc. (1985).

[2] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesly, Reading, Mass. (1986).

[3] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar, ''A Framework For Determining Useful Parallelism,'' *Proc. of the ACM 1988 Intl. Conf. on Supercomputing,* pp. 207-215 (July 1988).

[4] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, ''An Overview of the PTRAN Analysis System for Multiprocessing,'' *Journal of Parallel and Distributed Computing,* pp. 617-640 (1988).

[5] R. Allen and K. Kennedy, ''Automatic Translation of FORTRAN Programs to Vector Form,'' *ACM Trans. on Prog. Lang. and Systems* **90**(4) pp. 491-542 (October 1987).

[6] T. R. Allen and D. A. Padua, ''Debugging Fortran on a Shared Memory Machine,'' *Proc. of the 1987 Intl. Conf. on Parallel Processing,* pp. 721-727 (August 1987).

[7] V. Balasundaram and K. Kennedy, ''A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations,'' *Proc. of the ACM SIGPLAN '89 Conf. on Prog. Lang. Design and Implementation,* pp. 41-53 Portland, OR, (June 1989).

[8] R. M. Balzer, ''EXDAMS - EXtendable Debugging and Monitoring System,'' *Proc. of AFIPS Spring Joint Computer Conf.* **34** pp. 567-580 (1969).

[9] J.P. Banning, ''An efficient way to find the side effects of procedure calls and the aliases of variables,'' *Proc. of the 1979 ACM Symp. on Principles of Prog. Lang.,* pp. 29-41 San Antonio, TX, (January 1979).

[10] J.M. Barth, ''A practical interprocedural data flow analysis algorithm,'' *Comm. of the ACM* **21**(9) pp. 724-736 (September 1978).

[11] J. D. Choi, ''Parallel Program Debugging with Flowback Analysis,'' *Ph.D. Thesis (Also Computer Sciences Dept. Tech. Rep. #871),* Univ. of Wisconsin-Madison, (August 1989).

[12] J. D. Choi and B. P. Miller, ''Code Generation and Separate Compilation in a Parallel Program Debugger,'' in *Research Monographs on Parallel and Distributed Computing,* ed. D. Padua, MIT Press and Pitman Publishing (1990).

[13] K. Cooper, K. Kennedy, and L. Torczon, ''The Impact of Interprocess Analysis and Optimization in the $\mathbf{R}^n$ programming Environment,'' *ACM Trans. on Prog. Lang. and Systems* **8**(4) pp. 491-523 (October 1986).

[14] A. Dinning and E. Schonberg, ''An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection,'' *Procs. of ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming,* Seattle, WA, (March 1990).

[15] P. A. Emrath, S. Ghosh, and D. A. Padua, ''Event Synchronization Analysis for Debugging Parallel Programs,'' *Supercomputing '89,* pp. 580-588 Reno, NV, (November 1989).

[16] J. Ferrante, K. Ottenstein, and J. Warren, ''The Program Dependence Graph and Its Use in Optimization,'' *ACM Trans. on Prog. Lang. and Systems* **9**(3) pp. 319-349 (July 1987).

[17] C. J. Fidge, ''Partial Orders for Parallel Debugging,'' *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging,* pp. 183-194 Madison, WI, (May 1988). Also appears in

SIGPLAN Notices **24**(1) (January 1989).

[18] D. P. Helmbold, C. E. McDowell, and J.-Z. Wang, ‘‘Analyzing Traces with Anonymous Synchronization,’’ *Proc. of the 1990 Intl. Conf. on Parallel Processing,* pp. II-70–II-77 St. Charles, IL, (August 1990).

[19] R. Hood, K. Kennedy, and J. Mellor-Crummey, ‘‘Parallel Program Debugging with On-the-fly Anomaly Detection,’’ *Supercomputing ’90,* New York, NY, (November 1990).

[20] E. Horowitz and S. Sahni, *Fundamentals of Data Structures,* Computer Science Press (1983).

