PARALLEL PROGRAM DEBUGGING WITH FLOWBACK ANALYSIS

by

Jongdeok Choi

Computer Sciences Technical Report #871

August 1989

Parallel Program Debugging

with

Flowback Analysis

by

Jongdeok Choi

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1989

ABSTRACT

This thesis describes the design and implementation of an integrated debugging system for parallel programs running on shared memory multi-processors. The goal of the debugging system is to present to the programmer a graphical view of the dynamic program dependences while keeping the execution-time overhead low.

We first describe the use of *flowback analysis* to provide information on causal relationship between events in a program's execution without re-executing the entire program for debugging. 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, we use 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. The cornerstone of the incremental tracing concept is to generate a coarse trace during execution and fill incrementally, during the interactive portion of the debugging session, the gap between the information gathered in the coarse trace and the information needed to do flowback analysis using the coarse trace.

Then, we describe how to extend flowback analysis to parallel programs. Flowback analysis can span process boundaries; i.e., the most recent modification to a shared 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.

Finally, we describe the implementation of a prototype debugging system (called PPD) and provide performance measurements of the various parts of PPD. We describe the code-generation techniques for separate compilation used by PPD compiler. In general, the performance measurements of PPD have demonstrated the feasibility of our ideas and directions for debugging parallel programs.

ACKNOWLEDGEMENT

I would like to thank my advisor, Barton P. Miller, for his assistance and direction during this research. I feel greatly indebted to him for his guidance and patience in improving my writing and communication skills.

I would also like to express my gratitude to the readers of this thesis: Michael J. Carey, and Thomas Reps, who spent time on my thesis work and gave me valuable comments on my thesis. I would also like to thank the other members of my committee: David J. Dewitt, Mary K. Vernon, and John Beetem, for providing helpful comments.

I have had the good fortune of having Dave Cohrs as my officemate. His expertise on Unix has been very helpful to me. I also want to thank Rob Netzer for the discussions we have had on my thesis work, and for his effort in implementing the Parallel Program Debugger (PPD).

Most of all, my family members deserve my greatest gratitude. I am deeply grateful to my parents, my brother, my sisters and my in-laws for their support and love. My wife, Misook, and my daughters, Eunji (Agnes) and Hyeji (Christine), deserve special thanks for their sacrifice and hardships. I have not been able to answer Eunji's question of why I go to school twice a day (in the morning and evening) while she goes to nursery school only once a day. I will be a full-time daddy for you, Eunji and Hyeji.

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 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. This thesis addresses the design and implementation of a debugging system for parallel programs running on shared memory multi-processors (SMMP). In particular, it describes strategies to allow for efficient debugging of parallel programs without re-execution. The approaches described in this thesis allow the programmer to locate a bug in a parallel program, given an error (the manifestation of the bug), easily and with low system overhead. They also allow for easy detection of timing errors in the interactions of the co-operating processes. These strategies are tested in a test implementation called the *Parallel Program Debugger (PPD).*

Using a model of system failures [48], we divide program debugging into four steps: (1) observe and compare the external program behavior with the desired behavior, defining a *failure* as the deviation of the program behavior from the desired behavior; (2) locate and analyze the *error* — the erroneous internal state of the program that led to the failure; (3) find the *bug* — the program element that caused the error; and (4) modify the program to eliminate the bug. The purpose of first and second steps is to detect the error, while that of the third is to locate the bug that caused the error. The fourth step, error correction, is usually accomplished using text editors.

A ***debugger*** is a software system that, with possible hardware assistance, helps the programmer in some or all of the first three steps of debugging. An ideal debugger would have three characteristics. First, it would help the programmer to detect errors easily. Second, after the errors are detected, it would help locate the bugs that caused the errors. Third, its use would not change the program behavior nor introduce any overhead in the system.

Debugging is a difficult job because the programmer has little guidance in locating the bugs. To locate a bug that caused an error, the programmer must think about the causal relationships between the

events in a 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 for the programmer to locate the bug. The usual method for locating a bug is to execute the program repeatedly, each time placing breakpoints (to detect an error) closer to the location of the bug. An easier way is to track the events backward from the moment of detecting an error to the moment at which the bug caused the error, as is done by the *flowback* analysis [9] approach proposed by Balzer. In flowback analysis, the programmer can see, either forward or backward, how information flowed through the program to produce the events of interest. In this way, the programmer can easily locate bugs that led to the detected 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 in their interactions between processes. Such non-determinism often makes it difficult to re-execute the program for debugging purposes.

Third, interactions between co-operating processes in a SMMP system are frequent and can happen by accessing data in shared memory without any explicit synchronization activities, such as P or V semaphore operations.

Finally, it is often difficult for a parallel debugger to maintain the transparency of the debugger. For example, inserting a print statement in a sequential program to aid in debugging usually does not change the program behavior except for the additional output. In a parallel debugger, however, inserting such a statement can change the timing of the interactions between co-operating processes of the program, and hence may alter the program behavior.

The main objective of this thesis is to find ways to allow the user to debug parallel programs efficiently. More specifically, we want to overcome the difficulties in debugging parallel programs running on SMMP systems while providing a close approximation to our ideal debugger. The resulting debugger will provide direct help in locating bugs, while maintaining low system overhead. The major strategies we use for achieving this objective and the features resulting from these strategies are as follows:

1) We provide the user with information on causal relationships between program events to ease the task of debugging. This strategy is adopted from the idea of flowback analysis proposed by Balzer.

2) We extend the semantics of flowback analysis to be able to debug parallel programs.

3) We locate the causes of the timing errors of parallel programs in the interactions between co-operating processes. Such timing errors usually manifest themselves in data races in accessing shared data elements or in the form of a deadlock of the processes. Data races are in most cases bugs with only a few exceptions [52].

4) We do not re-execute the program for debugging purposes; thereby sidestepping the need for reproducible execution behavior. The strategies herein can be applied to parallel programs that lack *reproducibility* — the characteristic of producing the same execution behavior given the same input. The cost of re-execution is also eliminated.

5) The system overhead during a program's execution is kept low by gathering only a small amount of information during program execution. The gap between the information gathered during the program's execution and the information needed to debug the program is filled during interactive debugging. The result is that we introduce little overhead in the system in applying these debugging strategies. The selection of information to gather is based, in part, on semantic analysis of the program.

6) We can provide a globally consistent snapshot of the program state during the debugging session. Such a snapshot allows the user to experiment with program execution by changing values of variables to see the effect of such changes on the program behavior.

In this thesis, we are addressing the class of parallel programs that use explicit synchronization primitives, such as the semaphore, monitor, or Ada rendezvous. While we are not addressing automatic parallelization [5], many of the techniques described in this thesis might be applied in such systems. The techniques in this paper are described in terms of the C programming language [31], but they should generalize to many procedural languages. This thesis addresses a large part of the C language, including primitives for synchronization, but we do not discuss pointers and heap-allocated storage; that is a topic of future investigation. We also discuss how our methods can be extended to handle aliases resulting from

reference parameters in languages like Pascal or FORTRAN.

This thesis is organized as follows. We discuss research related to this thesis work in Chapter 2. Chapters 3 through 6 address the issue of how to debug a program without re-execution while introducing only a low execution-time overhead. In particular, Chapter 3 describes how we divide the process of debugging into several steps and how each of these steps works toward the goal of efficient debugging of parallel programs.

The basic mechanism we use for low execution-time overhead is to transfer the execution-time overhead partly to compile time and partly to debug time. We perform semantic analysis of the program at compile time, and generate detailed execution traces at debug time. A major purpose of the semantic analysis is to build a graph that shows *potential* dependences between program statements. We describe this graph structure in Chapter 4.

We use the graph built at compile time and the detailed traces generated at debug time to build a graph, at debug time, that shows *actual* dependences between program statements. We describe, in Chapter 5, how to generate such traces and how to build the graph that shows actual execution dependences.

We extend the semantics of flowback analysis to parallel programs by using a graph, called the *parallel dynamic graph*, that shows the inter-process interactions. Chapter 6 describes the parallel dynamic graph and how we use this graph to extend the semantics of flowback analysis to parallel programs. Chapter 6 also describes how to construct the parallel dynamic graph.

In Chapter 7, we describe implementation details of the system and discuss the performance of the system. Finally Chapter 8 summarizes the thesis work.

Chapter 2

Related Work

This thesis addresses the problem of debugging parallel and distributed programs. We divide previous research in debugging into three categories: *cyclic debugging* (debugging with repeated execution of the program), debugging with *reverse execution*, and debugging with traces of program execution. While there is some overlap between these three approaches, our division is based on the primary emphasis of each category. In describing the related research, we first introduce the ideas and then discuss their advantages and disadvantages.

2.1. Debugging with Re-execution

Cyclic debugging is widely used for sequential programming, partly because it is easy to understand and use and partly because it is easy to implement. In applying cyclic debugging to parallel programs, a major concern is that of how to ensure the reproducible program behavior required by cyclic debugging.

The non-deterministic nature of parallel programs generally derives from the interactions between the co-operating processes. The various systems that apply cyclic debugging to parallel programs seek to make the process interactions reproducible. Methods by Curtis [19] and Schiffenbauer [51] generate traces of the order and contents of the messages exchanged during program execution. These traces are used not to locate bugs, but to ensure that executions during debugging reproduce the behavior of the original execution; the information in the traces is used during debugging to deliver the same messages in the same order as in the original execution. This is accomplished by re-routing all communications through a centralized debugger process. The main drawbacks of these approaches are twofold: the cost of re-routing and saving messages during execution, and the possible changes in program behavior due to the changes in the timing of the process interactions.

In an effort to reduce the cost of saving information about process interactions, the Instant Replay mechanism [39] records only the order of the interactions. Reproducible program behavior during the debugging session is guaranteed by using the same input from the external environment and by imposing

the same relative order of events during debugging. The major contribution of this approach is that it succeeds in reducing the overhead generated by saving the contents of the interactions between the co-operating processes. Instant Replay views process interactions as accesses to shared objects and allows for the debugging of parallel programs running on SMMP systems. One drawback of this approach is that it requires the programmer to use certain primitives correctly so that accesses to each shared object can be serialized. The incorrect use of these primitives can introduce errors which may themselves require debugging.

The main advantage of cyclic debugging is that the user need not anticipate the types of bugs that exist in the program until an error is detected. The program can be re-executed to produce the same execution behavior. Two disadvantages in applying cyclic debugging to parallel programs are as follows. First, executing the entire computation several times while repeatedly setting breakpoints is costly. To reduce the overhead of re-executing the entire computation during debugging, *checkpointing*[19] is sometimes used to save the program state periodically. When an error is detected, the user can reset the system state to that of the previous checkpoint and can thus avoid re-starting the entire computation from the beginning. Checkpointing is also used in implementing reverse execution (described in the next section).

Second, cyclic debugging is difficult to apply to non-deterministic programs. Such non-deterministic program behavior can come from scheduling delays, language constructs such as *guarded commands*[20], or from changes in the external environment. Database processes are a good example of the third case since their execution behavior may depend on the contents of the databases they access.

STAD [32], a cyclic debugging system for sequential programs, has an interesting feature of using data-flow and control-flow analyses to set breakpoints. When the program halts due to a program failure or a breakpoint, the system asks the user to name the variables whose values seem to be wrong. The system then automatically sets breakpoints at those statements that might be the latest statements that wrote to those variables, and then re-executes the program. However, this work has been confined to monolithic programs — programs without subroutines — due to the complexity of interprocedural data flow analysis.

Algorithmic Debugging is a theory of debugging that uses queries on the compositional semantics of the program to be debugged [40, 55]. At some level of abstraction (such as a procedure for a sequential

program and a process for a concurrent program) the behavior of a program component must depend only on those of the subcomponents it invokes. If some component returns an incorrect result while all the subcomponents in invokes return correct results, that component is erroneous. The debugger locates such erroneous components by re-executing each subcomponent and querying the user about the correctness of the subcomponent. The model language used by Algorithmic Debugging is Prolog, and it is not clear how the method can be applied to non-functional languages such as the C language and what the overhead might be when applied to such languages.

2.2. Debugging with Reverse Execution

Debugging is inherently a backward process — to locate a bug that caused an error, the programmer is interested in past events that might have caused the observed error. In cyclic debugging, the programmer executes the program repeatedly, each time placing breakpoints closer to the location of the bug. By contrast, in debugging with reverse execution [1,22,46], the programmer can follow events backward to see the causal relationship between program events. One way to implement reverse execution is to save the program state periodically (called checkpointing) and to re-execute the program from the most recent checkpoint. For this reason, *partial re-execution* might be a better term for this type of approach.

The primary goal for the implementation of the Recap [46] system is *simplicity* — checkpointing is done by forking a process. The potential drawback of forking a process for checkpointing is the cost — in time and space — to capture the program state at each checkpoint. However, in systems where copy-on-write sharing is used to implement a fork [62], only modified pages will actually be copied.

The IGOR [22] system also attempts to reduce the cost of checkpointing by making copies of only pages modified since the last checkpoint. However, IGOR uses two system calls that are specifically implemented for checkpointing. One of them is used to identify pages modified since the last checkpoint, using a concept similar to that used in virtual memory systems. At intervals, IGOR uses this system call to save a copy of every page that has been modified since the last checkpoint. If the program has good locality of reference, only a fraction of the pages will be saved at each checkpoint. The other system call is used in specifying the interval of checkpointing.

Another method to reduce the cost of checkpointing is to save only the latest instance of values of variables for each statement, as done by Agrawal and Spafford [1]. In this scheme, each statement of a program has a *change-set* associated with it, consisting of all the variables that might be written to by that statement. During execution, only the latest instance of values of variables belonging to the change-set of each statement will be saved, resulting in a space overhead (roughly) proportional to the size of the program. However, their method saves only the latest instance of values of variables for each statement in a loop without regard to the actual number of loop iterations. This has the drawback that many previous loop iterations may be re-executed when the user wants to backtrack from a statement after a loop to a statement inside the loop.

2.3. Debugging with Traces

Debugging with traces has the advantage that it is not necessary to force the behavior of a previous execution to be repeated. This advantage is an important reason why generating traces is more attractive for debugging parallel programs than for debugging sequential programs. A trace generated during execution is usually large in size and sometimes difficult to understand. A major concern in debugging with traces is how to help the user to understand the generated traces and detect the events of interest.

Event-specification languages help to specify the desired program behavior and thus to detect events by comparing observed events with the specification of the events to be detected. Bates and Wileden's *behavioral abstraction* (BA) [14] is an example of an approach that uses such an event-specification language. BA can help the user understand the execution behavior of a program by identifying higher level abstractions from the basic trace events. By using the *event description language* (EDL) [13], the user can specify which basic events form higher level events. Thus the BA helps the user to specify a hierarchical abstraction of events. The motivation for BA is to help the user detect errors by comparing the high level description of the program execution with the higher level abstraction of the event traces. Other examples of event-description language are given in [8, 15].

Temporal logic [21, 24, 47, 49] is another form of event-specification language, one capable of expressing behavior over time by providing a formalism for describing the occurrence and patterns of events in time. For example, using temporal logic the user can formalize the specification of program

states over time such as "always **true**," "sometimes **true**" and "eventually **true**" [21]. Snodgrass's relational database query language, TQuel [57], incorporates concepts drawn from both event-specification languages and temporal logic. It defines an ***event expression*** and a ***temporal expression*** as part of a query expression so that the user can use any of them in a query (if necessary).

The main advantage of debugging with traces is that reproducible behavior is not required of the program, provided that trace generation is used during executions. This approach can therefore be applied to both deterministic and non-deterministic programs. Its main disadvantage is that generating traces all the time during execution can be costly in both time and space. To reduce the amount of trace information, the user must be able to anticipate the events for which traces should be generated and the granularity at which the traces should be generated. If the user cannot anticipate which events to trace, the user has to make traces of a large number of events so as not to miss any important events which may prove useful in error diagnosis.

The granularity of the events used to generate traces is also related to the size and usefulness of the generated traces. With a coarser granularity, the amount of trace data will be smaller but the generated trace may miss important details of interesting events. With a finer granularity, the generated trace will miss fewer important details, but the amount of generated trace data will be larger. Another drawback of event-specification languages is that if the specification of the events is complicated, the event specification itself may need to be debugged.

2.4. Flowback Analysis

In his early work on debugging [9], Balzer suggested a number of interesting approaches. These include generating traces during execution, and replaying the program execution either in the forward or backward direction during debugging. Balzer also suggested the use of flowback analysis as an aid to the user in locating bugs. When the user requests a flowback analysis for the value of a variable at a point in the program execution, the system analyzes how the information flowed through the program to reach the point for the variable to have the value. The system presents the result of the analysis in the form of an inverted tree consisting of nodes and edges, with the bottom node corresponding to the initial point of interesting (i.e., the point the user requested flowback analysis).

The nodes in the inverted tree represent execution events and the edges represent the data dependences between the events represented by nodes. Each node is associated with both a value and the program statement that produced that value. An edge from node *A* to node *B* represents data dependence from event *A* to event *B*: the statement associated with event *B* made use of the value produced by event *A*.

Balzer's idea is unique among the trace-based approaches in that it is an attempt to give direct help in locating bugs by showing the past flow of program execution and, in doing so, by showing the causal relationship among events. In a sense, Balzer's flowback analysis has much in common with the idea of reverse execution. However, reverse execution fails to provide the programmer with information on the causal relationships between the events in a program's execution. Since Balzer's suggestion is based on traces of events generated during program execution, it shares the advantages and disadvantages of other trace based approaches: it does not require reproducibility of the debugged program, but it can be costly to in generate the necessary traces.

2.5. Summary

Debugging parallel programs has been studied from many perspectives. By recording just the order of interactions between processes, Instant Replay succeeds in reducing run-time overhead to ensure reproducible program behavior of parallel programs in debugging with repeated execution.

Event specification languages such as Bates and Wileden's event-description language (EDL) provide help in understanding program behavior and specifying events of interest to the user, but have the drawback that if the specification of the events becomes complex, as is often the case when debugging parallel programs, the specification itself becomes susceptible to bugs. Also, event-specification languages are used with traces, which can be costly to generate.

The idea of reverse execution is similar to that of flowback analysis. However, unlike flowback analysis, reverse execution does not provide the programmer with information about causal relationships between events in a program's execution; the task of locating the events that cause a given event is still left to the programmer.

Although the approaches mentioned above generally provide good ways to help the programmer detect errors, except for flowback analysis they lack the ability to give guidance in locating bugs. The

programmer still has to guess the whereabouts of the bug that caused an error that has been detected. What we suggest as a valuable debugging facility is one that provides direct help to the user in locating the bug that caused an error while keeping the overhead introduced in the system to a minimum. Balzer's flowback analysis suggests one way in which a debugger can provide the user with guidance in locating bugs. However, his suggestion is based upon tracing and shares the advantages and disadvantages of other trace based approaches: it does not require reproducibility of the debugged program, but it can be costly in generating the required traces.

In this thesis, we describe the use of flowback analysis in debugging parallel programs. We first describe how to keep execution-time overhead 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 execution-time overhead low. We then describe how to extend the flowback analysis to parallel programs. The flowback analysis can span process boundaries; i.e., the most recent modification to a shared variable might be traced to a different process than the one that contains the current reference.

Chapter 3

Overview of the Debugging System

Our approach to debugging parallel programs is to provide direct help in locating bugs without re-executing the program. To provide direct help to locate bugs, we have adopted flowback analysis [9] to show the actual run time dependences. However, execution-time overhead is kept low by recording only a small amount of trace during execution.

3.1. Flowback Analysis and Incremental Tracing

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. The first option is expensive, and most often not practical for parallel programs because of unacceptable changes the debugger would introduce in timing of the interactions between processes. The second option is not practical for programs that lack reproducibility, as is often the case with parallel programs.

The strategy adopted here to overcome these difficulties is to use ***incremental tracing*** based on the idea of ***need-to-generate***. For the user, the effect will be the same as that of generating a trace of every event and state change of the program during execution, but with far less run time overhead since not every event will actually be traced. The cornerstone idea of the need-to-generate concept is to generate a small amount of information, called a *log*, during execution. We then fill incrementally, during the interactive portion of the debugging session, the gap between the information gathered in the log and the information needed to do flowback analysis using the log. In this manner, we can transfer the cost of generating traces from execution time to debug time, and also partly to compile time since we generate static information during compilation. We can reduce the run-time overhead of producing the log by applying

interprocedural analysis [18] and data flow analysis [30] techniques commonly used in optimizing compilers. Incremental tracing will be described in more detail in Chapter 5.

3.2. Three Phases of Debugging

We divide debugging into three phases: a *preparatory* phase, an *execution* phase, and a *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 files to be used in the debugging phase. During the execution phase, the object code generates the log that will be used in the debugging phase. The debugging phase begins when the program halts, either due to an error or to user intervention. More details on the structure of the compiler/linker will be given in Chapter 7.

3.2.1. Preparatory Phase

Figure 3.1 shows a simplified diagram of the components involved in the preparatory phase. Along with the object code, the Compiler/Linker produces the following additional items during the preparatory phase:

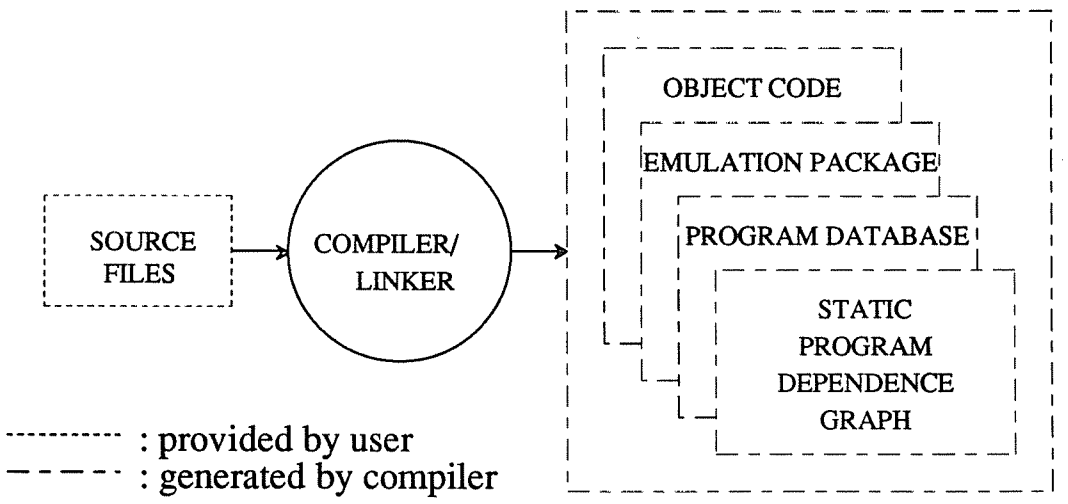


Figure 3.1.
The Preparatory Phase

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

2) the *static program dependence graph*, which shows the static (possible) data and branch dependences among components of the program;

3) the *program database*, which contains information on the program text such as the places where an identifier is defined or used.

3.2.2. Execution Phase

The object code produced during the preparatory phase plays a major role in the execution phase. Figure 3.2 shows the structure of the execution phase, during which the object code generates program output as well as a log that records dynamic information about program execution. The log is used by the emulation package during the debugging phase to generate traces for flowback analysis. Among the log entries are *postlogs*, which record changes in the program state since the last logging point, and *prelogs*, which record the values of the variables that might be read-accessed before the next logging point. The postlogs also allow for the program state to be restored to a previous moment of execution. The user could change the values of variables and re-start the program from the same point to see the effect of these

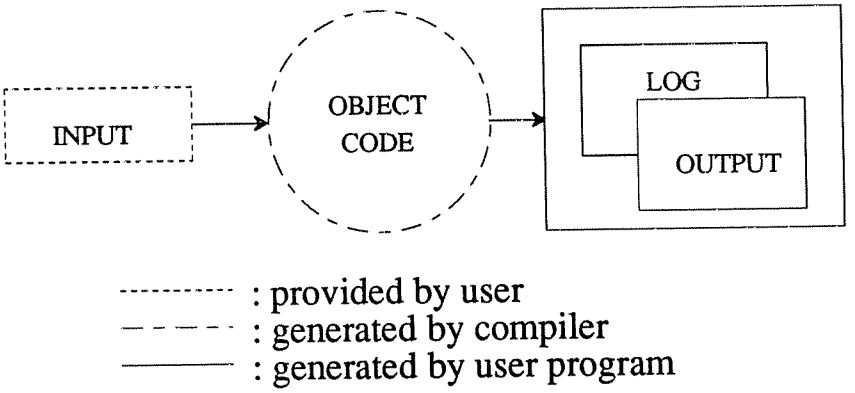


Figure 3.2.
The Execution Phase

changes on program behavior. The reason why it is necessary to generate prelogs as well as postlogs will become clear in Chapter 5.

3.2.3. Debugging Phase

Figure 3.3 shows the debugging phase, during which the PPD Controller plays the major role. The edges toward the PPD Controller in Figure 3.3 represent the sources of the information it uses to build the *dynamic program dependence graph*. The dynamic program dependence graph (also called *dynamic*

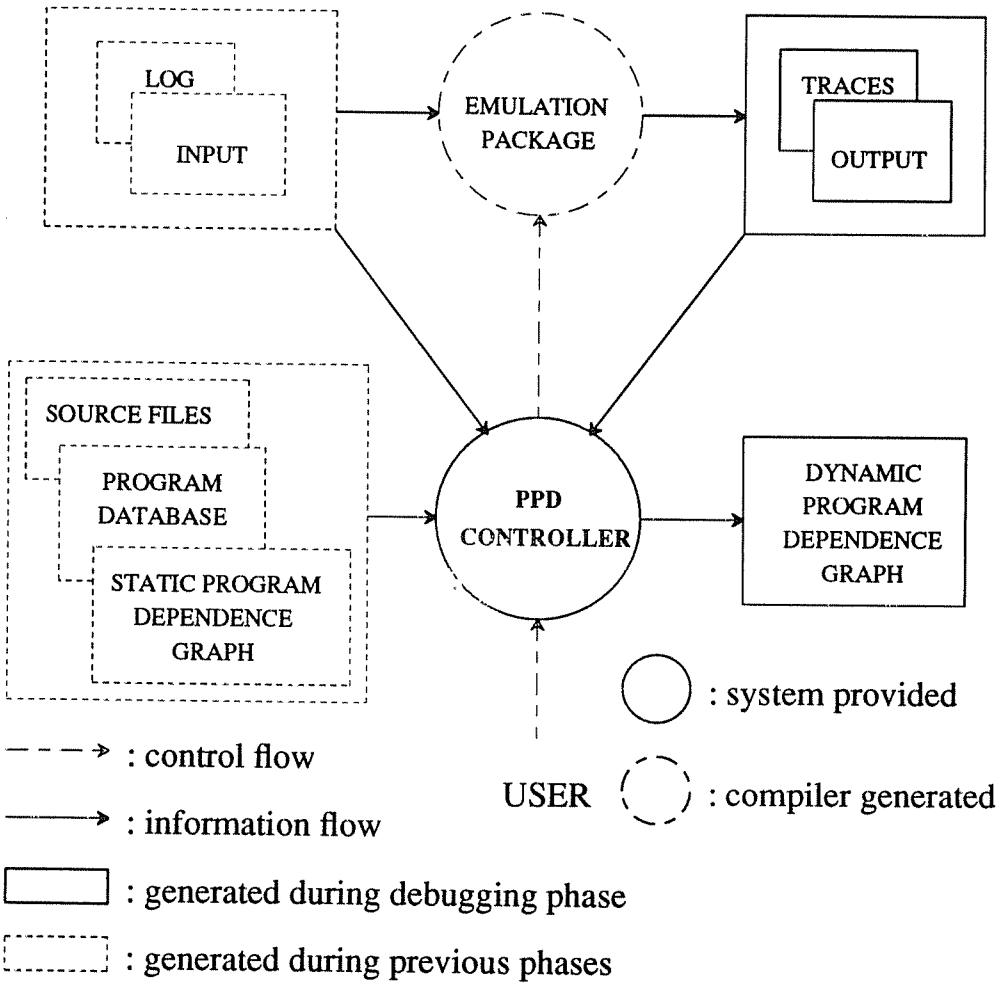


Figure 3.3.
The Debugging Phase

graph) shows the run-time dependences among program components, and is built incrementally during debugging. In building the dynamic graph, the PPD Controller calls the emulation package to obtain the necessary traces.

When the debugging phase starts, the PPD Controller presents the user with a portion of the dynamic graph that has the last statement executed being the root of an inverted tree. There exists a practical limit to the size of the graph, determined by the screen size. Furthermore, it is useless to provide a graph whose size is beyond the user's grasp. Thus, the portion of the dynamic graph presented to the user at any time is small in size, and the traces needed at one time in building a portion of the dynamic graph are also small in size.

When the user wants to see the dependences of events not seen in the current portion of the dynamic graph, the PPD Controller draws a new portion of the dynamic graph that shows the requested dependences. There are two possible cases here. In the first case, there are sufficient traces already generated to show the dependences requested by the user; the PPD Controller merely updates the portion of the dynamic graph presented to the user to show the requested dependences. In the second case, there are not sufficient traces; the PPD Controller directs the emulation package to generate the traces needed to show the requested dependences. In directing the emulation package, the PPD Controller consults with static information, such as the static program dependence graph and the program database, both of which were produced during the preparatory phase. Since not all events of a program execution are of interest to the user, not all events will generate traces during debugging. More details on tracing and the dynamic graph are provided in Chapter 5.

Chapter 4

Static Program Dependence Graph

The static program dependence graph (static graph) shows potential dependences between program components, such as *data dependences* [28] and *branch dependences*. The static graph is also the basic building block of the dynamic program dependence graph (dynamic graph).

The static graph is a variation on the dependence graphs introduced by Kuck [34]. Since then, there have been numerous variations which can be categorized into two classes according to their applications. First, the program dependence graph is often used as an intermediate program representation for the purpose of optimizing, vectorizing, and applying parallelizing transformations to a program [23, 33-35, 58]. The main concern here is to decide whether any potential dependences exist between two sets of program statements.

Second, the program dependence graph is also 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 statements that might affect the value of v at p [59]. Such slices can be used for integrating program variants [28] and for program debugging [23, 45, 59, 60]. 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. One common attribute of the two classes of static dependence graph applications is that neither uses dynamic information obtained during program execution. However, in Parallel Program Debugger (PPD), we augment the static graph with dynamic information, obtained during execution and debugging, to build the dynamic graph. Accordingly, the static graph structure in PPD differs in several ways from those of previous systems.

The structure of the static graph in PPD 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 data generated at execution time. A small amount of trace data 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. For example, to show execution-time dependences, it is not important to have identified dependences as *loop carried* [6] or *loop*

independent at compile time. 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 in showing the dependences requested by the user.

In this chapter, we describe a static graph consisting of two layers. The outer layer, called the *branch dependence graph (BDG)*, shows the branch dependences; the inner layer, called the *data dependence graph (DDG)*, shows the data dependences within the blocks of the branch dependence graph. We first discuss the two layers in detail, and then we describe the algorithms to build the data dependence graph. Interprocedural analysis is used in building the data dependence graph. With separate compilation, interprocedural analysis also allows us to avoid rebuilding the whole static graph from scratch when one or more modules of the program are modified. The separate compilation issue is described in detail in Section 4.5, where we describe how we use interprocedural analysis in building the static graphs.

Pointers and aliases make the semantic analysis of the program difficult. Currently, we handle pointers by simply tracing all uses of pointers in the log. This approach will be viable if the dynamic frequency of pointer references is low [43]. 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 [27,38]. We describe how to handle parameter aliases in Section 4.6.

4.1. Branch Dependence Graph (BDG)

The branch dependence graph consists of the branch dependence edges and nodes called control blocks. Figure 4.1 shows the branch dependence graph corresponding to subroutine "Wolf". There is one such branch dependence graph for each subroutine of the program, and each leaf block of the branch dependence graph represents a block of statements that are devoid of any conditional or loop control statements. Flow of control enters at the beginning of each control block, and the execution order of the statements in each leaf control block is strictly sequential; the leaf block thus represents a basic block. For a program with no **goto** statements in it, the branch dependence graph in our system would be identical to the *control dependence subgraph* described by Ferrante, et al. [23]. We describe how **goto** statements affect the structure of dynamic graphs in Chapter 5.

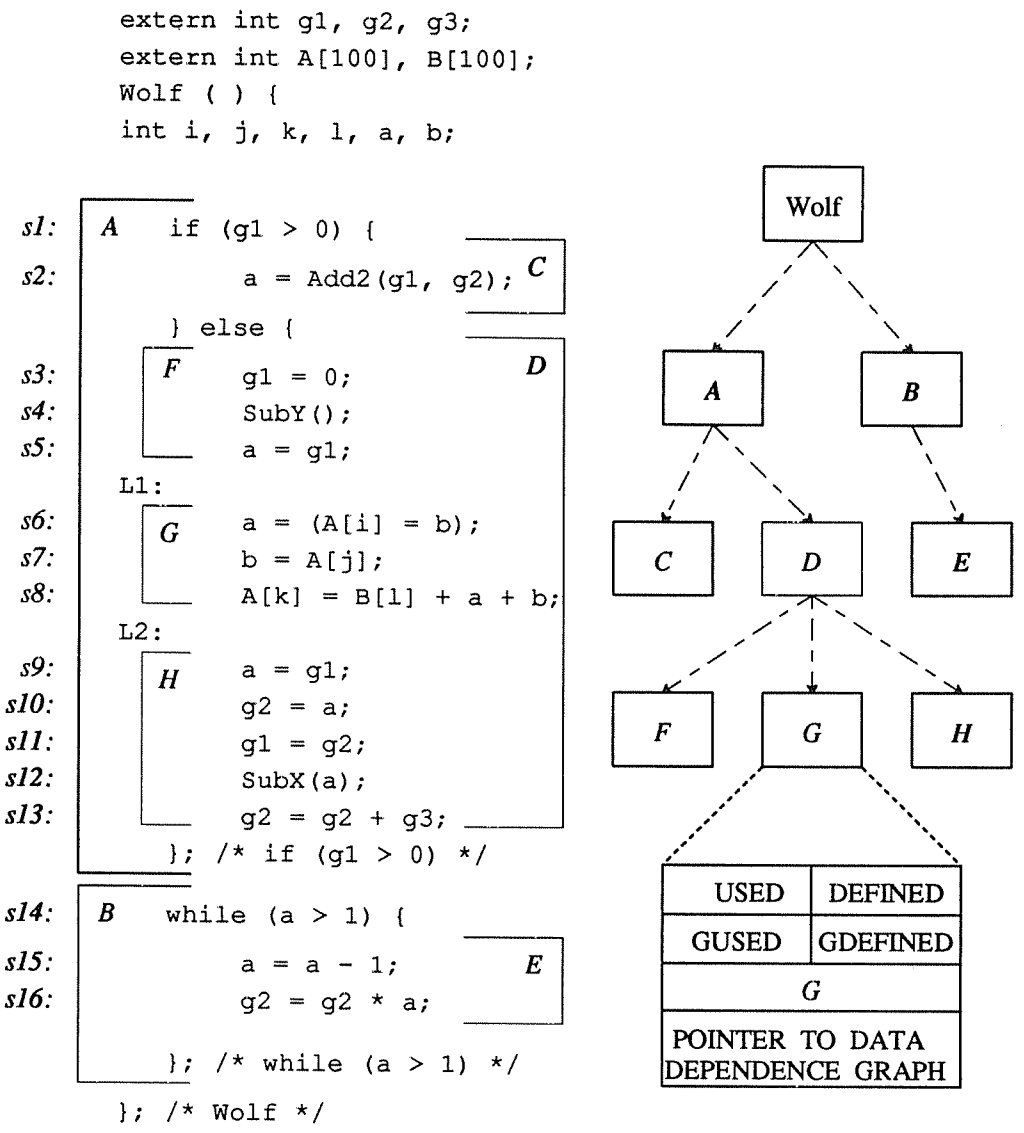


Figure 4.1.
A Sample Static Graph

There are four non-leaf block types needed for C programs. The first type represents conditional statements, such as **if** or **switch**, in which only one of the immediate descendent blocks will be executed. Block A in Figure 4.1 is of this type. During execution, either block C or block D will be executed. All other non-leaf block types execute all their descendent blocks in left-to-right order. 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 4.1 is of this second type.

The third and fourth non-leaf block types are not associated with any statement. The third type acts as a *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 [42]. We give more discussion on the granularity of an e-block in Chapter 5. 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 4.1 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.

Associated with each control block (except dummy blocks) are four sets of variables — *USED*, *DEFINED*, *GUSED*, and *GDEFINED* sets — and a data dependence graph. The USED set is the set of variables whose values might be referenced before they are defined by a statement in this block. The DEFINED set is the set of variables whose values might be defined by statements in this block. The GUSED set is the set of variables that might be used before they are defined in this block *or* in any block in a subroutine called from this block (following the transitive closure of calls). The GDEFINED set is similarly defined. While the USED and DEFINED sets are determined locally by inspecting the statements belonging to a block, the GUSED and GDEFINED sets can only be determined by interprocedural analysis. The GUSED and GDEFINED sets are described in more detail in Section 4.5 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 (USED, DEFINED, GUSED, and GDEFINED) for a summarizing block are the unions of the corresponding sets of all the descendent blocks that constitute the e-block. However, they do not contain variables that cannot be accessed outside of the corresponding e-block. For

example, those four sets for a subroutine do not contain variables that are local to the subroutine. The program database [42] 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 or not. (Sequent C has two additional key words to support parallel programming: **shared** and **private**.)

There are two kinds of summarizing blocks. The first type is a summarizing block representing a subroutine that constitutes an e-block. In this case, the summarizing block is also the root block of the static graph for the subroutine. The second type is a summarizing block representing a group of statements that constitute an e-block. In these two cases, the variables in the USED and DEFINED sets of the summarizing block are the variables that will be written to the log (as described in Chapter 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 those e-blocks that might use or modify a given variable. Chapter 5 discusses how these data structures work together with the log and with incremental tracing.

4.2. Data Dependence Graph (DDG)

Each control block (except for summarizing and dummy blocks) has a data dependence graph that shows the dependences between statements belonging to that block. Figure 4.2 shows a sample control block (basic block) and its data dependence graph for a segment of code. The data dependence graph has four node types: *ENTRY*, *EXIT*, *singular*, and *sub-graph nodes*. The ENTRY and EXIT nodes show the beginning and terminating points of a procedure [28]. The singular node represents an assignment statement, a control predicate in a statement such as an **if** or **case**, 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 internal details of such subroutines. There is one static graph for each subroutine. Each node of the data dependence graph is labeled with either an identifier or an expression, and the statement number. The constant node is labeled with the value and the statement number.

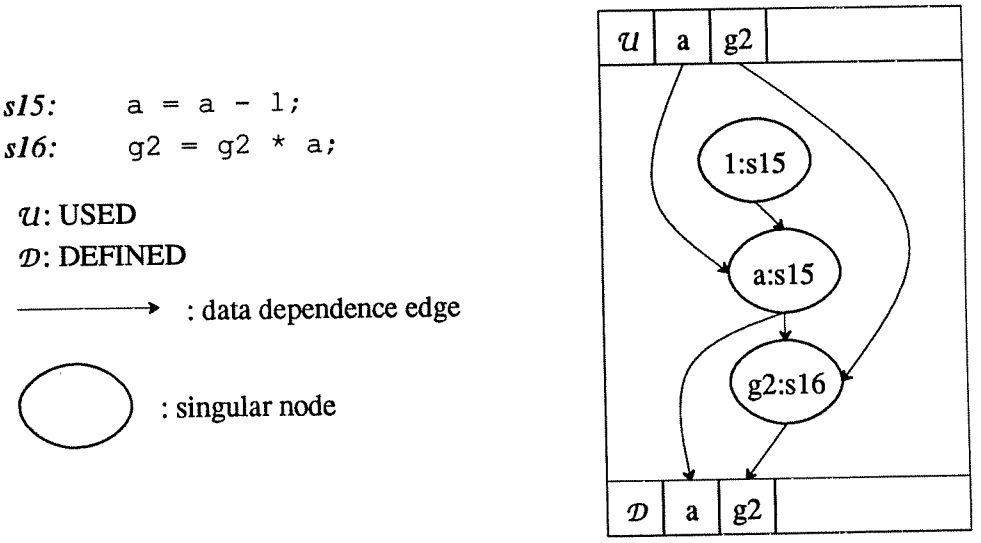


Figure 4.2.
Basic Block and Its Data Dependence Graph (DDG)
(Block E in Figure 4.1)

The data dependence graph has three edge types: *data dependence*, *flow*, and *linking edges*. The data dependence edge between two nodes represents a *true dependence* [6]. Flow edges represent the control flow of the program. The linking edge helps resolve the dependences that can only be determined during 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 4.4.

The top of the control block shows the variables in the USED set of the block and the bottom of the block shows the variables in the DEFINED set of the block. A data dependence edge from the USED 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 DEFINED entry for a variable shows the last statement in the block that modifies the variable. All the nodes in a data dependence graph are fully ordered according to the corresponding statements in the control block, because statements in a control block are sequential. This full ordering of nodes is represented by the flow edges connecting these nodes in a control block, so we can say that a given node is either after or before another node in a control block. We will not explicitly show the flow edges or ENTRY and EXIT nodes in the figures in this chapter.

Inter-block dependences (dependences between two statements belonging to different control blocks) are not resolved during 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 Chapter 5).

4.3. Parameters to Subroutines

To help map between formal parameters and the actual parameters of a subroutine during debugging, we use nodes labeled with "%" for the parameters. We create a node, called a *parameter node*, for each actual parameter passed to a subroutine and label it with "%" and the position of the formal parameter. For example "%1" represents the first parameter, and "%n" represents the n'th parameter. "%0" is used for the return value of a function. Figure 4.3 shows the static graph of control block C in Figure 4.1. The figure shows how actual parameters are mapped to the formal parameters of a called subroutine. The parameter node is a variant of the singular node.

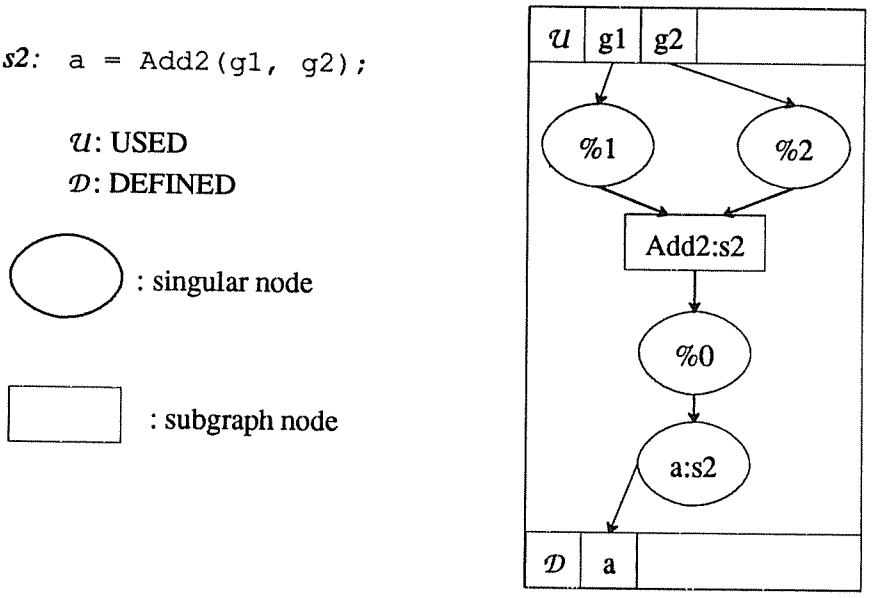


Figure 4.3.
DDG for Parameter Mapping
(Block C in Figure 4.1)

4.4. Arrays and Linking Edges

Array index values are usually unknown during 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 during debugging. We use the linking edge and two new node types, called the *index* and *select nodes*, to quickly determine array reference dependences. The two nodes are also variants of the singular node. We first describe the data dependence graph for a write to an array element, and we then describe the graph for a read from an array element.

To build the data dependence graph for a write to an array element, we first create a singular node for updating an array element. Nodes "A:s6" and "A:s8" in Figure 4.4 are examples of such a node. All the nodes that are created for writes to the same array are linked together by linking edges. We also create an index node for each array index. The index node is labeled with "%" and the index position, similar to a parameter node. There is a data dependence edge coming out of the most recent node that writes the index variable, and going into the index node. We then create data dependence edges out of the index nodes and into the node that writes the array element. Finally, we create data dependence edges out of the nodes representing the variables used as r-values in the assignment to the array, going into the node that writes the array element. For example, node "A:s6" in Figure 4.4, which represents the assignment to "A[i]" in statement *s6*, has three edges going into it: two data dependence edges and one linking edge.

A select node represents a read from an array element. A linking edge comes out of the most recent node that writes into the same array, and into the select node; data dependence edges come out of the index nodes and into the select node. The linking edge used in this case can be regarded as a *potential* data dependence edge. This linking edge and the linking edges connecting the nodes created for writes to the same array are used, during debugging, to identify data dependences that cannot be determined at compile time. The output of a select node represents the value of the array element. Node "b:s7" in Figure 4.4 represents the assignment of "A[j]" to "b" in statement *s7*. There is a data dependence edge out of a select node into the "b:s7" node. There exist two dependence edges into the select node: one linking edge out of the most recent node that writes the same array and one data dependence edge out of the index node. During debugging, we determine the index value of each index node used in either a read or write access.

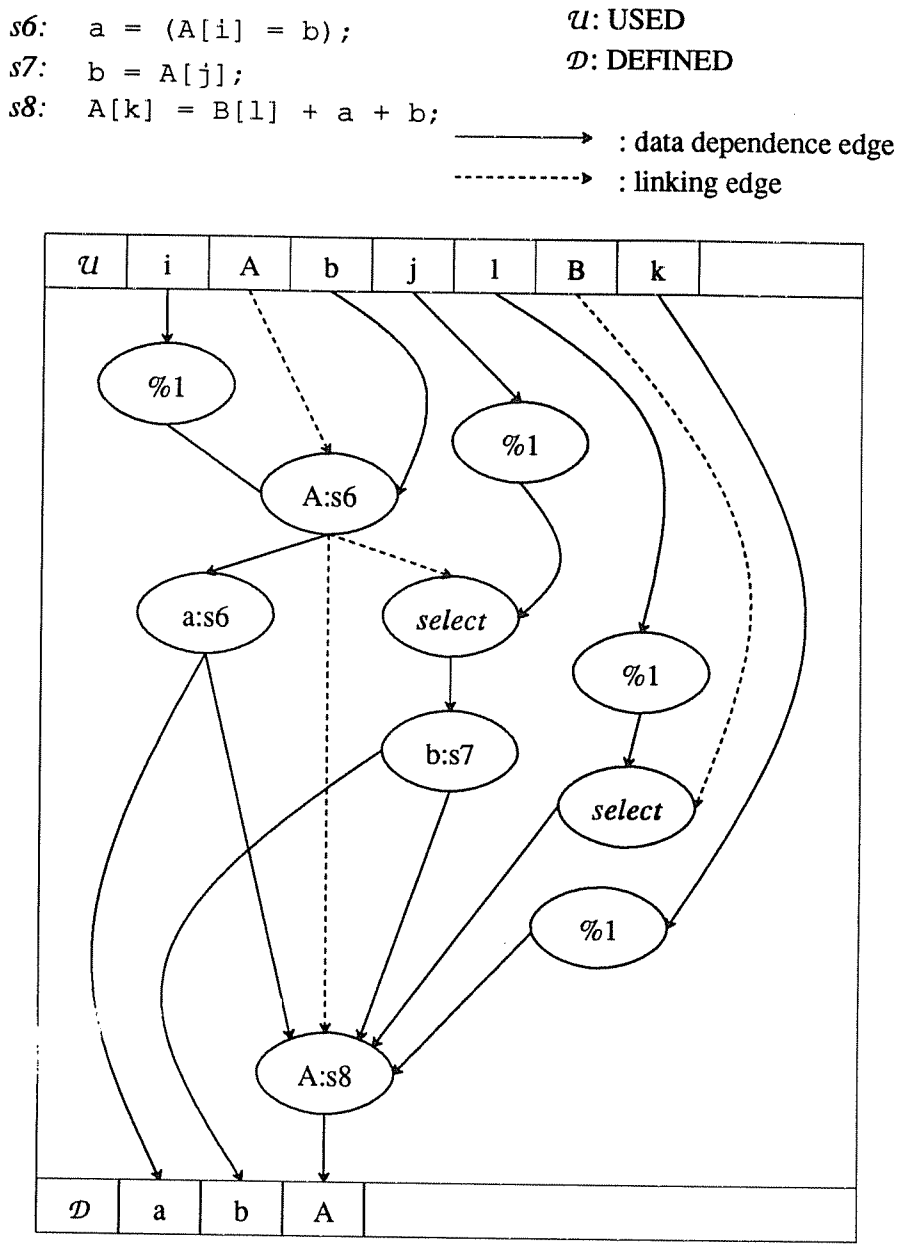


Figure 4.4.
DDG with Arrays and Linking Edges
(Block G in Figure 4.1)

For each select node, we follow the linking edges backward to identify the most recent node that actually wrote the array element of the same index value. We then create a data dependence edge (in the dynamic graph) that comes out of that node for the write to the array, and into the select node. The idea of a select node is similar to the idea used for array-related dependences in [45].

4.5. Interprocedural Analysis and Data Dependence Graphs

In this section, we describe the use of the GUSED and GDEFINED sets. Figure 4.5 shows an example of two possible static graphs for control block F of Figure 4.1. The graph on the left occurs if the

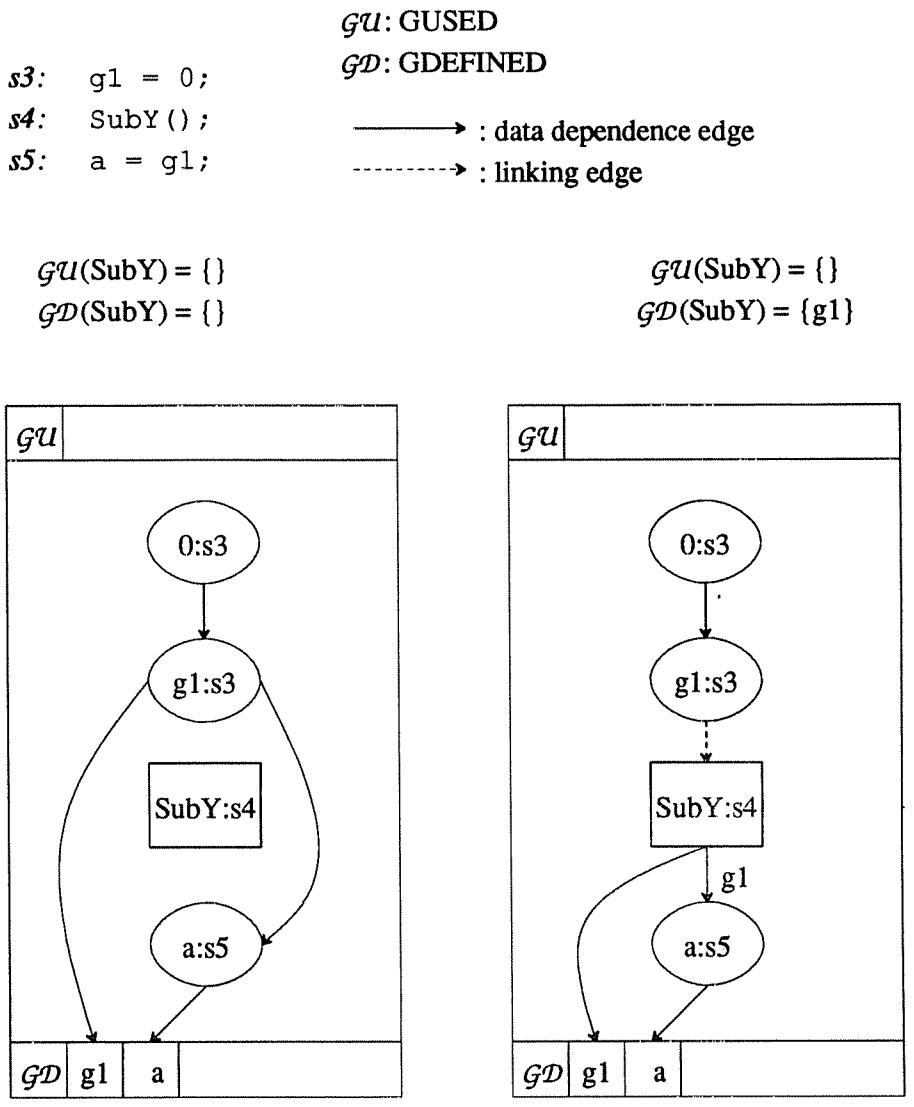


Figure 4.5.
DDGs With Different Summary Information
(Block F in Figure 4.1)

GDEFINED set of "SubY" is empty, and the graph on the right occurs if the GDEFINED set of "SubY" consists of "g1".[1] The GUSED and GDEFINED sets allows us to identify the potential dependences caused by a sub-graph node due to conditional control flows in the subroutine. Such potential dependences, caused by node "SubY:s4", are represented in the graph by two data dependence edges out of the node and a linking edge into the node. The linking edge is needed because GUSED and GDEFINED are sets of variables that might be accessed during the procedure call. When "SubY" does not actually modify "g1", we need to locate the node out of which the actual data dependence edge of "g1" into node "a:s5" should come in the dynamic graph. We need additional bookkeeping information to correctly identify node "g1:s3" in such a case, and the linking edge between "g1:s3" and "SubY:s4" serves that purpose. (Note that the linking edge was similarly used in the previous subsection.)