[21] S. Horwitz, J. Prins, and T. Reps, ‘‘Integrating Non-interfering Versions of Programs,’’ *ACM Trans. on Prog. Lang. and Systems* **11**(3) pp. 345-387 (July 1989).

[22] S. Horwitz, P. Pfeiffer, and T. Reps, ‘‘Dependence Analysis for Pointer Variables,’’ *Proc. of the ACM SIGPLAN 1989 Conf. on Prog. Lang. Design and Implementation*, (1989).

[23] K. Kennedy, ‘‘A Survey of Data-flow Analysis Techniques,’’ *Program Flow Analysis: Theory and Applications, S. S. Muchnick and N. D. Jones, Eds.,* pp. 5-54 Prentice-Hall, Englewood Cliffs, N.J., (1981).

[24] B. Kernighan and D. Ritchie, *The C Programming Language,* Prentice Hall, Inc., Englewood Cliffs, New Jersey (1978).

[25] D. J. Kuck, Y. Muraoka, and S. C. Chen, ‘‘On the Number of Operations Simultaneously Executable in FORTRAN-like Programs and Their Speed-up,’’ *IEEE Trans. on Computers,* pp. 1293-1310 (December 1972).

[26] D. J. Kuck, *The Structure of Computers and Computations,* John Wiley and Sons, New York (1978).

[27] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe, ‘‘Dependence Graphs and Compiler Optimizations,’’ *Proc. of the 1981 ACM Symp. on Principles of Prog. Lang.,* pp. 207-218 Williamsburg, Va., (January 26-28 1981).

[28] L. Lamport, ‘‘Time, Clocks, and the Ordering of Events in a Distributed System,’’ *Comm. of the ACM* **21**(7) pp. 558-565 (July 1978).

[29] L. Lamport, ‘‘How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,’’ *IEEE Trans. on Computers* **C-28**(9) pp. 690-691 (September 1979).

[30] J. R. Larus and P. N. Hilfinger, ‘‘Detecting Conflicts Between Structure Accesses,’’ *Proc. of the ACM SIGPLAN 1988 Conf. on Prog. Lang. Design and Implementation,* pp. 21-34 Atlanta, Georgia, (June 1988).

[31] B. P. Miller and J. D. Choi, ‘‘A Mechanism for Efficient Debugging of Parallel Programs,’’ *Proc. of the ACM SIGPLAN 1988 Conf. on Prog. Lang. Design and Implementation,* pp. 135-144 Atlanta, Georgia, (June 1988).

[32] B. P. Miller, ‘‘The Frequency of Dynamic Pointer References in ‘‘C’’ Programs,’’ *SIGPLAN Notices* **23**(6) pp. 152-156 (June 1988).

[33] R. H. B. Netzer and B. P. Miller, ‘‘Detecting Data Races in Parallel Program Executions,’’ in *Languages and Compilers for Parallel Computing,* ed. D. Gelernter, T. Gross, A. Nicolau, and D. Padua, MIT Press (1991). Also appears in *Proc. of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing,* Irvine, CA, (August 1990).

[34] R. H. B. Netzer and B. P. Miller, ‘‘Improving the Accuracy of Data Race Detection,’’ *Proc. of ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming,* Williamsburg, VA, (April 1991).

[35] K. J. Ottenstein and L. M. Ottenstein, ‘‘The Program Dependence Graph In A Software Development Environment,’’ *SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).

[36] R. Towle, ‘‘Control and Data Dependence for Program Transformations,’’ *Ph.D. Thesis (Also Dept. of Computer Science Tech. Report 76-788)*, University of Illinois, Urbana-Champaign, (March 1976).

[37] M. Weiser, ‘‘Programmers Use Slices When Debugging,’’ *Comm. of the ACM* **25**(7)(July 1982).

[38] M. Weiser, ‘‘Program Slicing,’’ *IEEE Trans. on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).

[39] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press (1989).

TABLE OF CONTENTS