The USED and DEFINED sets of a control block are purely local and can be determined by examining only the code belonging to that control block. However, the GUSED and GDEFINED sets of a control block require knowledge of what variables might be used or defined by subroutines called inside the control block; we need to do interprocedural analysis such as that done in [11, 12, 18, 29, 44]. The interprocedural summary information consists of two sets for each subroutine:

GDEFINED(P) is the set of non-local variables that might be defined by subroutine *P* itself or by a subroutine (transitively) called from *P*.[2]

GUSED(P) is the set of global variables that might be used before they are defined by subroutine *P* itself or by a subroutine (transitively) called from *P*.

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 4.2–4.4 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

1. Notice that for clarity, edge "g1" out of the sub-graph is labeled with the identifier. Also, recall that flow edges are not shown (but would be present in the actual graph).

2. We overload the use of USED, GUSED, DEFINED, and GDEFINED. For example, we can say GUSED(P) for a subroutine P and GUSED(B) for a control block B.

subroutines whose GUSED or GDEFINED set has changed need to be built again.

Our approach (heuristics) to including 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 coming out of the most recent node that wrote the variable and going into the sub-graph node representing the subroutine call.

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 GUSED 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 GDEFINED 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 GDEFINED set but is not in the GUSED set of the sub-graph node.

Figure 4.6 shows the changes made by interprocedural analysis. There are several differences between the pre-graph and the post-graph. First, the linking edge of "g2" into the sub-graph node in the pre-graph has been changed into a data dependence edge, because "g2" is in GUSED(SubX). Second, the data dependence edge of "g2" out of the sub-graph node into the node "g2:s13" has been disconnected from the sub-graph node and reconnected into the node "g2:s10", because "g2" is not in GDEFINED(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 GDEFINED entry for "g3" and the other into node "g2:s13". These edges are added because "g3" turned out to be in GDEFINED(SubX). Last, the data dependence edge from the USED entry for "g3" into node "g2:s13" is deleted. The linking edge out of "g1:s11" into the sub-graph node is intact because "g1" is in GDEFINED(SubX) but is not in GUSED(SubX).

4.6. Parameter Aliases

In this section, we describe how to extend our methods to handle aliases resulting from reference parameters in languages like Pascal or FORTRAN. Our approach is to analyze the program to identify the potential aliases for a given variable [11, 12, 27, 38, 50].

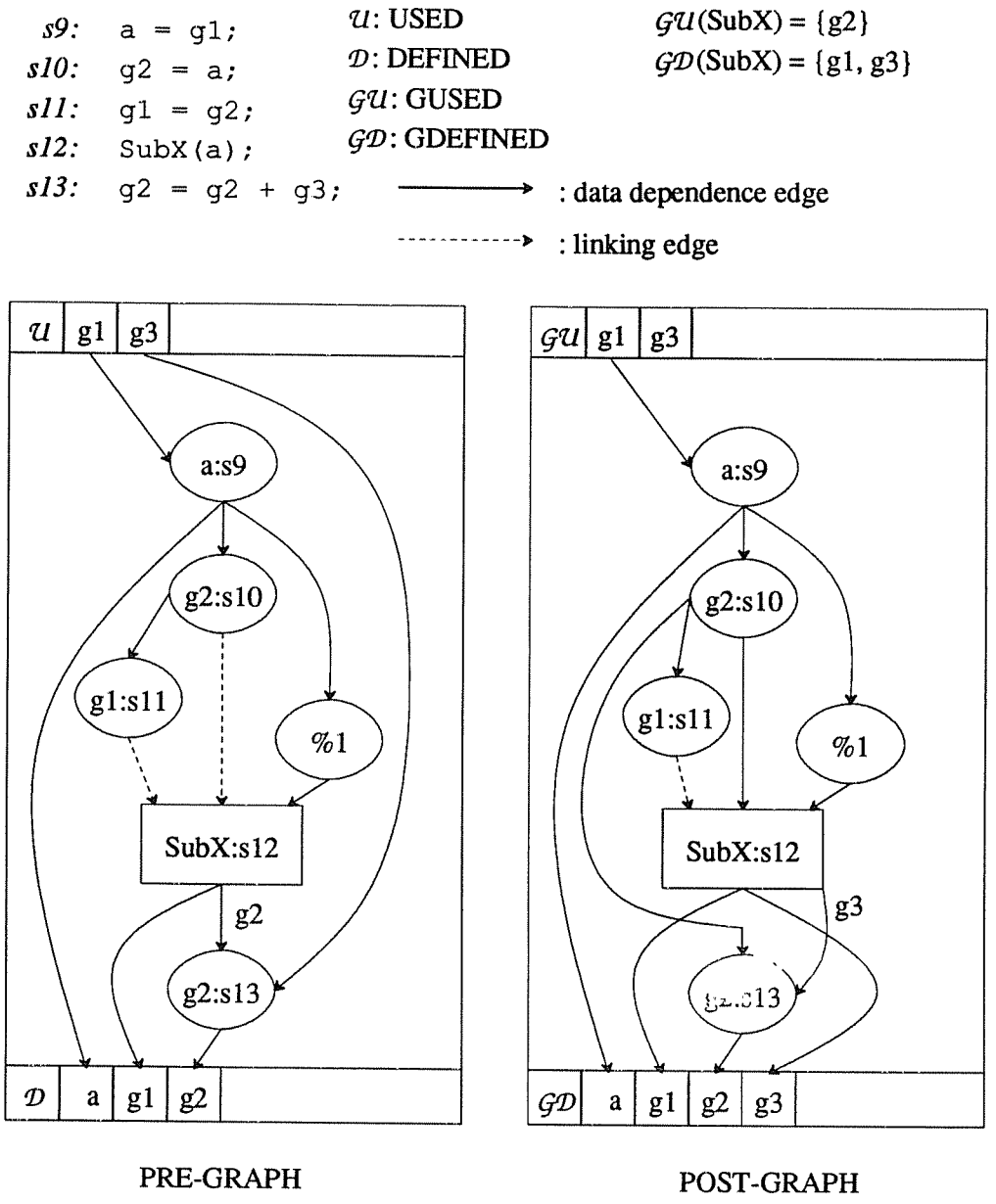


Figure 4.6.
DDG and Summary Information
(Block H in Figure 4.1)

In the static data dependence graphs, we link together (with linking edges) all of the nodes for writes to variables that are potential aliases. For a node that reads one of these variables, we create a data dependence edge out of the most recent node that writes one of these variables.

Actual dependences for aliases are discovered after program execution. The prelog for a subroutine contains additional information for parameters that are potential aliases. For each such parameter, the prelog also records the address of the actual parameter. Potential aliases whose addresses are the same during execution are indeed aliases. The linking edges in the static graph and the additional prelog information allow us to correctly identify dependences in the presence of parameter aliases.

4.7. Building the Data Dependence Graphs

The data dependence graph is built in two steps. First, the pre-graph for each control block is built, ignoring interprocedural dependences. Then, a post-graph is built from the pre-graph, incorporating information from the interprocedural analysis. Algorithms to build the pre-graph and post-graph are given in the Appendix.

4.8. Model of Dependence Analysis for Complex Objects

Program dependence graphs have been used in many areas of compiler technology. The purpose of using these graphs is to identify potential dependences in a program. When these graphs show the dependence between variables in a program, they typically treat the variables as whole objects. That is, a read or write to any part of the variable is assumed to have touched the entire variable. For scalar variables, this is a natural assumption; for more complex variables, such as shown in Figure 4.7, this assumption may be unnecessarily conservative. In this section, we describe the techniques we have adopted in PPD for dependence analysis of such complex objects. We do not currently include dynamic (heap allocated) data structures.

We first describe the trade-offs between the compile-time and debug-time costs for the overall performance of the debugger as related to the precision of static information. Such precision can be described as a spectrum in terms of the size of the memory object to which a dependence refers. We then describe the point in that spectrum adopted in the current implementation of PPD.

The current version of PPD does not fully use the structural information of complex objects, such as the sub-field of a structure array accessed with a variable index. The next version of PPD will extend the analysis in the current implementation to better utilize such structural information of complex objects. Sections 4.7.3 and 4.7.4 describe our model for complex objects to be used in future extensions of PPD.

```
int i, j, k;

typedef struct {
   int f, g;
} sType0;


struct {
   struct {
      sType0 A[100];
      int e;
   } s1;
} U;
```

```
sub0 ()
{
   U.s1.e = 0;
   U.s1.A[i].f = U.s1.e;
   U.s1.A[i] = U.s1.A[j];
}; /* sub0 */
```

Figure 4.7.
Examples of References to Complex Objects

4.8.1. Trade-Offs

PPD is an interactive debugger that builds the dynamic graph in order to show the dependences requested by the user. In general, building the dynamic graph during debugging will be less costly if more precise static information is computed at compile time. However, we should not unduly sacrifice compile-time efficiency for debug-time efficiency; we need to weigh the trade-offs between the compile-time and debug-time costs for the overall performance of the debugger.

The precision of the static information can be measured by the size of the memory object to which a dependence refers. This can be described as a spectrum. At one extreme of the spectrum is the case where you can only determine that a dependence exists, but not where the dependence lies in the address space. This could be the case for a freely-bound pointer. At the other end of the spectrum is the case where a particular memory cell could be determined.

There are several intermediate points on the spectrum that are of interest. One interesting point is the case where we can determine the structural subcomponent within a complex object. For example, we can determine the particular field of a structure or the particular row (or sub-array) of an array. In this case, we

are using knowledge about the construction of the object. This is the point we have adopted in PPD.

Further down along the spectrum, we might be able to determine something more precise than a structural subcomponent. For example, we might determine a particular range of the elements within an array row. As the static information increases in precision, we can potentially reduce the cost of the debug-time algorithms. However, the cost of semantic analysis increases with the precision of the static information supplied. Ultimately, the actual reduction in cost will be determined by experience with the techniques on a variety of real programs. This is the point we would like to investigate as a future research.

4.8.2. Model for Complex Objects

Determining the USED and DEFINED sets of a statement is the first step in determining the dependence relationship between statements. Determining such sets is a well-understood process for scalar variables. However, when an array element with a variable index is accessed, one possible approach is to assume that all the array elements are potentially modified even if only one array element is actually accessed [2]. This yields a USED or DEFINED set that is bigger in size and less precise than that which could be obtained if we knew exactly which array element is modified. By applying more sophisticated dependence decision algorithms for arrays [10, 61], we can compute smaller and more precise USED and DEFINED sets. However, the problem becomes more complex for variables with arbitrary nestings of arrays and structures.[3]

In the current implementation of PPD, the USED and DEFINED sets of a statement consist of object identifiers with three fields as follows: <*object-id, offset, size*>. For a scalar variable, the offset is zero and the size is the size of the variable. However, for a reference to a portion of a complex object, such as `U.s1.A[i].f` in Figure 4.7, the object-id is the symbol identifier of the whole object (e.g., `U`), the offset is the offset of the referenced portion of the complex object from the beginning of the address space occupied by the whole object, and the size is the size of the address space that can be potentially occupied by the referenced portion of the complex object. For example, in Figure 4.7, the PPD identifier of complex object `U.s1.e` is

3. We use C Programming Language terminology and examples in this section, since this is the current target language of PPD.

<U, offset of `s1` in `U` + offset of `e` in `s1`, size of `e`>,

the identifier of complex object `U.s1.A[j]` for an integer variable `j` is

<U, offset of `s1` in `U` + offset of `A` in `s1`, size of `A`>,

and the identifier of complex object `U.s1.A[3].f` is

<U, offset of `s1` in `U` + offset of `A[3]` in `s1` + offset of `f` in `A[3]`,

size of `f`>.

We regard the identifier of `U.s1.A[i].f` in the same way as `U.s1.A[j]`; we are not currently using the additional information about the sub-field of an array element accessed with a variable index. In the static graph of subroutine "sub0" in Figure 4.7, there will be a linking edge from the node for the write to `U.s1.A[i].f` to the node for the write to `U.s1.A[j]`. The actual part of the address space occupied by the two objects will be determined during debugging using run-time information. The next version of PPD will extend the analysis in the current implementation of PPD to utilize such additional information. We describe this extension in the remainder of this section.

Our goal is to compute relatively small and precise USED and DEFINED sets for complex objects. We first describe a model for representing accesses to multi-dimensional arrays. Then we describe the representation for nested structures. Finally, we describe a unified representation for arrays and structures with arbitrary nesting. We are currently working on extending these methods to handle unions as well. Also, the techniques described in this section are more general than what is needed for the C language.

Multi-Dimensional Arrays

We represent each array element that is read or written in a statement as an *array object*. Each array object is a linked list of nodes, with one node for each array dimension. Each node consists of two fields: ***label*** and ***next***. There are two types of nodes: *constant* and *wild card*. Constant nodes, representing constant indices, are each labeled with the value of the constant index. Wild card nodes, representing variable indices, are each labeled with "*". The first node of each array object is labeled with the array identifier. Figure 4.8 shows an example of two subroutines and the four array objects representing the array elements accessed in the subroutines. Object O_1 represents the array element write-accessed in statement S_1, and so on.

We say there is an *exact match* between two array objects O_i and O_j if

(1) neither array object has a wild card node, and

(2) $N_{i,k} = N_{j,k}$ for $1 \le k \le$ **minlen**(O_i, O_j), where $N_{i,k}$ is the label of the k'th node of O_i, and **minlen**(O_i, O_j) is the length of the list of the shorter of the two objects O_i and O_j.[4]

We say there is a *potential match* between two array objects O_i and O_j if,

for $1 \le k \le$ **minlen**(O_i, O_j),

(1) any of the $N_{i,k}$ and $N_{j,k}$ is a wild card node, or

(2) $N_{i,k} = N_{j,k}$.

The set of objects that satisfy the potential-match relationship is an improper superset of the set of objects that satisfy the exact-match relationship. Two array objects do not match at all if there is no potential match between them.

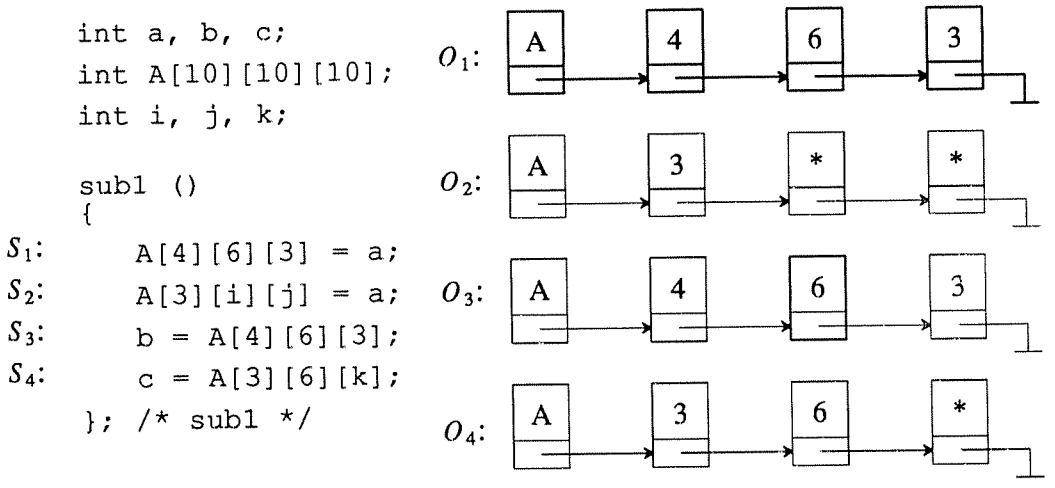


Figure 4.8.
Model for Multi-Dimensional Array

4. In C, the lengths of two array objects must actually be the same to match. However, in programming languages like PASCAL, where a whole array can be assigned at once, the lengths can indeed be different.

By using this representation, we can decide if there exists any exact or potential data dependence between two statements. In Figure 4.8, there is an exact data dependence of statement S_3 upon statement S_1, because object O_3 has an exact match with object O_1. There can be no match between O_1 and O_4, however. The compile-time process thus narrows the set of statements on which a given statement can depend, which will speed up the debug-time process of identifying the actual dependence.

Structures

The representation for nested structures is similar to the one that we use for array objects; each structure is represented by a linked list called a *structure object*. Each node of a structure object is labeled with a field identifier. However, there are no wild card nodes for structure objects. In Figure 4.9, O_1 and O_3 do not match any other objects in the figure. However, O_{2a} and O_4 match exactly with each other.

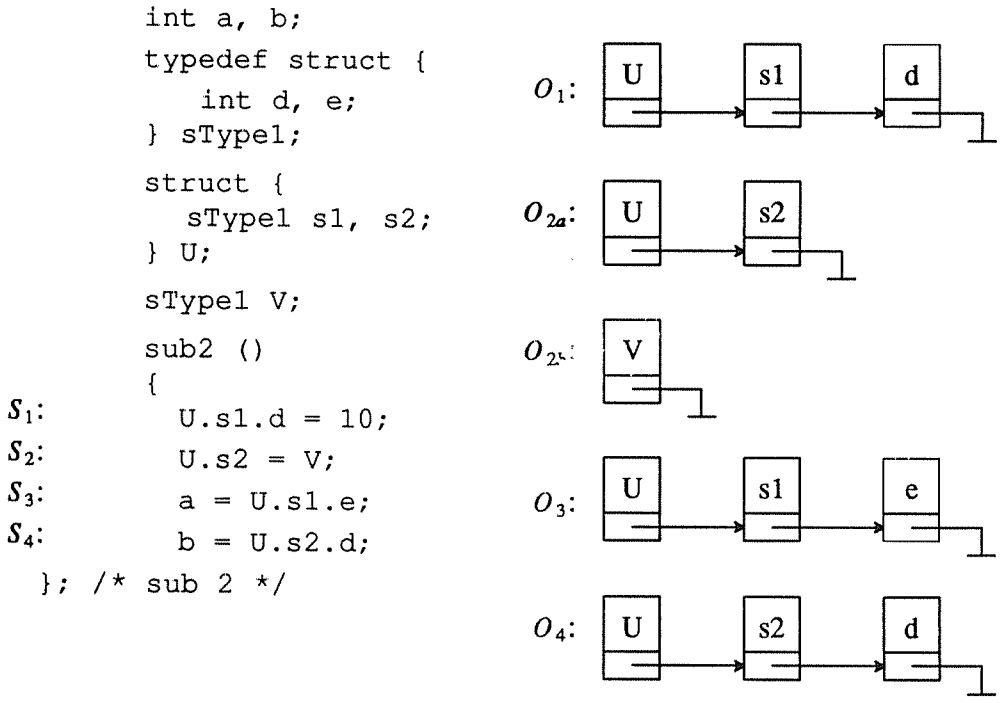


Figure 4.9.
Model for Structures

Arrays and Structures with Arbitrary Nesting

The array and structure objects that we have introduced are special cases of our representation for complex objects that are arbitrary compositions of arrays and structures. Figure 4.10 shows examples of such objects. In building the static graph for these procedures, we see that S_3 can depend on S_2 or S_1 depending on the values of i, j, and k during execution. However, S_4 cannot depend on S_1 or S_2.

```
int a, b, i, j, k;
struct {
 struct {
    sType1 s1[10];
    int a;
 } s3;
} B[10];
```

```
   sub4 ()
   {
S1:  B[i].s3.s1[j].d = 10;
S2:  B[4].s3.s1[k].d = 5;
   }; /* sub4 */

   sub3 ()
   {
     sub4 ();
S3:  a = B[4].s3.s1[j].d;
S4:  b = B[5].s3.a;

   }; /* sub3 */
```

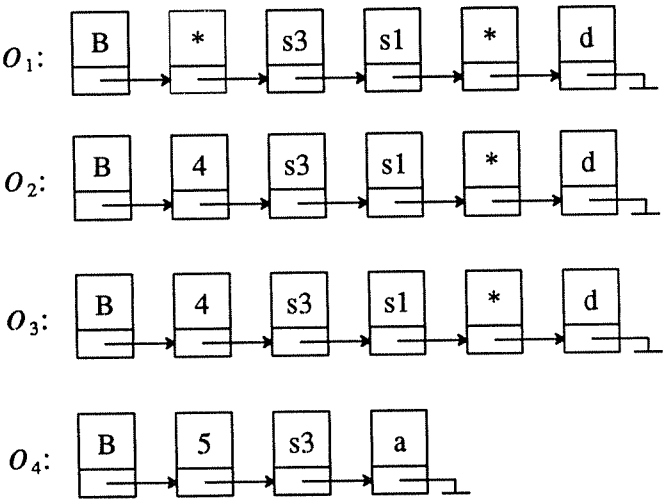


Figure 4.10.
Unified Model for Arrays and Structures

Algorithm 4.1 shows a function that determines whether two complex objects match, or potentially match. The time complexity of this function is $O(n)$ where n is the length of the shorter of the two objects. The length of an object corresponds to the structure nesting level and array dimensions; we expect n to be small in practice.

4.8.3. USED and DEFINED Sets for Complex Objects

Once we compute the complex objects that might be accessed by each statement, we can compute the USED and DEFINED sets of a code segment. For example, the DEFINED set of "sub4" in Figure 4.10 consists of objects O_1 and O_2. When the user wants to know the statement upon which S_3 depends, we must look in "sub4" because the DEFINED set of "sub4" contains objects that potentially match O_3 used in S_3. However, upon close examination, we find that the DEFINED set of "sub4" consisting of O_1 and O_2 has redundant information; a DEFINED set with only O_1 is enough to identify "sub4" as the

```
MatchObject (obj1, obj2: complex object)
  begin
      node1 ← first node of obj1;
      node2 ← first node of obj2;
      wildCard ← FALSE;
      while neither of node1 and node2 is NULL
          if any of the two nodes is a wild card node then
              wildCard ← TRUE;
          elseif the two nodes have different labels then
              return (NO_MATCH)
          elseif any of the two nodes is a UNION node then
              /* future work */
              return (POTENTIAL_MATCH);
          end if;
          node1 ← node1.next;
          node2 ← node2.next;
      end while;
      if (wildCard) then
          return (POTENTIAL_MATCH)
      else
          return (EXACT_MATCH);
      end if;
  end MatchObject;
```

Algorithm 4.1 An Algorithm to Test Object Matching

subroutine that contains objects potentially matching O_3.

In an effort to minimize such redundant information in the USED and DEFINED sets, we define a relation between two objects as follows.

Definition 1: Between two complex objects O_i and O_j, we say O_i is a *super object* of O_j if

---

```
GetSuperObj (obj1, obj2: complex object)
  begin
      node1 ← first node of obj1;
      node2 ← first node of obj2;
      supObj ← NULL;
      while neither of node1 and node2 is NULL
          if one of the two nodes is a wild card node then
              if node1 is a wild card node and (supObj = obj2) then
                  return (NULL)
              elseif node2 is a wild card node and (supObj = obj1) then
                  return (NULL)
              else
                  supObj ← the object whose node is a wild card node;
              end if
          elseif the two nodes have different labels then
              return (NULL)
          elseif any of the two nodes is a UNION node then
              /* future work */
              return (NULL);
          end if;
          node1 ← node1.next;
          node2 ← node2.next;
      end while;
      if node1 is NULL and (superObj = obj2) then
          return (NULL)
      elseif node2 is NULL and (superObj = obj1) then
          return (NULL)
      end if ;
      return (the object whose corresponding node is NULL);
          /* return any node if both are NULL */
  end GetSuperObj;
```

Algorithm 4.2 An Algorithm to Compute a Super Object

---

1) there is a potential match between O_i and O_j,

2) $\mathbf{len}(O_i) \leq \mathbf{len}(O_j)$, and

3) if $N_{j,k}$ is a wild node, then so is $N_{i,k}$ for all $k \leq \mathbf{len}(O_i)$.

For example, O_1 in Figure 4.10 is a super object of O_2 in the same figure. When a USED or DEFINED set contains an object and its super object, only the super object is retained in the set. Thus, the DEFINED set of "sub4" only needs to contain O_1.

Algorithm 4.2 shows a function that, given two objects, computes which one is a super object of the other (if either) and returns the super object. It returns NULL if neither one is a super object of the other. The time complexity of Algorithm 4.2 is the same as that of Algorithm 4.1 (i.e., $O(n)$).

Chapter 5

Dynamic Graphs and Incremental Tracing

The dynamic program dependence graphs show the causal relations between program events and states at execution. We build the dynamic graphs at debug time, using the static graphs generated at compile time and the detailed traces generated by incremental tracing at debug time. In this chapter, we first describe the dynamic graphs and then we describe incremental tracing.

5.1. Dynamic Program Dependence Graph (Dynamic Graph)

The dynamic graph shows the causal relations between program events and states during execution. In this section, we will use subroutine "Wolf" from Figure 4.1 to describe the dynamic graph and to illustrate how such a dynamic graph is built from the static graphs and fine traces.

The dynamic program dependence graph of a program P, denoted as G_p, is a directed graph consisting of four types of nodes and four types of edges. The four node types are the *ENTRY node*, the *EXIT node*, the *singular node*, and the *sub-graph node*. Each node represents a program event (execution of a program component). The four edge types are the *flow edge*, the *data dependence edge*, the *branch dependence edge*, and the *synchronization edge*.

Each node contains either an identifier or a predicate expression and a statement number. The ENTRY node in G_p is the point at which control is first transferred into the scope of G_p, and the EXIT node is the point at which control is transferred out of the scope of G_p. A singular node corresponds to the execution of an assignment statement, a control predicate such as an **if** statement, or branch statements such as **goto** in the program. When a singular node corresponds to an assignment statement, it is associated with the assigned value. When a singular node corresponds to a control predicate, it is associated with the value of the control predicate.

The sub-graph node, consisting of nodes and edges, is a way of encapsulating the execution details of several statements. These statements usually correspond to functions or subroutines supplied by the user or the system. When the user wants to know the execution details corresponding to the sub-graph node, the

debugger presents the user with the details of the sub-graph node by incrementally generating the fine traces for the execution instance corresponding to that sub-graph node.

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 (of the event represented by n_j) 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)* [23] is concerned about the *potential* program control flow in a program. More details on branch dependences and their relationship to control dependences are presented in Section 5.3.

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

5.2. 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 5.1, and the graphs for the remaining blocks were given back in Figures 4.2–4.6. 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.

Figure 5.2 shows the resulting dynamic graph of this execution. (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

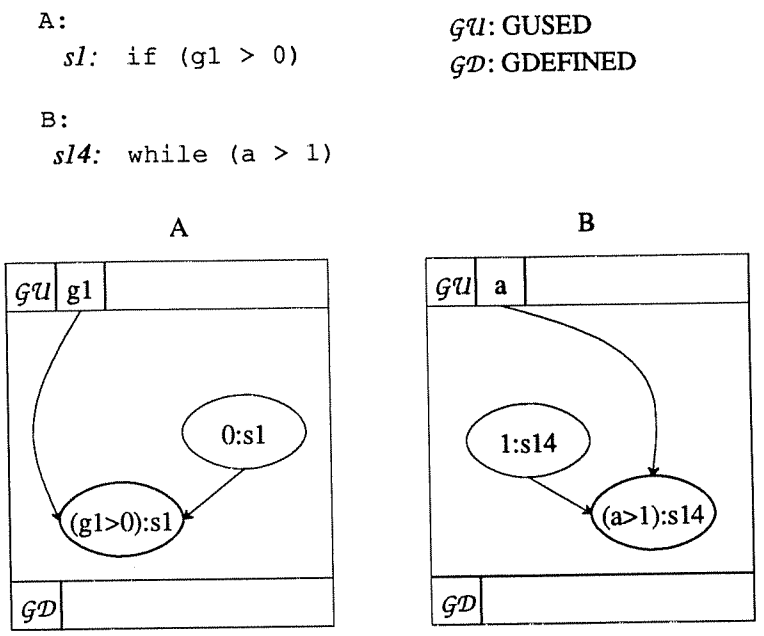


Figure 5.1.
Blocks A and B of Figure 4.1

dependence edges, we connect each variable in the GUSED set to the most recent GDEFINED 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 during 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 subgraph node bypass the sub-graph node in the dynamic graph. These data dependence edges will be now coming from the node 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.

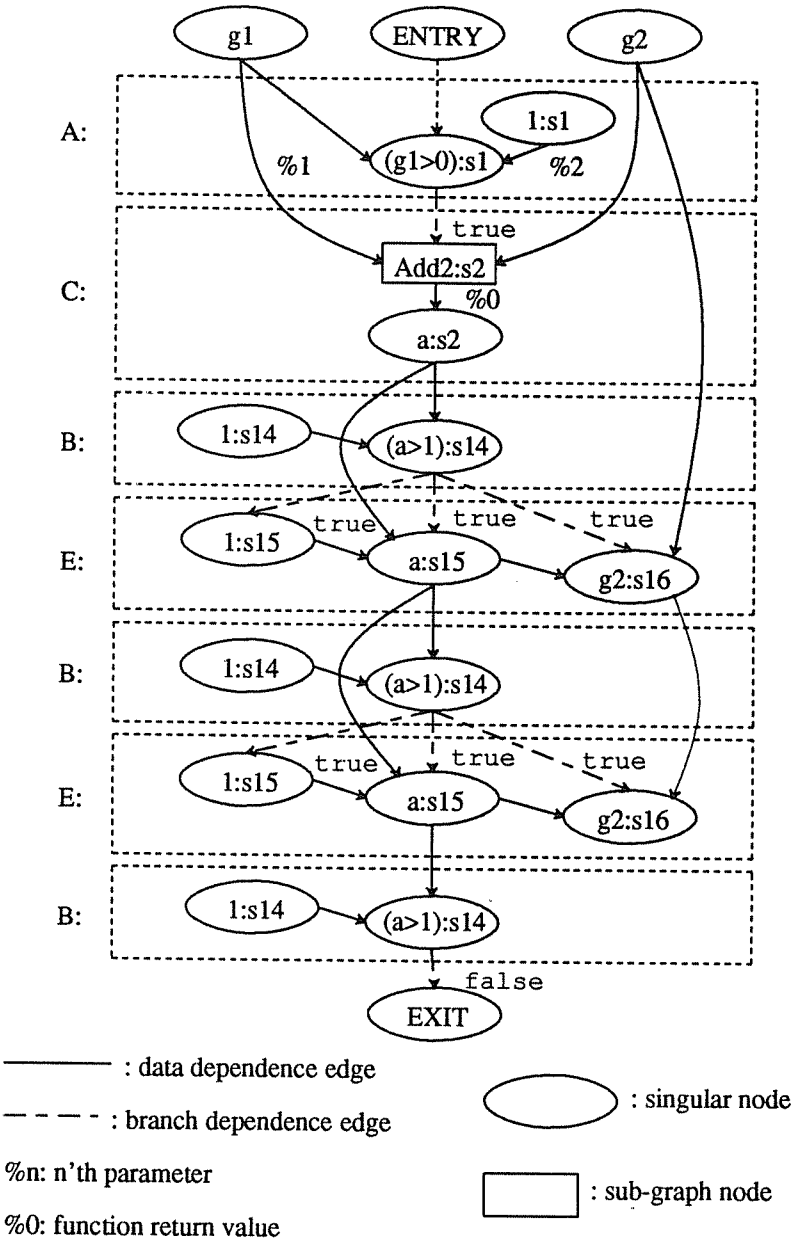


Figure 5.2.
An Example Dynamic Graph Showing Control Blocks

For example in the post-graph of Figure 4.6, if the execution of "SubX" actually wrote "g1", the linking edge coming out of node "g1:s11" 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 GDEFINED entry for "g1".

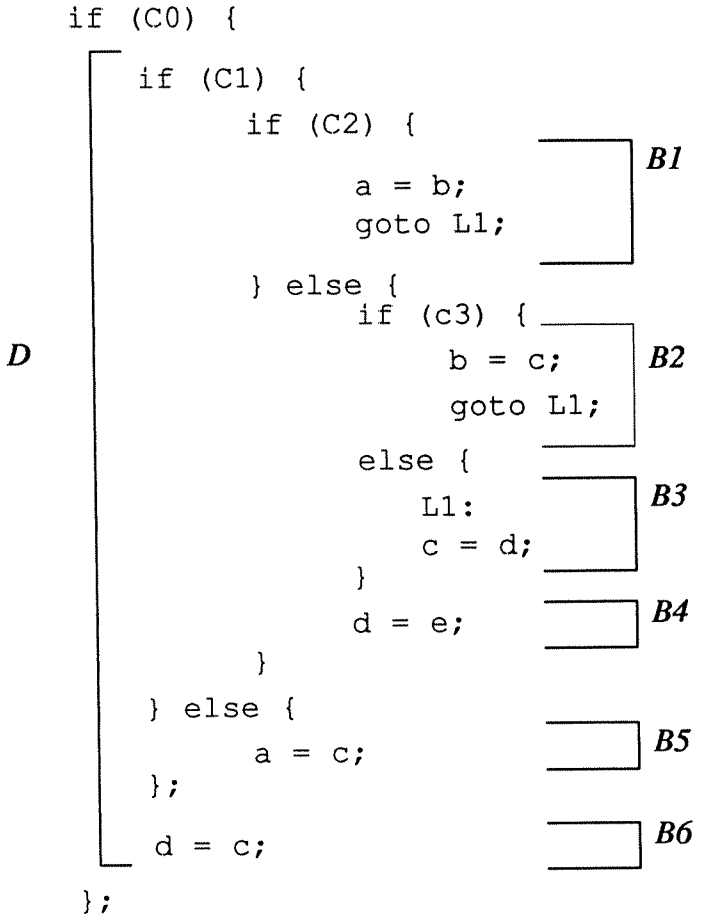


Figure 5.3.
A Sample Program Segment with goto Statements

The data dependence edge coming out of "SubX:s12" and going into the GDEFINED entry for "g1" would also be deleted in this case.

5.3. Dynamic Graph and goto Statements

Goto statements affect the process of building the dynamic *branch dependence graph* (the part of dynamic graph that does not show data dependence edges). We will use the example program segment in Figure 5.3 to first show how **goto** statements affect the dynamic branch dependence graph. We then compare this form of the graph to one based on control dependences used in the PDG by Ferrante, et al [23].

There are three ways program control can flow from a basic block (*source block*) to another basic block (*target block*) during the execution of a program written in a block-structured programming language like C; execution of a conditional branch statement such as an **if** or **case** statement, execution of an un-conditional branch statement such as a **goto** statement, execution of the last statement of a basic block that is not a branch statement. In the first and second cases, we always construct a branch dependence edge from the node representing the branch statement to the target block.

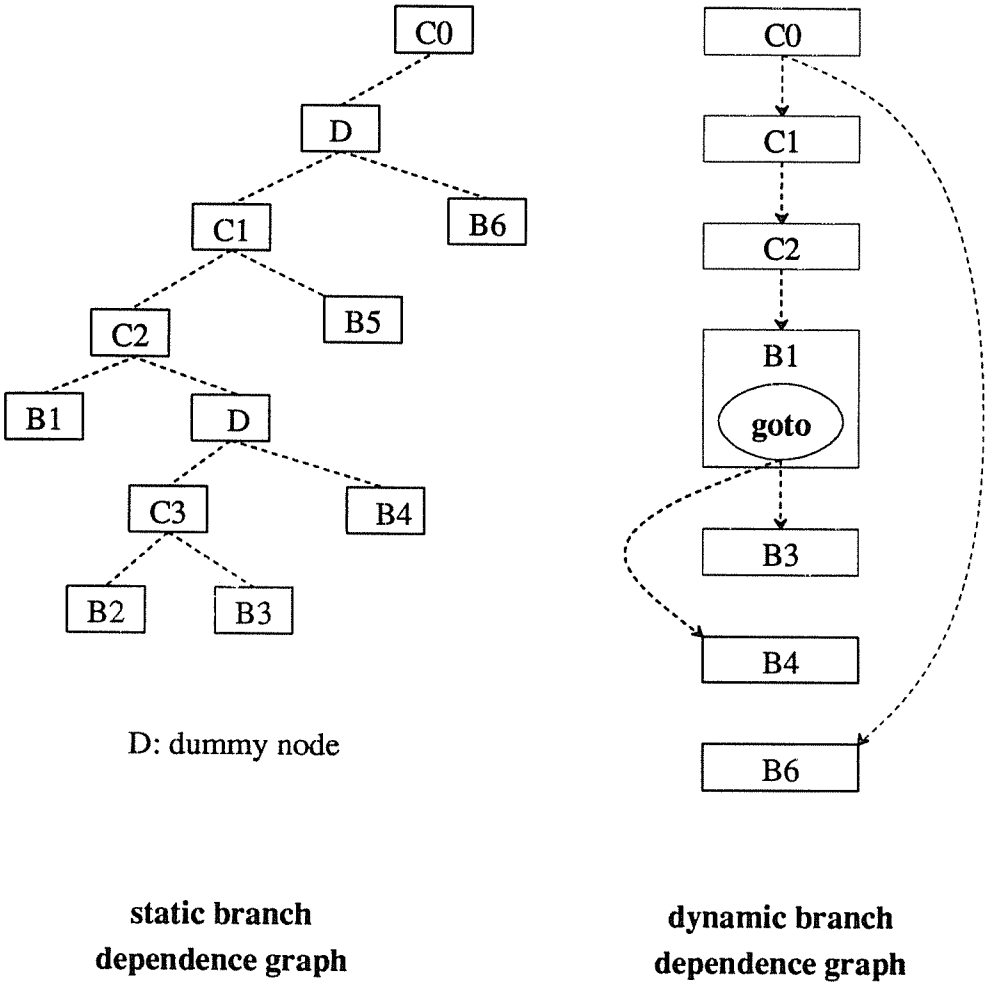


Figure 5.4.
Branch Dependence Graph in PPD

In the third case, we construct a branch dependence edge depending upon how control flow arrived at the target block. If program control would have not reached the target block without **goto** statements, we construct a branch dependence edge from the latest **goto** statement to the target block. Otherwise, we locate the most recent conditional statement that caused control flow to reach the target block. We will describe how we construct branch dependence edges for the third case using Figure 5.4. Figure 5.4 shows the static branch dependence graph and the dynamic branch dependence graph for an execution instance of the program segment in Figure 5.3.

We assume, for this execution instance, that the execution sequence of blocks is: C0, C1, C2, B1, B3, B4, B6. Note that B3 has a *dynamic* branch dependence edge from the **goto** statement of B1, while B3 has a *static* branch dependence edge from C3; the dynamic branch dependence graph shows that control flow reached B3 through C2 and B1, not C3, in this execution instance.

In this execution, control flow would not have reached B4 without the **goto** statement, so B4 has a dynamic branch dependence edge from the **goto** statement in B1. Control flow would reach B6 without the **goto** statements, so B6 has a dynamic branch dependence edge from C0, not from the **goto** statement of B1.

One simple test to determine whether a target block of the third case belongs to the category of B4 or B6 is as follows. We first locate the parent block of the target block. We then check if it lies on the path of control blocks that have executed so far. If the parent block is on this path, the target block belongs to the category of B6; otherwise, it belongs to that of B4.

The PDG by Ferrante, et al uses a somewhat different representation of dependences for **goto** statments. Figure 5.5 shows the (static) control dependence subgraph for PDG (of the program segment in Figure 5.3) and the corresponding (possible) dynamic control dependence subgraph.[1] Note that B3 is a *post-dominator* of C2 in the program segment given in Figure 5.3. PDG represents this relationship by showing a control dependence for B3 from C1, not C2. The (dynamic) branch dependence graph in Figure 5.4 (PPD) is better suited to show the causality of program events in an execution instance of a program than the graph in Figure 5.5 (PDG), because it shows how control actually flowed to a given point during

1. They do not actually build a dynamic graph.

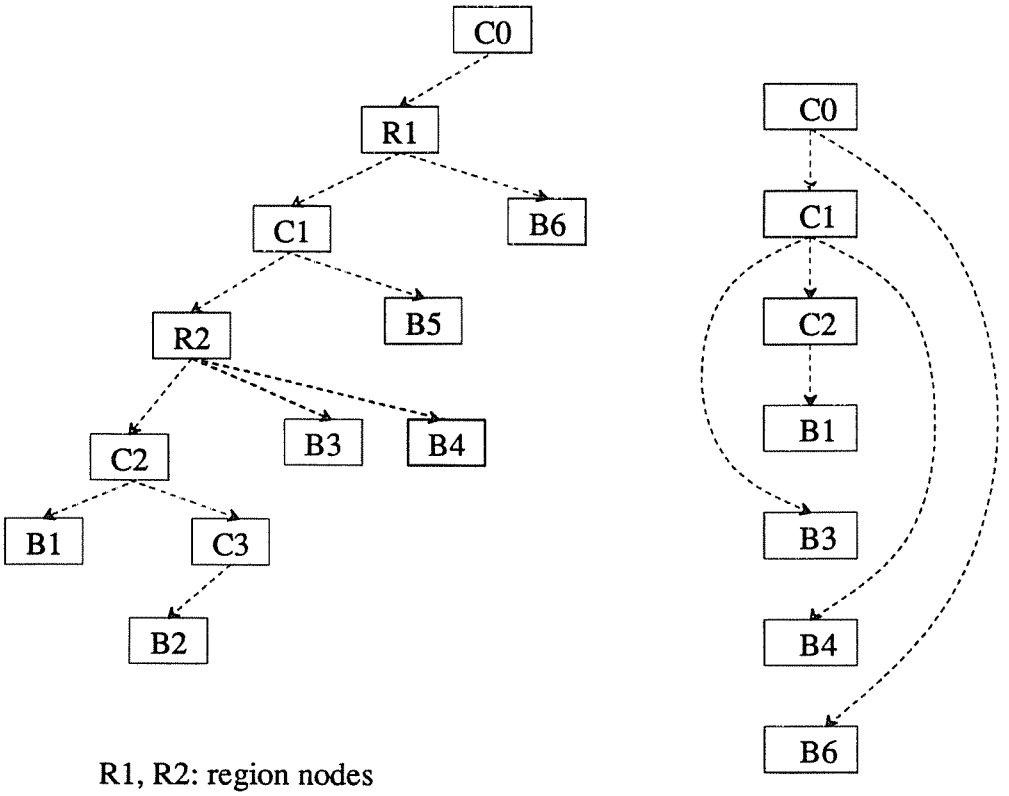


Figure 5.7.
Control Dependence Subgraph in PDG

execution. However, the dynamic control dependence subgraph in Figure 5.5, combined with that in Figure 5.4 (PPD), might be informative to show the possible behavior of the program in some other execution instances.

5.4. Incremental Tracing

The principal approach we take for efficient debugging of parallel programs without re-execution is to generate a small log during the execution phase, and then to generate the actual traces necessary to do flowback analysis during the debugging phase. The log generated during execution allows for incremental tracing during debugging; it allows us to only generate traces needed to show the relevant dynamic

dependences. During debugging, the log also allows for the restoration of the program state to previous points of program execution. In this section, we describe incremental tracing. We first assume the reproducible execution behavior of the program to be debugged, and in Chapter 6, we describe how to make these operations possible for parallel programs that lack reproducibility.

5.4.1. Emulation Blocks and Logs

As described in Chapter 3, among the log entries generated during execution phase are prelogs and postlogs. We call the activities to produce such log entries *prelogging* and *postlogging*.

The object code generated by the Compiler/Linker during the preparation phase contains code to generate prelogs and postlogs. By using static analysis, we divide the program into numerous segments of code called *emulation blocks (e-blocks)*. Each e-block starts with code to generate a prelog and ends with code to generate a postlog. The prelog consists of the values of the variables that may be read-accessed during the execution of the e-block. The postlog consists of the values of the variables that may be write-accessed during the execution of the e-block. 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 emulation block.

The i'th prelog and the corresponding postlog generated during program execution are called *prelog(i)* and *postlog(i)*, respectively. The time interval between the prelog and postlog of an e-block is

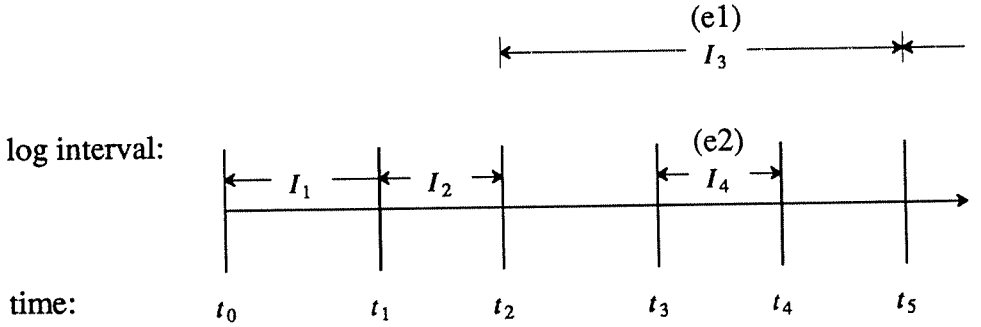


Figure 5.6.
Logging Points and Log Intervals

called the *log interval* and is denoted as I_i for the log interval starting with prelog(i). The log entries in log interval I_i are denoted as *LOG(i)*. The set of events in e-block "e" is denoted as E_e. Programs often contain loops, so a given e-block of a program may have several corresponding log intervals during execution. Figure 5.6 shows examples of logging points and log intervals.

USED(e) is the set of variables that may be read-accessed by E_e. *DEFINED(e)* is the set of variables that may be write-accessed by E_e. The prelog(i) generated by e-block "e" consists of the values of the variables belonging to USED(e) at the beginning of I_i, and the postlog(i) consists of the values of the variables belonging to DEFINED(e) at the end of I_i. (The reason why log entries need to include only local information will become clear in the next section.) To reproduce the same program behavior for interval I_i during the debugging phase, we use the program code corresponding to I_i (e-block "e"), the log entries generated during I_i (LOG(i)), and the same input as originally fed to the program during that log interval. USED(e) and DEFINED(e) are obtained while data dependence graph of E_e is built (as described in Chapter 4).

5.4.2. Nesting of Log Intervals

Log intervals are *nested* when one subroutine calls another. For example, in Figure 5.6 we assume that the e-block corresponding to log interval I_3 is made up of a subroutine named "Sub1" (and we will call this e-block "e1"). We also assume that the e-block corresponding to I_4 is made up of a subroutine named "Sub2" (and we will call this e-block "e2"). 'Sub2" is called from within "Sub1". Prelog(3) and postlog(3) are recorded at the start and end of I_3, respectively; prelog(4) and postlog(4) are recorded 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 the user is interested in the events between t_4 and t_5, the system retrieves prelog(3) to set up the program state and executes "e1" from the emulation package. When the execution of "e1" from the emulation package arrives at the point of the subroutine call to "e2", we do not execute "e2" from the emulation package. Instead, we update the program state using postlog(4) generated by "e2" during the execution phase, and then proceed with the execution until it arrives at the end of "e1". The unexecuted portion corresponding to "e2" will be represented as a sub-graph node in the dynamic graph. If the user wants to know more about the execution detail of the sub-graph node, we then generate detailed traces of

I_4 by executing "e2" of the emulation package. The initial state for this re-execution will be prepared with prelog(4) from the log file.

In general, several subroutines can be called during the execution of a given subroutine, yielding several nested log intervals. In this case, we update the program state with those nested postlogs according to their order in the log file. Often times, it is possible that several such log intervals nested in one log interval will be generated by a single e-block, particularly when a subroutine is called inside a loop. In this case, we update the program state with only the last nested postlog made by the same e-block because the content of this postlog would overwrite all the other postlogs made by the same e-block.

5.4.3. Object Code and Emulation Package

The Compiler/Linker generates the object code and the emulation package during the preparatory phase. The object code contains the conventional executable code along with code to generate log entries. The emulation package is similar to the object code, except that instead of code to generate log entries, it contains code to generate a trace of every useful event. This section describes how the object code and emulation package are used to perform flowback analysis. Figure 5.7 illustrates the debugging scenario described in this section.

Once a program is stopped (due to either an error or user intervention), the PPD Controller locates the last prelog whose corresponding postlog has not yet been generated by the object code (step 1 in Figure 5.7). The Controller sets up the initial state for the emulation package using this prelog (step 2). Then, the Controller locates the e-block of the emulation package corresponding to this prelog and executes it to generate the necessary traces (step 3) to build a fragment of dynamic graph such as shown in Figure 5.2 (step 4). The Controller then presents a simplified and abstract form of the dynamic graph to the user (step 5).

In building the dynamic graph, the Controller uses the traces generated by the emulation package, the static graph, and the program database (as shown in Figure 3.3). The traces generated in this example correspond only to the log interval of the last prelog generated, so the portion of the dynamic graph presented to the user is incomplete. After examining the dynamic graph, the user tells the Controller which dependences are of interest, and therefore which part of the dynamic graph to next extend.

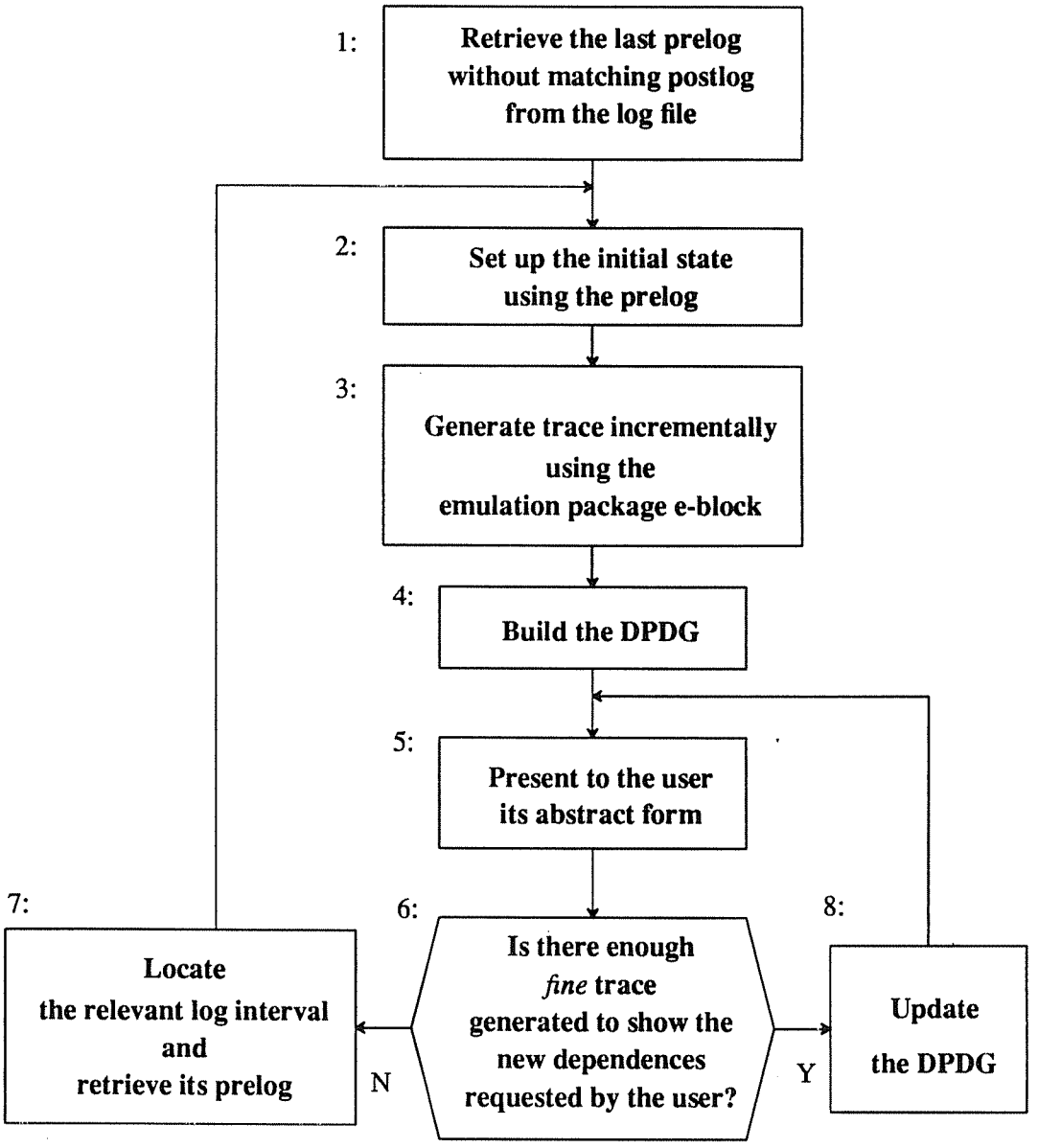


Figure 5.7.
Debugging Scenario

There are two possible cases here (step 6): first, there may already be sufficient traces generated to show the requested dependences; or, there may not be sufficient traces generated. In the first case, the Controller merely updates the dynamic graph or builds a new fragment of the dynamic graph using traces already generated (step 8). In the second case, the Controller locates, with the help of the static graph and

the program database, the log interval whose traces are needed to complete the requested part of the dynamic graph. The Controller retrieves the prelog for that interval from the log file (step 7) and sets up the initial conditions for the emulation package e-block (step 2 again). After that, the Controller locates the e-block corresponding to that log interval from the emulation package. The Controller then executes the emulation package e-block to generate the additional traces. Finally, the Controller either updates the dynamic graph or builds a new one. The entire process is repeated as necessary until the user has enough of the dynamic graph to locate the bug. Since only the portions of the dynamic graph in which the user is interested are generated, this technique is called incremental tracing.

5.4.4. Constructing E-blocks

In this section, we describe how to divide the program into numerous segments of code called e-blocks. The only condition for several consecutive lines of code to form an e-block is that the e-block must have 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. The entry point is where the prelog is made. The postlog is made at the exit point where control is transferred out of an e-block. One natural candidate for constructing an e-block is the subroutine since the entry and exit points 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 e-blocks large in favor of the execution phase, debugging phase performance will suffer. On the other hand, if we make the size of 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 an unacceptable performance degradation during the execution phase, it should also be large enough so as not to introduce unacceptable time delays in reproducing traces during the debugging phase.

Consider, for example, the case in which a subroutine contains loops. Even though their size may be small, the execution time for these loop components may be long and may introduce unacceptable time delays in reproducing the traces of an instance of the subroutine. E-blocks can be defined for such loops so that the debugging phase can proceed without spending an excessive amount of time re-executing the loops. If the user is interested in the execution details inside such loops, then the PPD Controller can re-

execute the e-blocks corresponding to the loops after all.

Small and frequently called subroutines can also 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 the small subroutines that correspond to leaf nodes in the call graph. In this case, the direct ancestor subroutines of these leaf subroutines *inherit* the USED sets and the DEFINED sets of the leaf subroutines, and they perform the logging for the descendent subroutines. Figure 5.8 shows two possible logs for the same program segment. In the figure, LOG A corresponds to the case where all the subroutines constitute e-blocks while LOG B corresponds to the case where only "SubA" (out of three subroutines shown) constitutes an e-block.

5.5. Log Structure and Locating Log Intervals

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 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 is connected to 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). In this section we describe the log structure and show, with an example, how to locate the log interval whose detailed traces are necessary to show the dependences requested by the user.

5.5.1. Log Structure

Figure 5.10 shows an example log file corresponding to an execution instance of the program in Figure 5.9. This program halted in the **while** loop of subroutine "Wolf" (given in Figure 4.1), during the second call to "Wolf" from "Cub". There are five e-blocks in this example: procedures "main", "Cub", "Wolf", "Add2" (called from "Wolf"), and the **while** loop in "Wolf". Log entries generated by the same e-block form a linked list. Each postlog has two pointers: one pointing to its corresponding

```
int g1, g2, g3, g4, g5;

SubA (i1)    int i1;
{
    g4 = SubB (i1);
    SubC (i1);
}; /* SubA */
```

```
SubB (a)    int a;
{
    return (a-g1+g5);
}; /* SubB */

SubC (b)    int b;
{
    g2 = g2 + b;
    g3 = g3 - b;
}; /* SubC */
```

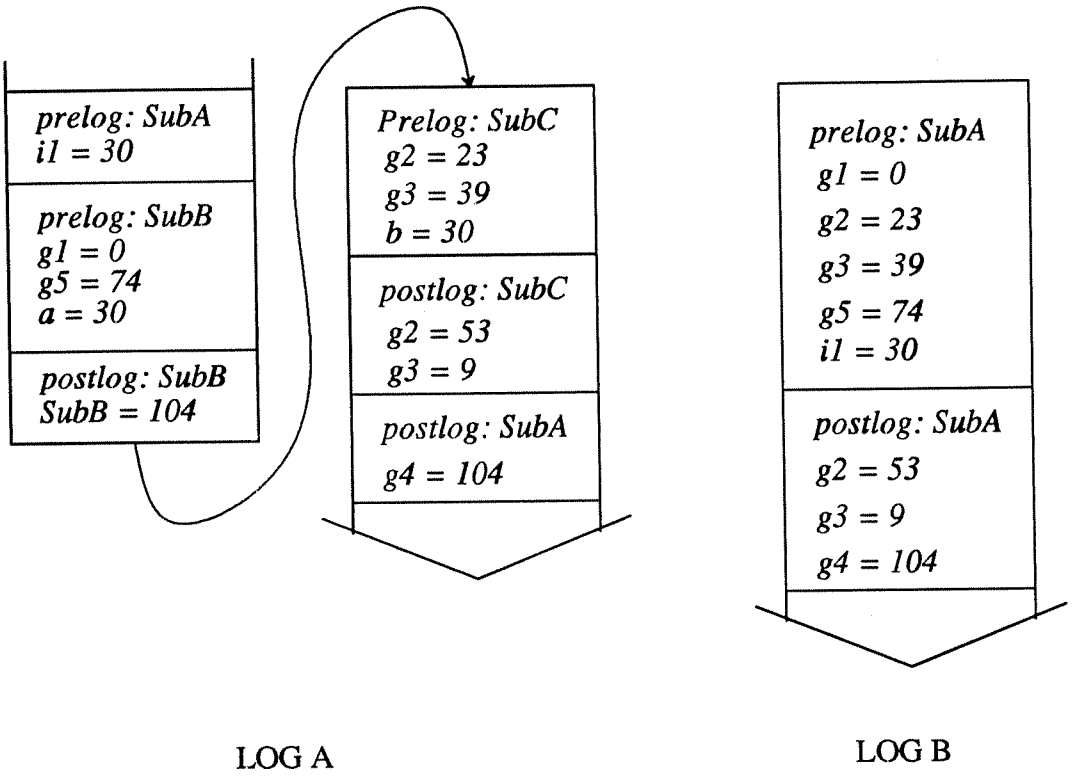


Figure 5.8.
A Sample Program with Two Different Logs

```
int g1, g2, g3;
main ()
{
   Cub ();
}; /* main */
```

```
Cub ()
{
   g1 = g2 = g3 = 0;
   Wolf ();
   if (g1 > 0)
     g2 = g1 + g2;
   Wolf ();
}; /* Cub */
```

Figure 5.9.
An Example Program

prelog, and the other pointing to the most recent postlog made by the same e-block. Each prelog has a pointer to its parent prelog (not shown in the figure). *E-pointers* is an array of pointers to the last log entry made by each e-block and is updated during program execution.

When the user wants to know the details of a hidden or dangling dependence, we need to identify the log interval whose fine traces are necessary to show the details. To identify such log intervals, we obtain the DEFINED set of each e-block during compile time and keep it as part of the program database[42].

We also keep in the program database, for each variable that might be accessed by one or more e-blocks, the list of e-blocks that contain the variable in their DEFINED sets. We call this list the *e-block table*. The e-block table in Figure 5.10 shows the list of e-blocks for three variables: "g1", "g2", and "g3".

5.5.2. Parameters

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 5.10, 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

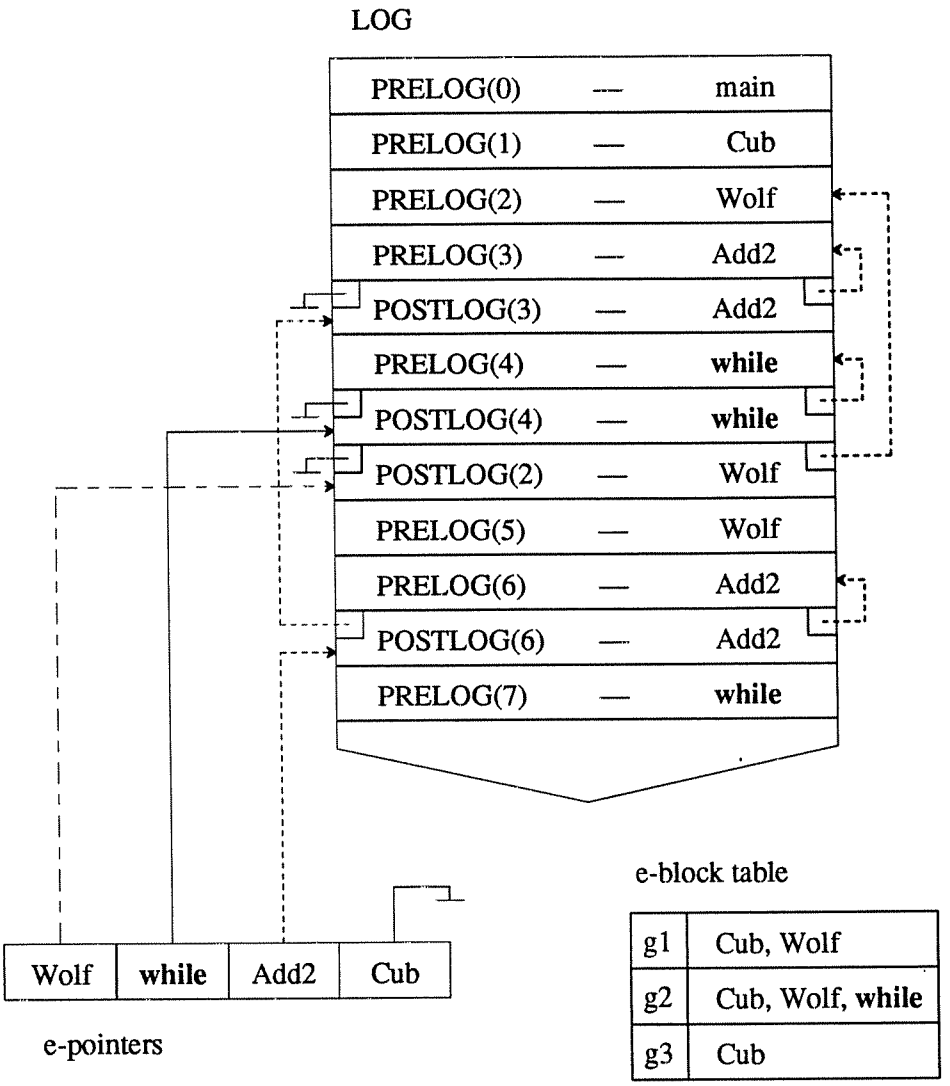
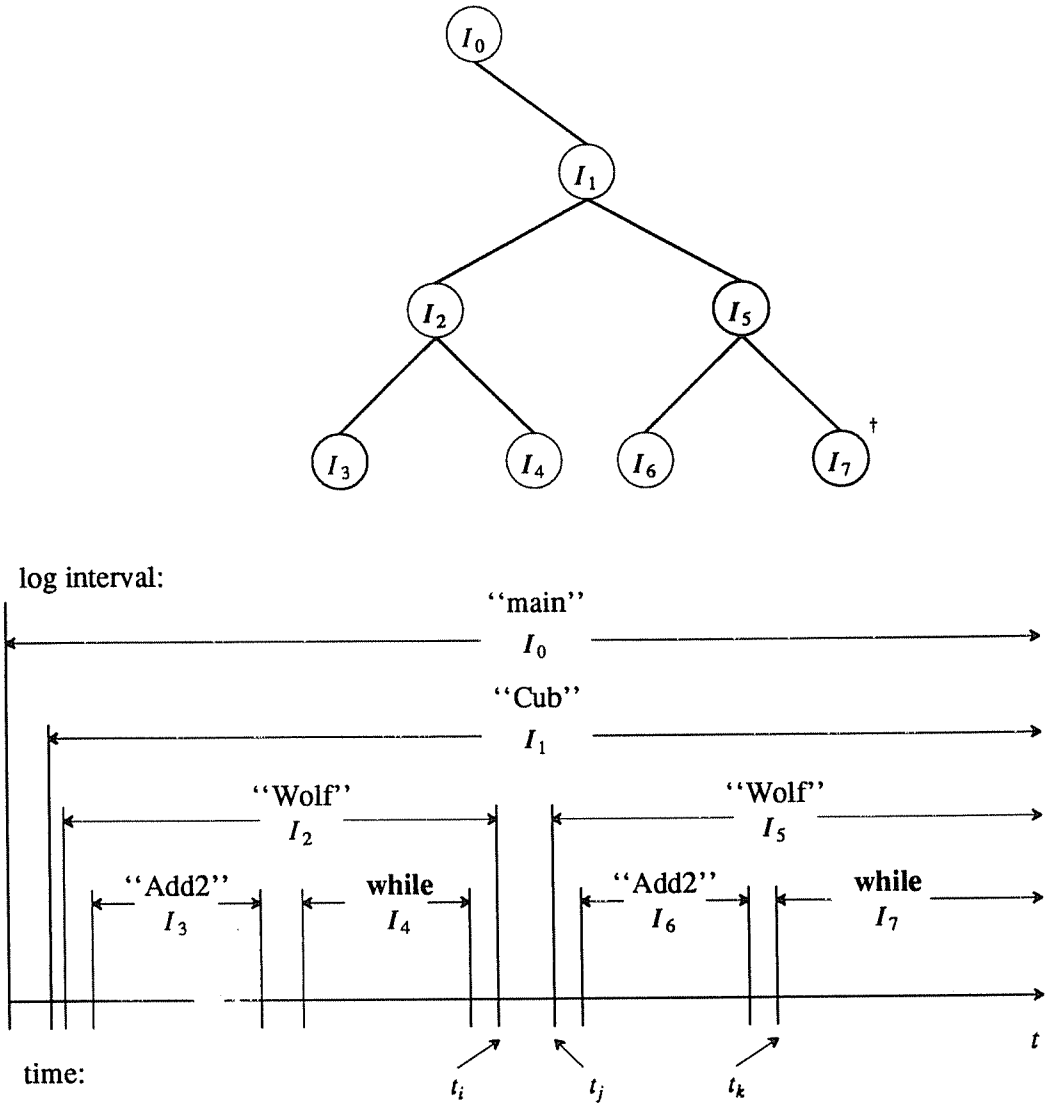


Figure 5.10.
LOG Structure (1)

generated in the execution order of the called subroutines.

5.5.3. Hidden or Dangling Dependences

Log entries are nested, and we can represent the log file by a tree as shown in Figure 5.11. Figure 5.11 also shows the timing diagram with corresponding log intervals for the example log in Figure 5.10.



† Current context log interval in the example.

Figure 5.11.
LOG Structure (2)

We will use Figure 5.11 to illustrate how to locate the candidate log intervals to show dangling or hidden dependences of a variable. Once we locate a candidate log interval to show the requested dependences, we generate a fine trace by using the log entries in that log interval and the emulation package e-block of the first prelog of that log interval. Note that the process of locating a candidate log interval and generating

detailed traces of the log interval may need to be repeated if the e-block did not actually modify the variable during that particular log interval.

To show dangling or hidden dependences of a variable in the *current* log interval, we need to locate the log interval that contains the most recent modification to the variable. There are two possible cases here. First, the most recent modification could have happened during a log interval whose execution already terminated before the current log interval started. In this case, that log interval has generated its postlog, and can be located by following the linked lists formed by the postlogs generated by the same e-blocks. We call that log interval the *previous interval.*

Second, the most recent modification could have happened during a log interval whose execution did not yet terminate; this case happens when the most recent modification belongs to one of the log intervals enclosing the current log interval. We call these enclosing intervals the *ancestor intervals*, because they correspond to the ancestor nodes of the current interval (node) in the log tree.[2] In this case, the postlog for this log interval has not yet been generated, and cannot be located by following the linked lists formed by the postlogs. For this case, we need to examine the ancestor log intervals (by generating detailed traces, if not already generated) whose DEFINED set contains the variable in question.

After we locate the previous interval and the ancestor intervals, we determine which of these log intervals to examine first based on their relative positions in the log tree. In general, between two log intervals I_a and I_b (where the postlog of I_a is before postlog of I_b), any of the ancestor log intervals of I_b, starting from the parent interval up to the most recent common ancestor of the two intervals, may have modified the program state between the two intervals. Thus, after we locate the previous interval, we need to examine all the ancestor intervals of the current interval up to the most recent common ancestor interval of the two intervals before we examine the previous interval. This is true even when I_b is an ancestor interval of I_a; in this case, I_b is also the most recent common ancestor of the two.

Algorithms 5.1a and 5.1b describe the process to show the dangling or hidden dependences. In Algorithm 5.1a, subroutine *CommonAncestor* returns the most recent common ancestor of the two intervals passed as its parameters.

2. We do not actually build the log tree from the log file. The log file can be simply viewed as a tree, because each nested log interval has a pointer to its parent log interval.

```
ResolveDependences (curr_ival: log interval; var_in_question: variable);
    begin
        /* initialize the variables used in the following loop */
        from the e-block table, retrieve the list of e-blocks that contain
           var_in_question in their DEFINED sets;
        cand_eblocks ← set of these retrieved e-blocks;
        if curr_val has finished its execution then
            prev_postlog ← postlog of curr_ival;
        else
            prev_postlog ← pointer to end-of-file;
        end if;
        top_ancestor ← curr_ival;
        do forever      /* not exactly */
                        /* in Algorithm 5.1b */
            prev_ival ← LocatePrevInterval (prev_postlog, cand_eblocks);
            /* Do not examine what has already been examined */
            ival_to_exam ← log interval enclosing top_ancestor;
            top_ancestor ← CommonAncestor (curr_ival, prev_ival);
                /* in Algorithm 5.1b */
            if (ExamAncestors (top_ancestor,
               ival_to_exam, var_in_question) = SUCCESS) then
                return (SUCCESS);
            end if;
                /* in Algorithm 5.1b */
            if (ExamIval (prev_ival, var_in_question) = SUCCESS) then
                return (SUCCESS);
            end if;
            prev_postlog ← postlog of prev_ival;
            if prev_postlog is the first postlog in the log file then
                return (FAILURE);
            end if;
        end do          /* do forever */
end ResolveDependences;
```

Algorithm 5.1a. An Algorithm to Resolve Dangling or Hidden Dependences

5.5.4. Example

In Figure 5.11, we assume the **while** loop of I_7 is the current interval with a dangling dependence for "g2", and we want to know the most recent modification of "g2". In this example, I_2 is the previous interval — the interval that has the most recent postlog with "g2" in it. I_5, I_1, and I_0 are the ancestor intervals of the current interval. I_1 is the most recent common ancestor of the current interval and the previous interval.

When we look at the timing diagram in Figure 5.11, we can see that "g2" might have been modified after the termination of I_2 and before the start of I_7 (between t_i and t_k). The time interval between t_i and t_k can be partitioned into two intervals: one belonging to I_1 (between t_i and t_j) and the other belonging to I_5

```
LocatePrevInterval (curr_postlog: postlog, cand_eblocks: set of eblocks)
    begin
        locate the most recent postlog before curr_postlog by following
          the linked lists formed by the postlogs of cand_eblocks;
        return (log interval of this most recent postlog);
end LocatePrevInterval;

ExamIval (ival: log interval; var_in_question: variable)
    begin
        examine ival, possibly by generating detailed traces;
        if dependence for var_in_question is resolved in ival then
            return (SUCCESS)
        else
            return (FAILURE);
        end if;
end ExamIval;

ExamAncestors (top_ancestor, ival_to_exam: log interval;
      var_in_question: variable)
    begin
        do forever
            if (ival_to_exam = null) then
                return (FAILURE);
            end if;
            if ival_to_exam is an ancestor of top_ancestor then
                /* higher than top_ancestor */
                return (FAILURE);
            end if;
            if (ExamIval(ival_to_exam,
                  var_in_question) = SUCCESS) then
                return (SUCCESS);
            end if;
            ival_to_exam ← parent of ival_to_exam;
        end do;
end ExamAncestors;
```

Algorithm 5.1b. An Algorithm to Resolve Dangling or Hidden Dependences (continued)

(between t_j and t_k).[3] Of the two time intervals, the interval belonging to I_5 is the closer to the current interval (I_7). I_5 and I_1 both have "g2" in their DEFINED sets.[4] So, we have to examine I_5 and I_1, in that order, before we examine I_2, which is the previous interval.

If one of the intervals that we examine has modified "g2", then we are done. If we again fail to resolve the dependence for 'g2" with the previous interval (I_2), we locate the next previous interval (I_4 in this example; note that the postlog of I_4 precedes the postlog of I_2) and iterate, performing the whole process repeatedly until either the dependence is resolved or no more log intervals are left to examine. In the second case, we would have found a bug, namely the use of an uninitialized variable.

5.5.5. Logging and Parallel Programs

In the case of a parallel program, there is one log file for each process. Also, the incomplete part of the dynamic program dependence graph presented to the user may cross process boundaries if an event of one process is dependent upon an event of a different process. In this case, the PPD Controller locates the second process and its log interval in order to generate the traces necessary to show the requested dependences. In Chapter 6, we describe how to locate the second process and how to determine the log interval of the second process needed for generating traces.

5.6. Arrays and the Log

For an e-block with array accesses, it is not possible (in general) to compute JSED and DEFINED sets that contain only those array elements that are actually accessed in the e-block. One approach to addressing this problem 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 trace data 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 indices have a (possibly transitive) data dependence on the loop control variable. With a

---

3. Though time interval between t_j and t_k might also be regarded as belonging to I_1, we say it belongs to I_5 because I_5 is a sub-interval of I_1.

4. I_6 does not have "g2" in its DEFINED set. Otherwise, I_6, not I_2, would have been the previous interval.

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 identifier, the address of the array element accessed, and the accessed (read or updated) value of the array element at the time the access is made.

Chapter 6

Dependence Graphs and Parallelism

The discussion so far has described mechanisms to efficiently implement flowback analysis and incremental tracing; parallel programming has been addressed only to a limited extent. In this chapter, we discuss the problems that arise in applying flowback analysis and incremental tracing to parallel programs and mechanisms to handle these problems. First, we describe the potential problems in applying incremental tracing to parallel programs that lack reproducibility, and we describe our solution to the problems (such as additional logging for shared variables). Second, we describe the methods we use in applying flowback analysis and incremental tracing to parallel programs, including our approaches to event ordering and data-race detection.

6.1. Simplified Static Graph and Shared Variables

So far we have assumed the reproducibility of the debugged program in describing incremental tracing. In this section, we present the problems in applying these operations to parallel programs that lack reproducibility and also describe our solutions to the problems. Our solutions to these problems uses a graph called the *simplified static graph*, which is a subset of the static graph that abstracts out everything except for the potential interactions between processes.

6.1.1. Simplified Static Graph

Figure 6.1 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 can make a prelog that saves the value of "SV" at the beginning of the e-block representing the subroutine. The value of "SV" will not be changed until it is first accessed in the statement indicated by the arrow. Hence, we need at most one prelog and one postlog for each e-block for sequential programs.

However, let us now consider the case of a parallel program. If "SV" is a shared variable, we cannot guarantee that the value of "SV" saved at the beginning of the e-block will be the same when

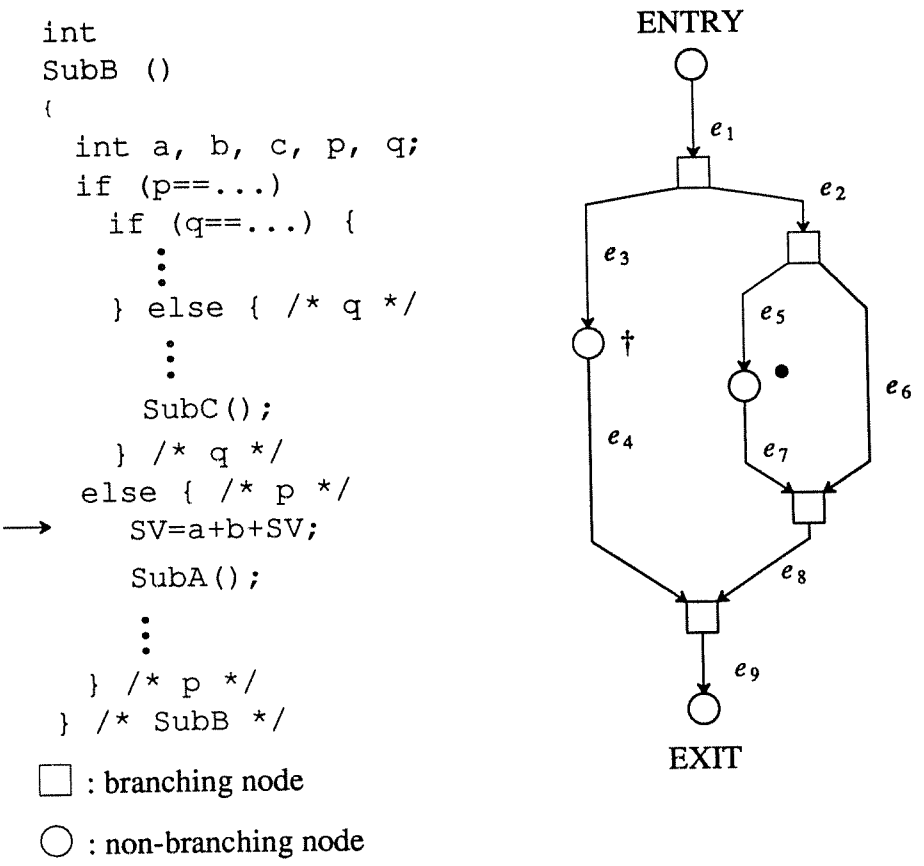


Figure 6.1.
A Subroutine and Its Simplified Static Graph

"SV" is first read in the e-block; other processes may have changed the value of "SV" between these two moments. Reproducible program behavior is not guaranteed, so traces generated during the debugging phase may not be the same as those that might have been generated during the execution phase. We need to record more run time information to ensure the reproducibility of parallel programs. Such additional information will be used either to detect data races or to restore the program state for read-accessed shared variables. The simplified static graph allows us to determine what additional information we have to generate and where in the program we have to generate this information for a parallel program.

Figure 6.1 shows the simplified static graph for subroutine *SubB*. The simplified static graph is a subset of the static graph with only one edge type (flow edge). ENTRY, EXIT, and sub-graph nodes (corresponding to subroutine calls) are the same as in the static graph. However, singular nodes include only control predicates (shown as branching nodes) and synchronization operations such as P and V semaphore operations. In our examples, we will use only the semaphore operations (P and V operations) as synchronization operations; however, this approach can easily be generalized to other synchronization primitives. In constructing the simplified graph, we treat a subroutine call (a sub-graph node in the graph) that may perform semaphore operations during its execution (or during the execution of any subroutine that might be transitively called by it) in the same way that we treat a semaphore operation. Such information is obtained by interprocedural analysis.

In constructing the simplified static graph, we are not interested in other singular nodes, so they are omitted from the simplified static graph. Thus, the non-branching nodes in the simplified static graph correspond only to ENTRY nodes, EXIT nodes, synchronization operations, or subroutine calls. Branching nodes in the simplified static graph represent the possible control transfers from **if** or **case** statements.

We also compute the set of read-accessed shared variables and the set of write-accessed shared variables for the statements represented by each edge, and we keep this information in the program database. This information is used to detect data races in accessing shared variables. A more detailed discussion of the issue of detecting data races is presented later in this chapter.

6.1.2. Synchronization Units and Additional Logging

We need to generate additional log entries for shared variables to debug parallel programs. By partitioning the program into *synchronization units*, the simplified static graph allows us to identify what additional information we need to generate and where in the program we have to generate that information. We define the synchronization unit as follows:

Definition 6.1

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.

Thus 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 6.1 each constitute a synchronization unit. We use such synchronization units both in determining the additional run-time logging to be performed for debugging parallel programs and in determining the execution points where such information is to be generated.

The object code generates an additional prelog at the beginning of each synchronization unit for those shared variables that could be 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. If a synchronization unit does not contain any accesses to a shared variable, then the synchronization unit does not generate a prelog for shared variables.

If no data races exist in accessing shared variables, we can ensure repeatable execution behavior of the parallel program by using this prelog for shared variables. If data races exist, then we can detect and show the causes of the races. Further discussions of the method for detecting data races in an execution instance of a parallel program appears in Sections 6.2.3 – 6.2.5.

Synchronization units are used in detecting data races; we use them in constructing bit vectors of the basic blocks potentially executed between synchronization operations [7]. The trace records for these bit vectors (generated during run time) and the sets of read-accessed and write-accessed shared variables in each basic block (computed during compile time) allow us to determine the sets of read-accessed and write-accessed shared variables between two synchronization operations, thus allowing us to detect data races (as will be explained shortly).

6.2. Parallel Dynamic Graph and Ordering Concurrent Events

In this section, we discuss the problems that arise in applying flowback analysis to parallel programs along with mechanisms to handle these problems. Ordering events belonging to different processes allows for the detection of data dependences that exist across process boundaries. It also allows for the detection of data races in interactions between co-operating processes. Detecting data races in a parallel program consists of two steps: detecting the events that cannot be ordered with respect to each other, and detecting the read/write or write/write conflicts on shared variables between those unorderable events.

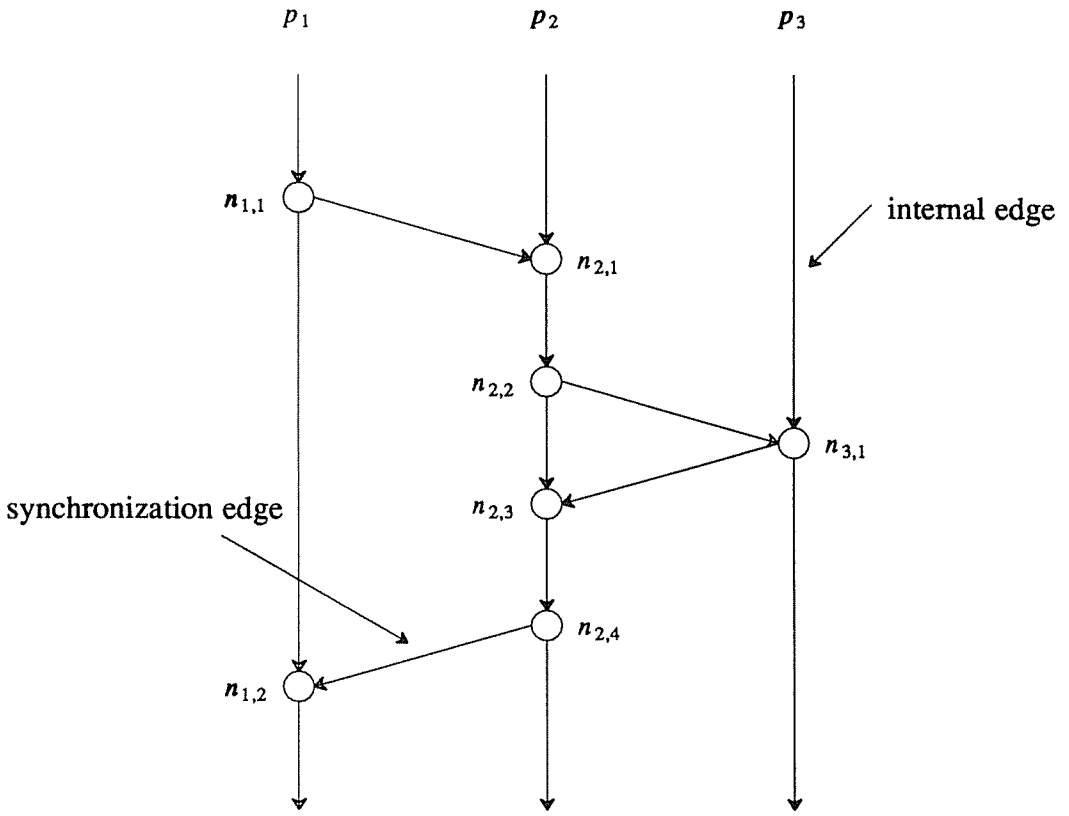


Figure 6.2.
An Example Parallel Dynamic Graph

6.2.1. Parallel Dynamic Graph

The *parallel dynamic program dependence graph* (or *parallel dynamic graph*) [42] allows us to detect the unorderable sets of events in a parallel program. The parallel dynamic graph is a subset of the dynamic graph that abstracts the interactions between processes while hiding the detailed dependences of local events. This graph contains only one node type, called *synchronization node*, and two edge types, *synchronization edges* and *internal edges*. Figure 6.2 shows an example of a parallel dynamic graph. A synchronization edge between two synchronization nodes represents a causal relationship between those nodes. 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.

There exists a *happened-before relationship* [37] between the two nodes of an edge, such that the event of the source node happened before the event of the sink node. In Figure 6.2, node $n_{1,1}$ represents the event of V semaphore operation, and node $n_{2,1}$ represents the P semaphore operation unblocked by the V operation of node $n_{1,1}$.

An internal edge represents a group of flow edges in a process's execution, hiding zero or more local events between the pair of synchronization nodes connected by the edge. In Figure 6.2, all the events of process p_1 that happened before event $n_{1,1}$ also happened before all those events of process p_2 that happened 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.

6.2.2. Constructing Synchronization Edges

A synchronization edge between two nodes represents a causal relationship between the events represented by the nodes and allows for the partial ordering of the program's concurrent events. In this section, we describe how to construct synchronization edges for various synchronization operations such as semaphore operations. In general, we construct a synchronization edge between two concurrent synchronization events if we can identify a causal relationship between them.

Semaphores and Synchronization edges

Semaphore operations, such as V and P, are used in controlling accesses to shared resources by either releasing resources (through a V operation) or acquiring resources (through a P operation). We construct a synchronization edge from the operation that releases resources (V) to the operation that acquires those released resources (P).

There are two cases to be considered here. 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. Implementation details of how we identify those matching semaphore operations are described in Chapter 7. Other synchronization operations that can be treated similarly are the monitor [25] and the binary semaphore (whose integer value can range only between 0 and 1).

Messages and Synchronization Edges

When a system uses messages rather than semaphores for inter-process communication, we define a synchronization edge by a source node for the event of sending a message, a sink node for the event of receiving the message, and a synchronization edge from the sending node to the receiving node.

If the sender is blocked until the message is received by the receiver, we also define a sink node at the sender for the event of unblocking and a synchronization edge from the receiving node to this sink node of the sender. The node at the receiver becomes the sink node of the synchronization edge representing the event of sending and receiving a message, and the source node of the synchronization edge representing the unblocking of the sender process. In Figure 6.2, node $n_{2,2}$ corresponds to a blocking send, while node $n_{3,1}$ represents the receiving of the message. Node $n_{2,3}$ represents the unblocking event. The edge between $n_{2,2}$ and $n_{2,3}$ contains zero events.

Once we identify the sender and the receiver of each message, data dependences through messages can be resolved in a manner analogous to resolving dependences through assignment statements. Defining the value of a message to be sent is similar to defining the value of a variable, and using an element of a received message is similar to using a variable. The major difference is that while variables in assignment statements are identified by their unique identifiers (which are assigned by the compiler), each element used in a message is identified by its size and its offset from the beginning of the message.

Rendezvous and Synchronization Edges

For the Ada rendezvous [3], we define a source node (in the graph for the caller process) for the event of calling the rendezvous, a sink node (in the graph for the callee process) for the event of accepting the call, and a synchronization edge between the two nodes. We also define a source node (in the graph for the callee) for the event of exiting from the rendezvous block, a sink node (in the graph for the caller) for the event of returning from the rendezvous call, and a synchronization edge between the two nodes. The

internal edge (in the graph for the caller) between the event of calling the rendezvous and the event of returning from the call contains zero events since the caller is suspended during the call.

We can treat remote procedure calls (RPC) in a similar way using two synchronization edges, one for calling to, and another for returning, from the RPC. Message semantics that require the receiver to send a special *reply* message [16] to the sender are similar to the rendezvous.

6.2.3. Simultaneous Edges and Race Conditions

In the parallel dynamic graph, each internal edge represents a set of events bounded by the surrounding synchronization operations, and any two events belonging to an internal edge are totally ordered. Any two events belonging to two different internal edges are ordered if and only if the two internal edges are ordered in the graph. Thus, unorderable events can be detected by locating internal edges that cannot be ordered. These unorderable edges are called *simultaneous edges*. Once we detect simultaneous edges, we can use the semantic information gathered during compilation to find any read/write or write/write conflicts between them.

We define the partial ordering of nodes and the edges of the parallel dynamic graph using the "$\rightarrow$" operator as follows [37]:

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.

Since an internal edge of a process consists of local events of the process, we can order concurrent events by ordering the internal edges to which the concurrent events belong. If there is no ordering between two internal edges, then a potential data race exists.

We will now define data races more formally. As a reminder, note that we have stated that, 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 e_1 and n_2 is the source node of e_2.

Definition 6.2

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 6.3

READ_SET(e_i) is the set of the shared variables read-accessed in edge e_i. *WRITE_SET*(e_i) is the set of the shared-variables write-accessed in edge e_i.

Definition 6.4

We say two simultaneous edges e_1 and e_2 are *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$(e_2) = \emptyset$.

In other words, two simultaneous edges e_1 and e_2 are race-free if no read/write conflict or write/write conflict exists between them.

Definition 6.5

An execution instance of a program is said to be race-free if all pairs of simultaneous edges in the execution instance are race-free. An execution instance of a program is said to have data races if it is not race-free.

The reason why one can say only that an execution "instance" of a parallel program is race-free is that one cannot tell if a parallel program is race-free unless one considers every possible event (internal or external to the program) that may affect the timing of the program's process interactions.

6.2.4. Ordering Events

This section describes an algorithm that orders events in different processes by examining the parallel dynamic graph. For each node n in the graph, the algorithm computes two vectors, *before*$[n]$ and *after*$[n]$, such that for each process p, *before*$[n][p]$ is the sequential number (which is unique within a given process) of the latest node that happened before node n in process p, and *after*$[n][p]$ is the sequential number of the earliest node in process p that happened after node n. For each process, by definition, *before* $[n_{p,i}][p]$ and *after* $[n_{p,i}][p]$[1] are equal to i (statements labeled with ‡ in Algorithm 6.1). Also for each process p, *before* $[n_{p,i}][q]$ monotonically increases with increasing i, and *after* $[n_{p,i}][q]$

1. Node labels are subscripted with the process number and the node number (both are greater than 0). Thus, node $n_{p,i}$ can be read as "node i of process p".

monotonically decreases with increasing i (statements labeled with † in Algorithm 6.1). This information can be used to determine the ordering between any two nodes $n_{p,i}$ and $n_{q,j}$. There are three cases:

(1) node $n_{p,i}$ happened before node $n_{q,j}$

if $j \geq after[n_{p,i}][q]$,

(2) node $n_{p,i}$ happened after node $n_{q,j}$

if $j \leq before[n_{p,i}][q]$, and

(3) nodes $n_{p,i}$ and $n_{q,j}$ are unorderable

if $before[n_{p,i}][q] < j < after[n_{p,i}][q]$.

The event ordering algorithm shown in Algorithm 6.1 operates in two passes. The first pass computes the *before* vectors by a top-down traversal of the parallel dynamic graph. Since the *before* vector of a node can be computed from the *before* vectors of its predecessors, a topological ordering of the graph gives a natural traversal order. The second pass computes the *after* vectors similarly, but in the reverse direction. Since each pass considers every edge exactly once, and each vector is of length p, both the time and space complexity of the algorithm are $O(ep)$, where e is the number of edges in the graph and p is the number of processes. The topological sort can be performed in O(number of nodes) time and space [53] and thus is dominated by the rest of the computation. Figure 6.3 shows a parallel dynamic graph labeled with the *before* and *after* vectors after performing the node ordering algorithm.

6.2.5. Detecting Data Races

Once we order the events in a program execution, we then can detect data races. If there exists a read event to a shared variable that occurred simultaneously with a write event, then a read/write data race exists. If there exists a write event to a shared variable that occurred simultaneously with another write event, then a write/write data race exists. This section describes an algorithm that detects data races by examining the parallel dynamic graph and the *before* and *after* vectors of the graph.

For edge $e_{p,i}$ of process p (which is bounded by nodes $n_{p,i}$ and $n_{p,i+1}$), nodes $before[n_{p,i}][q]$ and $after[n_{p,i+1}][q]$ give the boundary of edges (of process q) that are simultaneous with $e_{p,i}$. Once we identify a pair of simultaneous edges, we check for the presence of data races according to Definition 6.5.

```
OrderEvents (PG: parallel dynamic graph)
  var
      travOrder: array of node;
  begin
      travOrder ← tsort(PG);  /* perform topological sort */
      OrderEventsForward(PG, travOrder);
      OrderEventsBackward(PG, travOrder);
  end OrderEvents;

OrderEventsForward (PG: parallel dynamic graph; travOrder: array of node)
  begin
      before [n ]←<0,...,0> for each node n;
      foreach node n_{p,i} in the order of travOrder
          for j ← 1 to numProcesses do
              before [n_{p,i}][ j ] ← MAX(before [b ][ j ]) for all source nodes b
                  of inward synchronization edges of n_{p,i};
/* † */           if i>1 then
                  before [n_{p,i}][ j ] ← MAX(before [n_{p,i}][ j ], before [n_{p,i-1}][ j ]);
              end if
/* ‡ */       before [n_{p,i}][ p ] ← i;
  end OrderEventsForward;

OrderEventsBackward (PG: parallel dynamic graph; travOrder: array of node)
  begin
      after [n ]←<∞,...,∞> for each node n;
      foreach node n_{p,i} in the reverse of travOrder
          for j ← 1 to numProcesses do
              after [n_{p,i}][ j ] ←  MIN(after [b ][ j ]) for all sink nodes b
                  of outward synchronization edges of n_{p,i};
/* † */           if i < last node in process p then
                  after [n_{p,i}][ j ] ← MIN(after [n_{p,i}][ j ], after [n_{p,i+1}][ j ]);
              end if
/* ‡ */       after [n_{p,i}][ p ] ← i;
  end OrderEventsBackward;
```

Algorithm 6.1. Node Ordering Algorithm

The race-detection algorithm shown in Algorithm 6.2 operates in a single pass. Routine RaceCheck in Algorithm 6.2 uses Definition 6.4 to see if the two edges are race-free. For a parallel dynamic graph with e edges, the worst-case time complexity of Algorithm 6.2 is $O(e^2)$. In practice, we can expect the number of edges that are simultaneous with a given edge to be bounded by some constant, effectively yielding $O(ep)$ time complexity, where p is the number of processes. The space complexity is also $O(ep)$. DataRaceBackward can be incorporated into OrderEventsBackward in Algorithm 6.1, resulting in no additional space overhead.

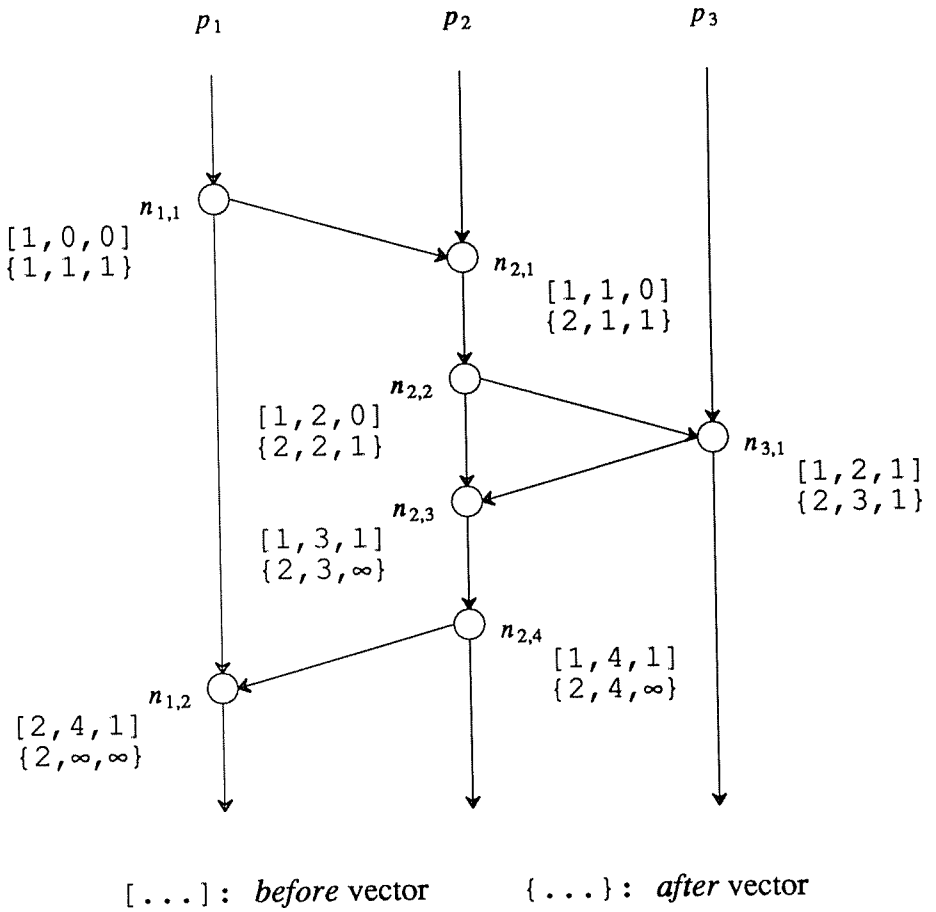


Figure 6.3.
Parallel Dynamic Graph with Time Vectors

6.2.6. Data Dependences for Parallel Programs

We can extend the program's data dependence edges across process boundaries for a race-free execution instance using the algorithm shown in Algorithm 6.3. Given a read event for some shared variable, its data dependence is located by finding the most recent write event that happened before that read. This can be accomplished as follows. First, for each process, the latest edge that can be ordered before the read event is located using the *before* vector computed by Algorithm 6.1. These edges give a boundary beyond which all events either occurred simultaneously with the read event or after the read event. Second, the log for each process is scanned backwards from this boundary to find an edge whose WRITE_SET contains the shared variable in question (subroutine LatestEdge in Algorithm 6.3). Finally,

```
DetectRaces (PG: parallel dynamic graph)
  var
      travOrder: array of node;
  begin
      travOrder ← tsort(PG); /* perform topological sort */
      initialize checked[n_i] = false for each node n_i;
      before[n_p,0] ← <0,...,0> for each process p;
      DetectRaceBackward(PG, travOrder);
  end OrderEvents;


DetectRaceBackward (PG: parallel dynamic graph; travOrder: array of node)
  begin
      foreach node n_p,i in the reverse of travOrder
          checked[n_p,i] = true;
          for j ← 1 to numProcesses do
              k ← before[n_p,i][j];
              if n_p,i is not the last node of process p then
                  l ← after[n_p,i+1][j];
              else
                  l ← 1 + (last node number of process j);
              end if
              for m ← k to l - 1 do
                  if not checked[n_j,m] then
                      RaceCheck (e_p,i, e_j,m);
                  end if
  end DataRaceBackward;
```

Algorithm 6.2. Algorithm to Detect Data Races

all such write events are ordered to determine which one actually occurred last. A data dependence can then be drawn from this event to the read event.

The time complexity of the first step is $O(p)$, where p is the number of processes. The second step is to locate, for each process, the most recent postlog that write-accessed a given variable (described in Section 5.4). The time complexity of this step is, in the worst case, $O(e)$, where e is the number of edges in the graph; for each process, we might need to scan all the edges of the process above the boundary located by the first step. The time complexity of the third step is $O(p \log p)$, by sorting the edges using the *after* vectors — we can decide the ordering of two edges by looking at the *after* vectors of the nodes immediately following the two edges. The first and third steps do not use space in addition to that used by the node-ordering algorithm. The second step requires additional space for debug-time traces.

```
InterProcessDataDep (PG : parallel dynamic graph; e_{p,i} : edge;
                     SV : shared variable)
  /* e_{p,i} contains a dangling dependence to SV */
  begin
     for j ← 1 to numProcesses do
        latest_nodes[ j ] ← before[n_{p,i}][ j ];
        latest_edges[ j ] ← LatestEdge (PG, j, latest_nodes[ j ], SV );
     latest_edge_of_all ← SortEdges (latest_edges);
     re-execute e-block(s) that generated log interval(s) corresponding
       to latest_edge_of_all;
     locate the last node that wrote to SV in latest_edge_all ;

  end InterProcessDataDep;
```

Algorithm 6.3. Algorithm to Identify Interprocess Dependences

Figure 6.4 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 exists between the event in $e_{2,1}$ that modified "SV" and the event in $e_{3,1}$ that read "SV". If there exists another event that modified "SV" in edge $e_{1,1}$, $e_{1,2}$, $e_{2,2}$, $e_{2,3}$ or $e_{2,4}$ (i.e., edges simultaneous to $e_{3,1}$), then we cannot tell which event actually modified "SV" last — because a data race exists.

6.3. Postlogs and Restoration of Program States

Halting co-operating processes of a program in time so as not to lose any valuable information about the global state is difficult for parallel programs [41]. However, restoring the program state to a prior point of program execution solves that problem since we can easily restore the program state to any point of program execution for each halted process. The postlogs generated during execution will allow for such restoration during the debugging phase. The accumulation of the information carried by all the postlogs of process k, from the first postlog, $postlog_k(1)$, up to $postlog_k(i)$, is the same as the information carried by the program state of process k at the time at which $postlog_k(i)$ is made. Therefore, we can restore the program state of process k by using the postlogs from $postlog_k(1)$ up to $postlog_k(i)$. The program state at

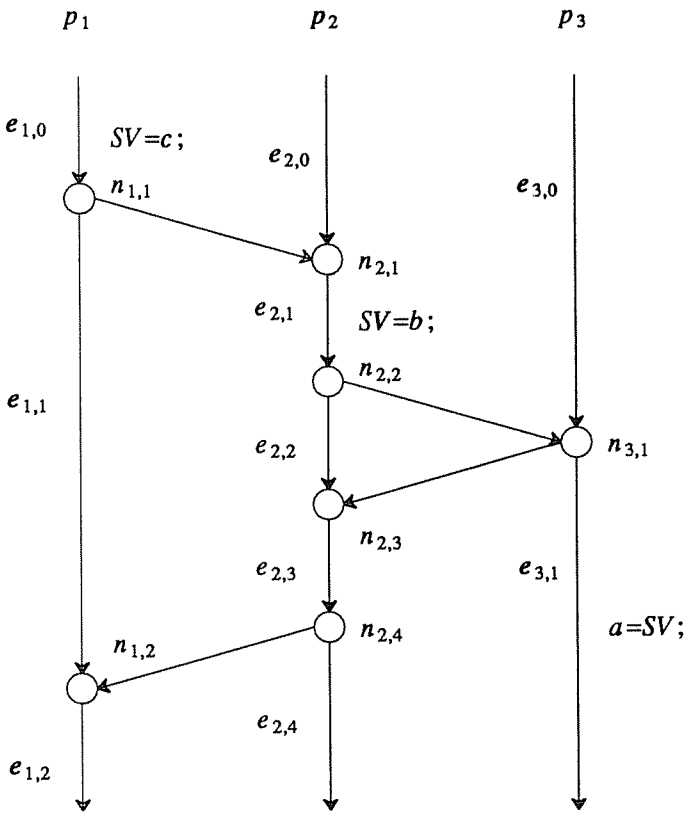


Figure 6.4.
Dependences between Concurrent Events

any time after that can be restored by using the restored program state and the object code.

Chapter 7

Implementation

In this chapter, we describe the implementation of PPD and provide performance measurements of the various parts of PPD. We first describe the compiler, and then describe how we define and identify (at debug time) synchronization dependences for semaphore operations such as P and V. We compare the performance of our instrumented compiler with the Sequent C Compiler [54] in terms of the compilation time, the size of the produced object code, and the execution time of the object code. We also present the measurement results of the run-time trace sizes of various programs compiled by our compiler, and discuss the interactive response time of PPD in resolving a dependence requested by the user during debug time.

7.1. Four Phases in Compilation

The PPD compiler consists of four phases as shown in Figure 7.1. During the first phase, the compiler generates assembly code with labels in it to uniquely identify the places where future modifications might be needed. During the second phase, the compiler does interprocedural and shared-variable analysis, and makes decisions on logging code optimization. During the third phase, the assembly code files are modified in accordance with the information generated during the second phase. These assembly code files are assembled and linked together during the fourth phase to yield the object code and the emulation package (described in Chapter 3). Piecewise data structures generated during the first three

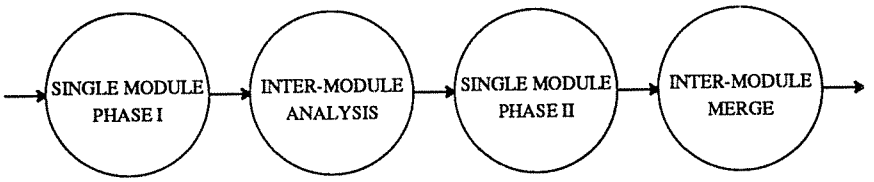


Figure 7.1.
Four Phases in Compilation

phases are also merged into global data structures during this fourth phase.

The first phase is a single-module phase. During this phase, the compiler generates two assembly code files (one to be executed during run-time and the other to be executed during debug-time) from each source module. The compiler also generates local static program dependence graphs (the pre-graphs described in Chapter 4) and a local program database for each module (file) during this phase.

The second phase is an inter-module analysis phase. During this phase, the compiler does interprocedural and shared-variable analysis of the program using the static graphs and the program databases generated during the first phase. Based upon the results of these analyses, the compiler generates global information such as global static program dependence graphs (the post-graphs described in Chapter 4). It also makes decisions on log optimization (described in Chapter 4) and on logging for shared variables (described in Chapter 6).

The third phase is another single-module phase, during which the compiler modifies each assembly code file generated during the first phase using information from the second phase.

The fourth phase is an inter-module merge phase, during which the assembly files are assembled and linked into the object code and the emulation package. Also, pieces of global information generated during the second phase are merged into one global program database during this phase. We describe each phase in more detail in the following sections.

7.1.1. First Phase (Single-Module Phase I)

Figure 7.2 shows the first phase of compilation for two example source modules (''x.c'' and ''y.c''). During this phase, the compiler generates two assembly files for each source module: one for the object code and the other for the emulation package. It also generates, for each source module, a local static graph file and a local program database file. The compiler uses a directory named *PPD* (under the working directory) to store files that would not be created by the original, unmodified compiler.

In generating the assembly files, the compiler does not optimize the logging code; it constructs one e-block out of each subroutine and one e-block out of each loop. For nested loops, it currently constructs only one e-block from the outermost loop. Each e-block in the object-code assembly files starts with code to generate a prelog entry and ends with code to generate a postlog entry. However, these assembly files

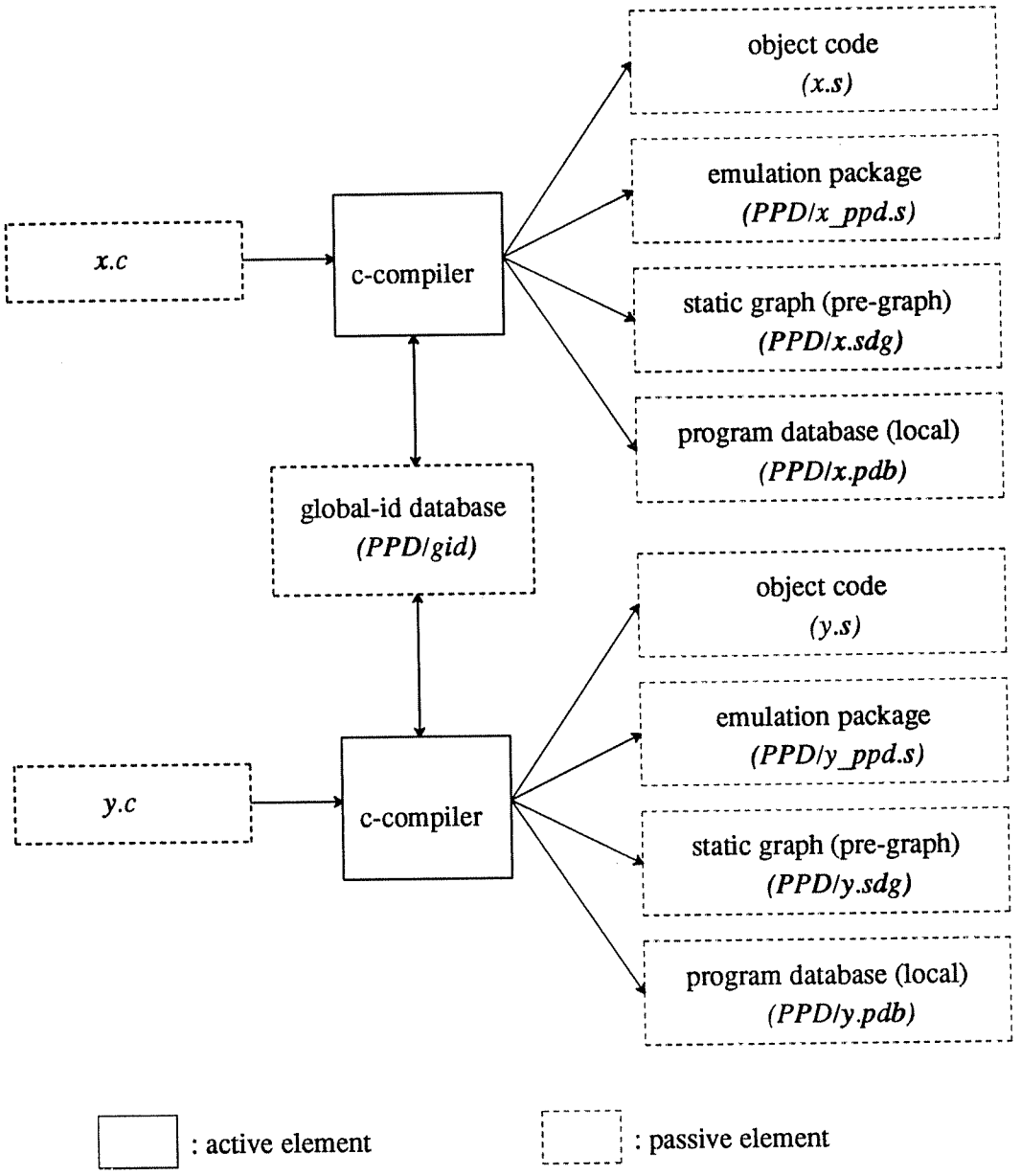


Figure 7.2.
Single-Module Phase (I)

do not have logging code for the shared variables accessed in each synchronization unit. Logging for shared variables is determined during the second phase, and logging code is inserted into the assembly code files during the third phase.

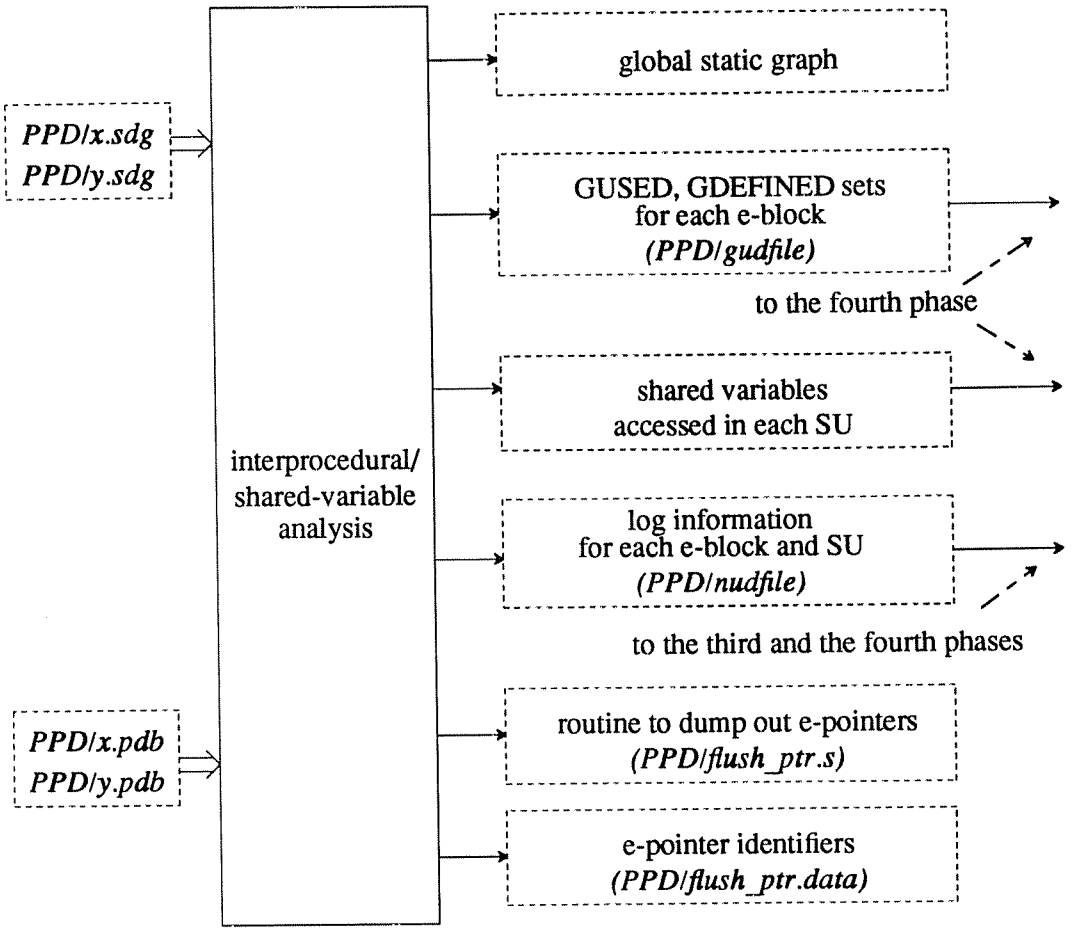
To ease the modification of the assembly files, the compiler inserts unique labels at places in the assembly files where logging for shared variables might become necessary. Unique labels are also inserted into the assembly files to delimit the code related to the construction of e-blocks. These labels are used, during the second and third phases, to identify the places where the logging code should be optimized. A more detailed description of the modifications is given in later sections.

The static graphs generated during the first phase are pre-graphs (described in Chapter 4) that lack any interprocedural information. The program database contains information about symbols declared in each module. It also contains the USED and DEFINED set information for each e-block. Symbols such as variables or subroutine names are assigned unique identifiers during this phase. Such identifiers, not symbolic names, are used throughout the compile phases in identifying each symbol. The local program database also contains the information needed to map the identifiers into symbolic names. In compiling several modules of a program, the compiler maintains the same unique identifier for a global name by using the *global-id database*. Whenever a new global symbolic name is encountered in compiling a module, the compiler consults with the global-id database to see if the global name has already been assigned an identifier, and uses it if already assigned. If the name has not yet been assigned an identifier, the compiler assigns a new identifier to the global symbolic name and registers it in the global-id database.

7.1.2. Second Phase (Inter-Module Analysis Phase)

The second phase, shown in Figure 7.3, is the interprocedural and shared-variable analysis phase. During this phase, the compiler builds the static call graph of the program, and computes the global used and defined sets of each e-block (the GUSED and GDEFINED sets described in Chapter 4). The static call graph built by PPD is unique in that there is one node for each e-block; the nodes in the graph correspond not only to the subroutine call sites in the program but also to loops that constitute e-blocks. During this phase, the compiler also builds the global static graphs (the post-graphs described in Chapter 4) using the static call graph, the local static graphs, and the GUSED and GDEFINED sets.

The compiler decides how to optimize the logging code using the call graph and the GUSED and GDEFINED sets. The compiler first determines ***non-eblock*** subroutines — subroutines that will not form e-blocks. Subroutines corresponding to the leaf nodes in the static call graph are typically in this class.



SU: synchronization unit

Figure 7.3.
Inter-Module Analysis Phase

However, a subroutine that either contains a loop or contains accesses to a static variable (in the C language) always forms an e-block. The compiler next identifies the *parent* e-blocks that call those non-eblock subroutines; these parent e-blocks will do the logging for the non-eblock subroutines. Finally, the compiler decides what changes need to be made to the logging code of the parent e-block. The information about the changes to the logging code is recorded in file *PPD/nudfile*. PPD/nudfile is used in the later phases.

The compiler also computes the synchronization units and the sets of shared variables read in each of them (as described in Chapter 6). This computation uses the static graphs and local program databases. The compiler then generates logging code for shared variables and identifies the places in the assembly code where the logging code should be inserted. The resulting information is also recorded in PPD/nudfile.

As described in Chapter 5, postlogs generated by the same e-block are linked in a list. "E-pointers" is an array that contains pointers to the last log entry made by each e-block and is updated during program execution. The e-pointer array needs to be written to a file when the execution terminates. The compiler generates an assembly file that will write the e-pointer array to the log file when the program terminates. *PPD/flush_ptr.s* is the assembly file that contains the code to write out the e-pointers. This file will be compiled and linked with the object code during the fourth phase. *PPD/flush_ptr.data* is the file containing the list of e-pointers written by PPD/flush_ptr.s. This file is used by the Controller, when debugging is initiated, to match each e-pointer value with the corresponding e-block.

The interprocedural analysis in the current version of PPD cannot deal with cycles in the static call graph; i.e., it cannot deal with recursive calls. The next version of PPD will be capable of dealing with cycles in the call graph; a cycle in a call graph can be handled by collapsing all the nodes of a cycle into a single node whose GUSED and GDEFINED sets are the unions of those collapsed nodes.

7.1.3. Third Phase (Single-Module Phase II)

During the third phase, shown in Figure 7.4, the compiler makes changes to the object code and the emulation package assembly files generated during the first phase. The changes to the object code files are related either to the construction of e-blocks or to generating log entries for shared variables. First, the compiler deletes logging code that is no longer needed, such as code to generate prelogs and postlogs for non-eblock subroutines. Second, the compiler identifies and modifies the logging code of the parent e-blocks that will do logging for those non-eblock subroutines. Finally, the compiler identifies the places where additional logging for shared variables is needed and inserts the proper logging code at these places.

The compiler also modifies the emulation package for the non-eblock subroutines. Figure 7.5 shows two example subroutines and their corresponding emulation package e-block code. Entry point "_sub1" is used when subroutine "sub1" is called from other e-blocks. Upon entering "_sub1", the emulation

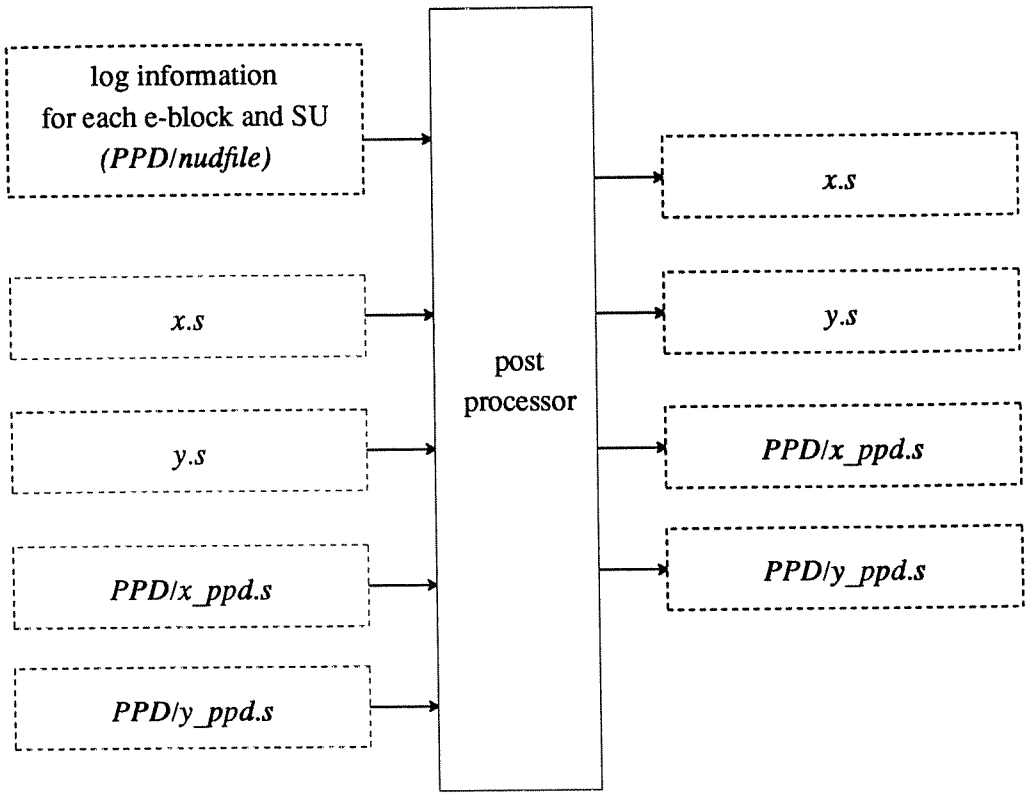


Figure 7.4.
Single-Module Phase (II)

package updates the program state with the corresponding postlog generated by "sub1" during run time. The program control then returns to the calling e-block and continues generating detailed traces of the calling e-block. Entry point "_sub1_second_entry" is used by the Controller to generate detailed traces for "sub1". If the compiler decides that "sub1" will not be an e-block, the *post processor* (see Figure 7.4) deletes the code between labels "_sub1" and "_unique_label_for_sub1". The effect is that the normal code of "sub1" will execute and generate detailed traces when entry point "_sub1" is entered from other e-blocks during debug time.

The post processor scans each assembly code file looking for the unique labels; such labels are planted into the assembly code files during the first phase. Whenever one of these labels is read, the post processor consults with the table built from PPD/nudfile to see if any modifications to the assembly file are necessary. The time complexity of this process is roughly proportional to the size of the assembly files.

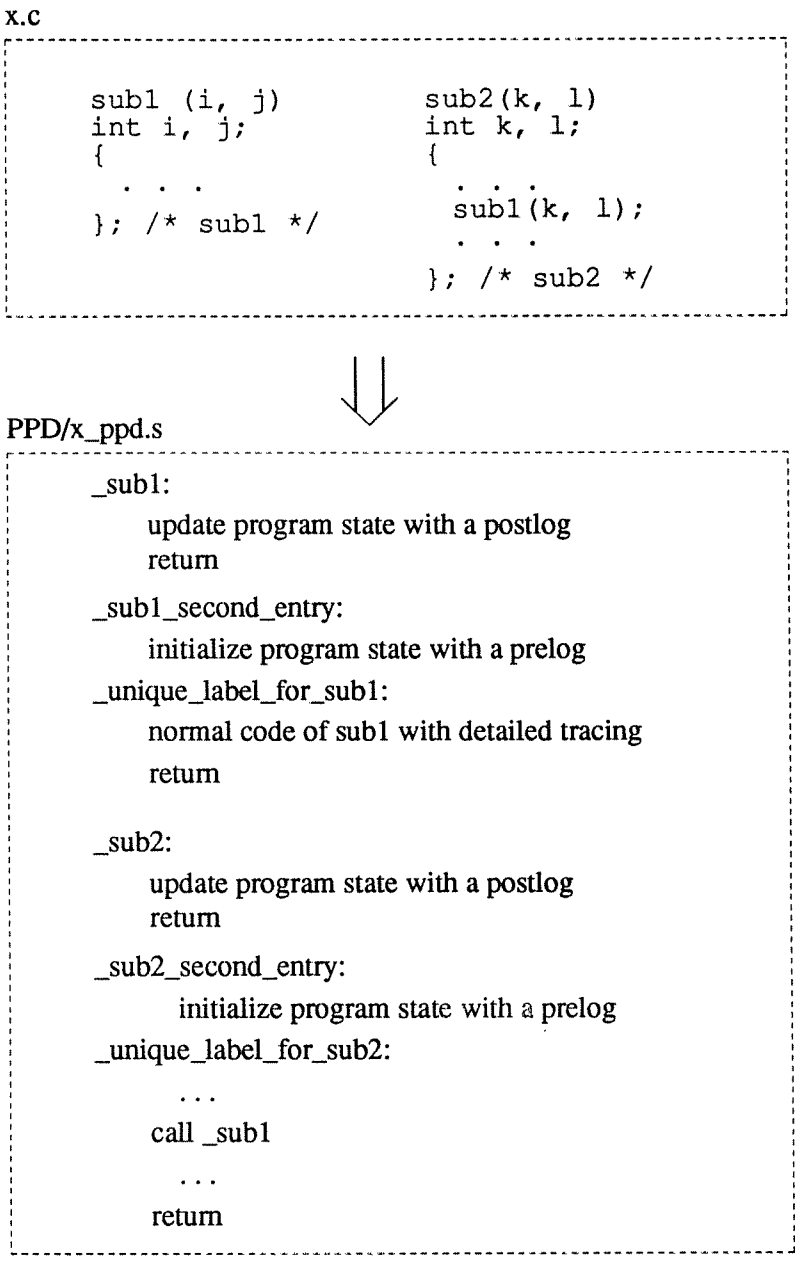


Figure 7.5.
Example Emulation Package E-blocks

7.1.4. Fourth Phase (Inter-Module Merge Phase)

The fourth phase, shown in Figure 7.6, is the inter-module merge phase. During this phase, the assembly files, generated during the first phase and modified during the third phase, are compiled into two

executables: one to be executed at run-time (the object code) and the other to be executed at debug-time (the emulation package). Also, various information generated during the previous phases is merged into a global program data base to be used by the Controller at debug-time. File "controller.o" contains the Controller routines that control the debug time execution of the emulation package.

7.2. Synchronization Edges and Semaphores

As described in Chapter 6, we construct synchronization edges in the parallel dynamic graph for semaphore operations by matching the $n'th$ V operation on a semaphore to the $(n+i)'th$ P operation on the same semaphore, where $i(\geq 0)$ is the initial value of the semaphore variable. In this section, we describe the implementation details of how we identify those matching semaphore operations.

The standard semaphore operations are P, V, and the initialization of the semaphore variable [56]. The PPD compiler must generate code to trace the information needed to match P and V operations. The information generated for each P or V operation is the type of operation, the semaphore identifier, and the serial number of the operation. The serial number of a P (V) operation on a semaphore variable is the value of a counter that gets incremented for each P (V) operation on that semaphore. There are two counters for each semaphore variable: one for P operations and the other for V operations. These counters are initialized with zero at the start of the program. The information generated for the initialization of a semaphore variable is the type of operation, the semaphore identifier, and the initial value.

Recording information about the order of execution of semaphore operations requires atomic incrementing and reading of the serial numbers. If the semaphore operations are directly supported by the operating system kernel, tracing code added to the kernel will typically be atomic. However, for the first version of PPD, we have decided to implement library routines to be used by the user instead of modifying the semaphore operations in the kernel.

The library routines use the semaphore operations supported by the target system to implement the same functions as the original semaphore operations. They also maintain counters associated with the semaphore variables to generate serial numbers for the semaphore operations, and they generate detailed traces of the operations. Using the library routines has the advantage that they are easy to build, compared with making local changes to the kernel code. However, using library routines has the disadvantage that

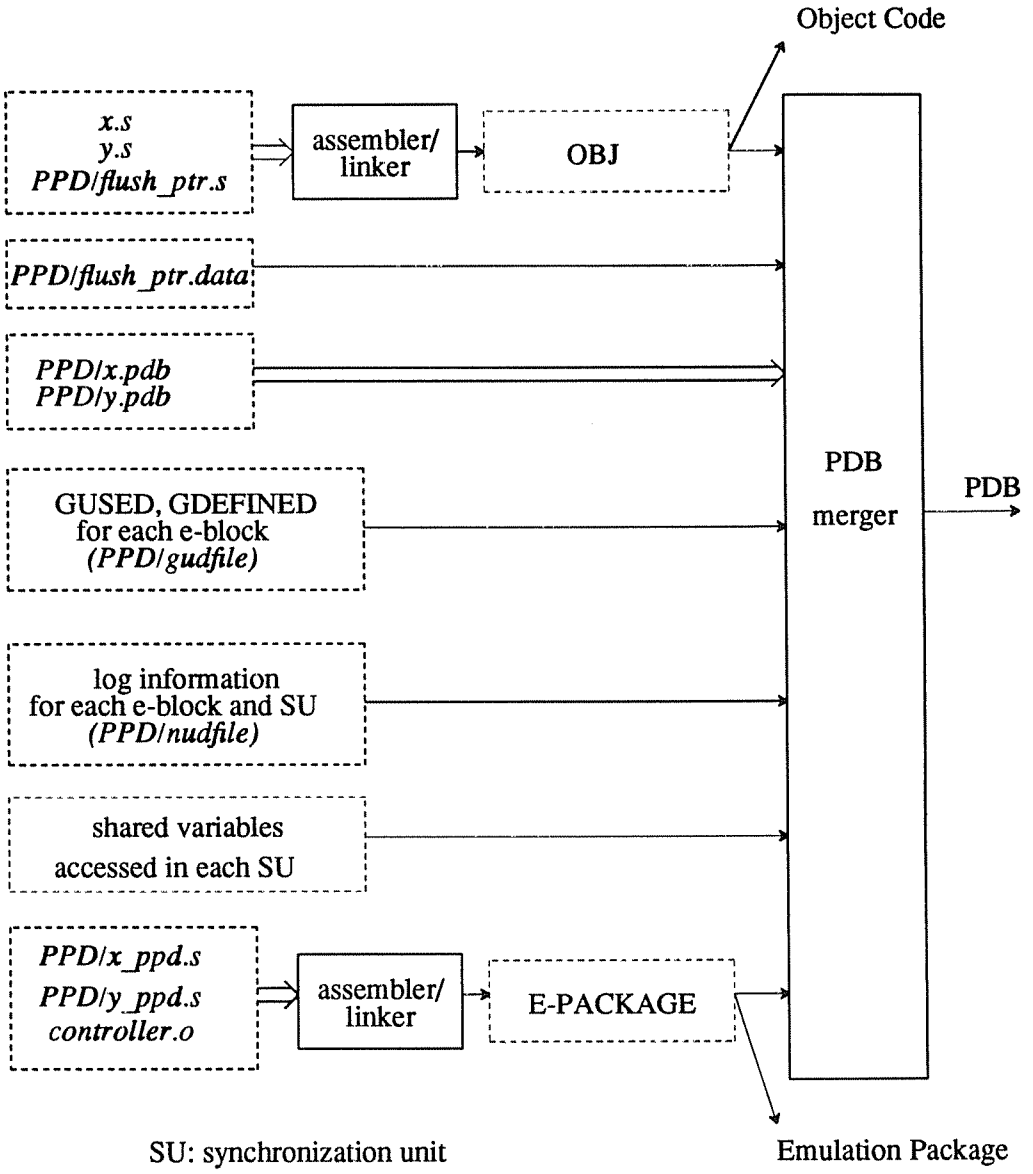


Figure 7.6.
Inter-Module Merge Phase

they need to contain extra synchronization operations to protect the counters associated with the semaphore variables.

The three library routines supplied by PPD to support the semaphore operations are *PPD_INIT*, *PPD_P*, and *PPD_V*. PPD_INIT(*init_val*) allocates a new semaphore variable and two counters — *p-counter* for P operations and *v-counter* for V operations. It also allocates two lock variables for the new semaphore variable — *p-lock* for P operations and *v-lock* for V operations. The semaphore variable is initialized with init_val, a non-negative integer, while the counters are initialized with zero. PPD_INIT returns the identifier of the new semaphore variable. PPD_INIT also generates a trace record with the type of the semaphore operation, the semaphore identifier, and the initial value.

Figure 7.7 shows the implementations of the PPD_P and PPD_V operations. The major difference (beside the P and V operations themselves) between the PPD_P and PPD_V operations is that PPD_P atomically increments the counter after the P operation while PPD_V atomically increments the counter

```
PPD_P ( sem_var )
    int counter_val;
    begin
      P ( sem_var.sem );
      lock ( sem_var.p-lock );
      counter_val = sem_var.p-counter++;
      unlock ( sem_var.p-lock );
      GenerateTrace ( P_SEM, sem_var, counter_val );
end PPD_V;

PPD_V ( sem_var )
    int counter_val;
    begin
      lock ( sem_var.v-lock );
      counter_val = sem_var.v-counter++;
      unlock ( sem_var.v-lock );
      V ( sem_var.sem );
      GenerateTrace ( V_SEM, sem_var, counter_val );
end PPD_V;
```

Figure 7.7.
Implementation of Semaphore Operations in PPD

before the V operation. All PPD_P operations on a given semaphore variable are now serialized by the synchronization operation in them added to atomically increment the p-counter of the semaphore variable. All PPD_V operations on a same semaphore variable are also serialized by the synchronization operation in them added to increment the v-counter of the semaphore variable.

7.3. Performance Measurements

This section presents measurements of the overhead caused by PPD on compile time, and on the size and execution time of application programs. We compare the performance of the PPD compiler with that of the Sequent Symmetry C Compiler in the following categories: compilation time, size of the object code, and execution time of the object code. (We have modified the Sequent Symmetry C Compiler in building the PPD compiler.) We also present measurements of the execution-time trace size. Figures 7.8–7.17 summarize the results of the comparison. We will also discuss the trade-off between execution-time efficiency and debug-time efficiency later in this section.

We used 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 $O(n^2)$ [26]. 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 to multiply pairs of scalar elements of the two matrices. The subroutine does not contain a loop or any accesses to static variables, making it a target of log optimization.

SH_PATH_1 computes the shortest paths from a city to 99 other cities using an algorithm described by Horowitz and Sahni [26]. 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 of the other cities. CLASS is a program that emulates course registration for students, such as registering for courses and dropping courses. We obtained the test results of CLASS by running it with an input file containing 130 such registration activities. CLASS also can be run as an interactive program.

The CLASS test program consists of 5 separately compiled modules (source files). Each of the other programs consists of a single module. SH_PATH_1 and SH_PATH_2 each contain a subroutine that actually computes the shortest path. This subroutine is called 10 times by the main procedures. Thus, the

actual execution times of these two programs are approximately 1/10 of the times shown in Figure 7.12. SORT contains a subroutine that sorts the array of 100 integers twice, first in ascending order, and then in descending order. The subroutine is called 10 times by the main procedure. Thus, the actual execution times of SORT are approximately 1/20 of the times shown in Figure 7.12. Some of the test programs show no difference with log optimization; all subroutines have loops or accesses to static variables in them.

7.3.1. Compilation Time

Figure 7.8 summarizes the measurements of complete (compiling to linking) compilation times. CPU time is the sum of user time and system time. In general, PPD compiler takes less than four times as long as the Sequent Compiler. To compare the compilation time of the two compilers in more detail, we measured the times spent in each step of compiling program MATRIX, as shown in Figure 7.9. For the

		Sequent Compiler	PPD compiler w/o log optimization (overhead in %)	PPD compiler w/ log optimization (overhead in %)
SORT	CPU	1.4	6.1 (272%)	6.1 (272%)
	Elapsed	2.5	9.3 (272%)	9.3 (272%)
MATRIX	CPU	1.4	5.4 (286%)	5.9 (321%)
	Elapsed	1.8	6.8 (278%)	8.4 (367%)
SH_PATH_1	CPU	2.0	7.0 (250%)	7.0 (250%)
	Elapsed	3.0	8.1 (170%)	8.1 (170%)
SH_PATH_2	CPU	1.7	6.8 (300%)	6.8 (300%)
	Elapsed	2.3	7.9 (243%)	7.9 (243%)
CLASS	CPU	5.1	15.9 (212%)	18.1 (255%)
	Elapsed	5.8	18.7 (222%)	21.7 (274%)

Figure 7.8.
Compilation Time Measurements
(time in seconds)

Sequent Compiler, we divided the compilation into two steps: The first step (row 1) is to generate an assembly file from the source file and is done by the C compiler; this step corresponds to the first phase of the PPD compiler (row 4). The second step (row 2) is to generate an executable file from the assembly file and is done by the assembler and the linker; this step corresponds to the fourth phase of the PPD compiler (row 7).

The first phase of the PPD compiler takes about twice as long as that of the Sequent Compiler, which is caused by the time spent on dependence analysis and in generating three additional files: one assembly file for the emulation package, a static dependence graph file, and a local program database file. The fact that the fourth phase of the PPD compiler takes longer than that of the Sequent Compiler can be accounted for by the time spent in generating the two executables (object code and emulation package) and in merging piecewise data structures into large, global data structures.

	compile phase	CPU	ELAPSED
Sequent Compiler	(1) ccom	0.6	0.8
	(2) assembler and linker	0.7	0.9
	(3) **total**	**1.3**	**1.7**
PPD Compiler with log optimization	(4) first phase	1.3	2.0
	(5) second phase	0.3	0.5
	(6) third phase	0.5	0.8
	(7.1) *object code assemble & link*	*1.3*	*1.4*
	(7.2) *emulation package assemble & link*	*1.3*	*1.5*
	(7.3) *global database*	*1.2*	*1.2*
	(7) fourth phase	3.8	4.1
	(8) **total**	**5.9**	**7.4**

Figure 7.9.
Details of Compiling Time for Program MATRIX
(time in seconds)

The fourth phase of the PPD compiler consists of three sub-phases: the first phase generates the object code (row 7.1), the second phase generates the emulation package (row 7.2), and the third phase merges all of the piecewise data structures into a global program database (row 7.3). Figure 7.9 shows that the times spent in each of the three sub-phases are roughly proportional to the sizes of the executable files that they produce; the sizes of the files are given in Figure 7.11.

A limitation of the current version of PPD compiler is that it repeats phases 2–4 whenever one of the source modules is modified, resulting in excess overhead for separate compilation. Figure 7.10 compares the compilation time of the entire program CLASS with the compilation times for the cases when only one of the four source modules is re-compiled. The figure shows the high overhead in re-compiling even a small module. We are currently working on *incremental semantic analysis* methods, which can identify and isolate the portion of the semantic information that needs to be updated due to changes in a

	source code size (bytes)	Compiler	CPU time	Elapsed time
CLASS (all)	5488	Sequent	4.9	5.7
		PPD	*18.1*	*21.7*
change.c	3230	Sequent	2.7	3.0
		PPD	*14.7*	*16.6*
init.c	1199	Sequent	1.7	2.0
		PPD	*13.3*	*15.1*
print_class.c	602	Sequent	1.5	1.8
		PPD	*12.9*	*15.4*
main.c	147	Sequent	1.1	1.5
		PPD	*12.6*	*14.3*

Figure 7.10.
Example Re-compilation Times of Program CLASS
(time in seconds)

given part of the program.

7.3.2. File Sizes

Figure 7.11 shows the source and object code sizes of the programs tested. It also shows the size of the program database and static graph files. The size of the executables generated by the PPD compiler are 47 – 66% larger than the executable files generated by the Sequent Compiler. In general, small programs have large proportional size increases, because the PPD compiler generates additional code (of fixed length) for procedure `main` to initialize the logging routines. MATRIX has the smallest original code size, yet it has the largest proportional increase in size (66%).

Programs with small subroutines have potentially large increases in object-code size due to the additional code to generate a prelog entry and a postlog entry for each subroutine call. However, log optimization will often reduce the logging overhead for these programs. CLASS has several small subroutines and has a relatively large proportional size increase (61%) before log optimization. However, it has a smaller size increase (47%) after log optimization.

7.3.3. Execution Time

The goal of the PPD design is to minimize execution time overhead without unduly burdening the other phases of program execution. Figure 7.12 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.

		SORT	MATRIX	SH_PATH_1	SH_PATH_2	CLASS
Source Code		1025	967	1691	1683	5488
Sequent Compiler Object Code		20016	18355	20668	20667	29574
PPD Object Code (overhead)	without log-optimization	30612 (53%)	30556 (66%)	31574 (52%)	31323 (51%)	47809 (61%)
	with log-optimization	30612 (53%)	30520 (66%)	31574 (52%)	31323 (51%)	43621 (47%)
PPD Emulation Package (overhead)	without log-optimization	39779 (99%)	39875 (117%)	44994 (118%)	44484 (115%)	58688 (98%)
	with log-optimization	39779 (99%)	39933 (118%)	44994 (118%)	44484 (115%)	58532 (98%)
Program Database		2476	2287	3421	3241	7432
Static Graph		2091	1408	3265	3221	7410

Figure 7.11.
File Sizes
(sizes in bytes)

		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 7.12.
Execution Time Measurements
(time in seconds)

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 dumping out 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 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. However, by using a more sophisticated analysis of dependences for complex data objects (described in Section 4.7), we hope to generate a smaller log entry containing only the particular row (or other part) of the matrix that is actually accessed.

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.

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.

7.3.4. Execution-Time Trace Size

Figure 7.13 shows the sizes of the execution-time traces (log) generated by the test programs. MATRIX has the largest performance improvement from using log optimization among our group of test programs. The performance of program CLASS became slightly worse with log optimization (i.e., trace

	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 7.13.
Execution-Time Trace Size Measurements
(sizes in bytes)

size increased) because the parent e-blocks generated additional log information for some non-eblock subroutines that were never invoked.

7.3.5. Trade-Off between Run Time and Debug Time

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 the debugging phase, execution phase performance will suffer. In this section, we present initial results for the cost of debug time re-execution. The results are still quite limited, and we are currently performing a more extensive set of experiments.

Figure 7.14 shows the execution times (original execution and re-execution) and debug-time trace sizes of various e-blocks from our test programs. The e-block from SORT consists of a singly-nested loop,

	Execution Time			Debug-Time Trace Size	Elapsed-Time Overhead (†)
	original CPU	Re-execution CPU	Re-execution ELAPSED		
e-block 1 (SORT)	< 0.1	0.1	1.7	0.37 Mbytes	8.9%
e-block 2 (MATRIX)	8.6	160.5	165.5	57.76 Mbytes	7.9%
e-block 3 (SH_PATH_1)	0.1	3.8	4.8	1.24 Mbytes	69.2%
e-block 4 (SH_PATH_2)	10.5	> 364.8	> 422.8	> 117.79 Mbytes	2.8%

† (from Figure 7.12)

Figure 7.14.
Execution Times and Trace Sizes
(sizes in bytes)

while the e-block from MATRIX is made of a triply-nested loop. By constructing a single e-block out of the triply-nested loop from MATRIX, we were able to reduce the execution phase overhead, but with a large debug-time overhead: 166 seconds in re-execution time and 57.76 Mbytes of debug-time trace, generating more than one million detailed trace records. For a comparison, the execution time of MATRIX itself is 13 seconds, and its execution-time trace size is 0.12 Mbytes (with log optimization). Note that we would only re-execute the log (and incur this cost) if we were interested in the details of a dependence within the loop.

The e-block from SH_PATH_1 is constructed out of a singly-nested loop that computes the shortest paths from one city to the 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. Re-execution for the e-block from SH_PATH_1 took about 5 seconds with 1.24 Mbytes of trace. Re-execution of the e-block from SH_PATH_2 terminated because of insufficient file space for the detailed traces. At that time the e-block from SH_PATH_2 had re-executed for more than 7 minutes with more than 117.79 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. The additional e-blocks could nest, potentially forming an e-block around each of the nested loops. Alternatively, it is possible to divide a section of code into contiguous e-blocks. In this case, we would only need to re-execute the e-blocks that potentially contain the dependences in which we are interested. It may also be possible to decide dynamically, at execution time, whether or not to generate logs for an e-block. This dynamic decision could be based on the amount of time already spent in a loop. The decision of how to form e-blocks affects only program performance, so we are free to change these decisions without affecting the logic of the program. Of course, there is additional overhead involved in making these dynamic decisions, so we need to experiment to see if this mechanism would indeed be beneficial.

7.4. Discussion of Measurements

In this section, we have provided performance measurements of the various parts of PPD. The measurements show the expected increases in compilation time for the PPD compiler. Also, the measurements indicate that we need to take further advantage of separate compilation. The second phase

(intermodule and shared variable analysis) of the PPD compiler does not need to be completely repeated. We are currently researching techniques to incrementally apply changes from static graphs of individual files to the global static graph. Similar techniques must be developed for the other global data structures.

The increase in execution time from generating the logs varies quite a bit for the various test programs (0–75%). However, the larger increases in execution time come from test programs that access only parts of arrays in loops. With a more sophisticated dependence analysis for complex data objects, we expect substantial improvements to be possible for such programs.

Execution-time trace sizes for our test programs were generally small (less than one Mbytes in all cases). However, the measurements show that we need more experiments to better understand the balance between the trace size during execution and the response time during debugging. Naive tracing can generate large log files, but with the addition of some basic optimizations, the size of the log files can be reduced to something quite reasonable. These optimizations must be traded-off with the cost of generating the detailed tracing during the interactive part of debugging.

In general, the performance measurements of PPD described in this section have demonstrated the feasibility of our ideas for debugging parallel programs. However, PPD is a complex system and these studies are only preliminary. We need more experiments and further research to learn how to better balance the trace size during execution and the response time during debugging.

The test programs used thus far in the performance measurements of PPD are, in general, small in size. However, we think the results obtained with these program will scale proportionally to larger programs. As features are added to our debugging system, we are extending the types and sizes of test programs that we are studying. As we gain experience, we should be able to better evaluate our optimizations and to design new ones.

Chapter 8

Conclusions and Future Research

8.1. Conclusions

In this thesis, we have described the design and implementation of a debugging system for parallel programs running on shared memory multi-processors. The goal of the debugging system is to present to the programmer a graphical view of dynamic program dependences while keeping the execution-time overhead low.

First, we have described the use of *flowback analysis* to provide information on causal relationship between events in a program's execution without re-executing the entire program for debugging. 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, we use 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. The cornerstone idea of the incremental tracing concept is to generate a coarse trace during execution and fill in incrementally, during the interactive portion of the debugging session, the gap between the information gathered in the coarse trace and the information needed to do flowback analysis.

We have also described how to extend flowback analysis to parallel programs. Flowback analysis can span process boundaries; i.e., the most recent modification to a shared 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.

As part of extending flowback analysis to parallel programs, we have described how to detect data races in the interactions between co-operating processes. By using the read and write sets of each event, obtained by semantic analysis during compile time, we can decide whether there exists any potential

read/write or write/write conflict between unordered events.

Finally, we have described the implementation of a prototype debugging system (called PPD) and provided performance measurements of the various parts of PPD. We have described the code-generation techniques for separate compilation used by the PPD compiler. In general, the performance measurements of PPD have demonstrated the feasibility of our ideas for debugging parallel programs. However, PPD is a complex system and the tests in the thesis are only preliminary. We need more experiments and further research to learn how to better balance and optimize the system.

8.2. Future Research

Through the design and prototype implementation of PPD, we have shown the feasibility of using flowback analysis and incremental tracing to build an effective and efficient parallel program debugger. There are several issues that should be investigated further. The most immediate issue is the handling of pointers and dynamic data structures. Currently, we handle pointers and dynamic objects in a manner similar to the way we handle randomly accessed arrays; we generate a special log entry for the pointer identifier, the address pointed to by the pointer, and the value of the object pointed to by the pointer. However, we need to investigate ways to reduce the potentially large amount of execution-time traces due to pointers and dynamic objects by using a method similar to [27, 38].

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.

A limitation of the current version of the PPD compiler is that it repeats the entire interprocedural and shared-variable analysis phase whenever one of the source modules is modified. We are currently working on incremental semantic analysis methods, which can identify and isolate the portion of semantic information to be updated due to changes in some part of the program.

The current implementation of PPD does not handle library routine calls with well-known side effects (such as "read" in the C language) properly. One way to approach this case is to construct an e-

block for each of the library routine calls so that the side effects can be recorded as a postlog entry. We are planning on implementing a scheme using this approach.

Performance measurements of PPD show that increase in the execution time from generating the logs varies quite a bit for the various programs (0 – 75 %). However, large increases in the execution time come from test programs that access only part of arrays in loops. We need more research on the sophisticated analysis of dependences for complex objects (as described in Chapter 4) to make the compiler smart enough to generate a smaller log entry containing only the particular parts of array that are actually accessed.

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 the debugging phase, execution phase performance will suffer. The performance measurements show that we need more experiments and research to better evaluate our current optimizations and to design new ones.

PPD is a debugger for parallel programs running on shared memory multi-processor systems. The ideas in PPD can also be used to debug programs running on a system that uses messages rather than semaphores by resolving dependences via messages. Dependences via messages can be resolved by identifying each statement (of the sender process) that modified a part of the message sent and identifying statements (of the receiver process) that read-accessed an overlapping part of the message received. We are currently investigating this issue.

Another research issue is how we can apply flowback analysis to debugging code generated by vectorizing [6] or parallelizing compilers [4,36]. In these cases, the compilers introduce additional parallelism or synchronization in the user's program to achieve concurrent execution. The basic techniques in PPD should be applicable to these compilers. Whether a synchronization operation in a program is introduced originally by the user or is introduced afterward by the compiler does not affect the basic analyses. Also, PPD relies on the dependences between statements and not on the execution order of statements. Parallelizing compilers preserve the essential dependences so that they do not change the semantics of the program. We must investigate how to map the dependences in the code generated by these compilers into the dependences perceived by the user in the original program.

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.

Appendix A

Buidling the Data Dependence Graphs

The data dependence graph is built in two steps. First, the pre-graph for each control block is built, ignoring interprocedural dependences. Then, a post-graph is built from the pre-graph.

Building the Pre-Graph

Algorithm A.1a shows an algorithm to build the pre-graph given the statements in a control block. BuildPreGraph consists of a main loop that first creates a new node and then connects that node by using calls to ConstructInEdge and ConstructOutEdge, or BuildPreWithSubrtn. ConstructInEdge and ConstructOutEdge create data dependence and linking edges for singular nodes; they are shown in

---

```
BuildPreGraph (cblock: control block);
    begin
        DEFINED(cblock) ← ∅;
        USED(cblock) ← ∅;
        foreach statement stmt in cblock in control flow order
            if stmt does not have a subroutine call then
                newnode ← new (singular node);
                foreach variable vused used at stmt
                        /* in Algorithm A.1b */
                    ConstructInEdge (vused, newnode,
                        cblock, DATA_DEP_EDGE);
                end for
                vdefined ← the variable written at stmt;
                    /* in Algorithm A.1b */
                ConstructOutEdge (newnode, vdefined, cblock);

            else /* stmt has a subroutine call */
                newnode ← new (sub-graph node);
                    /* in Algorithm A.1d */
                BuildPreWithSubrtn (newnode, stmt, cblock);
            end if
        end for
end BuildPreGraph;
```

Algorithm A.1a. An Algorithm to Build DDG

Algorithm A.1b. In this description of the algorithm, we assume that there is only one *l-value* for each statement, so ConstructOutEdge is called only once for each statement while ConstructInEdge is called once for each variable used in the statement. BuildPreWithSubrtn deals with subroutine calls and is described in Algorithm A.1d. In practice, the main loop of BuildPreGraph is embedded in the parser of the compiler.

ConstructInEdge (in Algorithm A.1b) creates incoming edges for a new node. It uses LookupNode

---

```
ConstructInEdge (vused: variable; newnode: node;
        cblock: control block; edgetype: edge type);
    begin
        if vused is not an array then
                /* in Algorithm A.1c */
            lastnode ← LookupNode (vused, cblock);
            create an edge of edgetype out of lastnode and into newnode;
        else
            selectnode ← new (singular node);
            create an edge of edgetype out of selectnode and into newnode;
            lastnode ← LookupNode (vused, cblock);
            create a linking edge out of lastnode and into selectnode;
                /* in Algorithm A.1c */
            ConstructIndexNodes (selectnode, vused, cblock);
        end if
end ConstructInEdge;


ConstructOutEdge (newnode: node; vdefined: variable; cblock: control block);
    begin
        if vdefined ∈ DEFINED(cblock) then
            delete the edge into the DEFINED(cblock) entry for vdefined;
        else    /* vdefined has not been defined in this block yet */
            DEFINED(cblock) ← DEFINED(cblock) ∪ vdefined;
        end if
        if vdefined is an array then
            lastnode ← LookupNode (vdefined, cblock);
            create a linking edge out of lastnode and into newnode;
                /* in Algorithm A.1c */
            ConstructIndexNodes (newnode, vdefined, cblock);
        end if
        create a data dependence edge out of newnode
          and into the DEFINED(cblock) entry for vdefined;
end ConstructOutEdge;
```

Algorithm A.1b. An Algorithm to Build DDG (continued)

(in Algorithm A.1c) to locate the most recent node that writes the variable used in the statement. ConstructInEdge then creates a dependence edge out of the located node and into the new node. If the variable used is an array element, ConstructInEdge first creates a select node for the read from the array, then creates a dependence edge out of the select node and into the new node, and finally creates a linking edge out of the located node and into the select node. ConstructInEdge then goes on and creates index nodes for the variables used as the indices for the read array and creates edges out of the index nodes into the select node. This final step is done by ConstructIndexNodes in Algorithm A.1c.

ConstructOutEdge (in Algorithm A.1b) inserts an entry for the newly written variable in the DEFINED set of the control block. If the written variable is already in the set, the routine deletes the existing dependence edge into the DEFINED entry for that variable. If the written variable is an array element, ConstructOutEdge first locates the most recent node that writes to the same array, by calling LookupNode, and then creates a linking edge out of the located node into the new node. ConstructOutEdge then creates index nodes for the variables used as the indices for the write to the array,

---

```
LookupNode (thisvar: variable; cblock: control block) : node;
    begin
        if thisvar ∈ DEFINED(cblock) then
            returnnode ← node that is connected by an
            edge into the DEFINED(cblock) entry for thisvar;
        else    /* thisvar has not been written in this block yet */
            USED(cblock) ← USED(cblock) ∪ thisvar;
            returnnode ← USED(cblock) entry for thisvar;
        end if
        return (returnnode);
end LookupNode;

ConstructIndexNodes (newnode: node; varray: variable; cblock: control block);
    begin
        foreach vindex used as an index of varray
            indexnode ← new (singular node);
            label indexnode with "%" and index position;
            create a data dependence edge out of indexnode and into newnode;
            ConstructInEdge (vindex,
              indexnode, cblock, DATA_DEP_EDGE);
        end for
end ConstructIndexNodes;
```

Algorithm A.1c. An Algorithm to Build DDG (continued)

---

and creates edges out of the index nodes into the new node, by calling ConstructIndexNodes. ConstructOutEdge always creates a data dependence edge from the new node to the DEFINED entry for the written variable.

Algorithm A.1d describes BuildPreWithSubrtn, which builds the data dependence graph of a statement in the presence of a subroutine call. BuildPreWithSubrtn is called by BuildPreGraph in Algorithm A.1a. The presentation of Algorithm A.1d is simplified to assume that each statement has at most one subroutine call. It also excludes the possibility that some input parameters can be expressions rather than a single variable, and it also assumes that a statement with a function call has a simple expression consisting of only a target variable (for the returned value) and a function call. However, such restrictions can be easily removed with small modifications to the algorithm. For example, we can deal

```
BuildPreWithSubrtn (subnode: sub-graph node; stmt:statement;
            cblock : control block);
    begin
        foreach input parameter inparm of the subroutine call
            parmnode ← new (singular node);
            label parmnode with "%" and the parameter position of inparm;
            ConstructInEdge (inparm, parmnode,
                cblock, DATA_DEP_EDGE);
            create a data dependence edge out of parmnode
                and into subnode;
        end for
        foreach global variable gvar in DEFINED(cblock)
            ConstructInEdge (gvar, subnode, cblock, LINKING_EDGE);
        end for
        if stmt is a function call then
            targetnode ← new (singular node);
            label targetnode with "%0";
            create a data dependence edge out of subnode
                and into targetnode;
            ConstructOutEdge (targetnode,
                target variable of function call, cblock);
        end if
        foreach global variable gvar in DEFINED(cblock)
            /* assume it will be defined by the subroutine*/
            ConstructOutEdge (subnode, gvar, cblock);
        end for
end BuildPreWithSubrtn;
```

Algorithm A.1d. An Algorithm to Build DDG in the Presence of Subroutine Calls

with expression parameters by treating the expressions as if they were assignment statements to the nodes representing the parameters. We can deal with a function call that is a part of an expression assigned to a variable by creating a data dependence edge out of the node labeled with "%0" to the target node of the assignment.

The branch dependence graph and the pre-graph form of the data dependence graph are built at compile time while the parse tree is being built. The cost for building these graphs should be dominated by the cost to build the data dependence graph. In building a data dependence graph of a control block, we first locate, for a new node, nodes upon which there exist dependences. We then create edges out of these nodes into the new node. Locating the nodes requires two operations for each variable used as an r-value in the expression corresponding to the new node: First we look into the DEFINED set of the control block for the variable, and then we either follow the edge backward or else look into the USED set. We can bound the time requirement of the two operations by a small constant even for large USED and DEFINED sets by using a hash table to store each set. For a control block with n nodes, the worst case time complexity of building a data dependence graph is thus $O(n^2)$, which is the actual complexity when each node has incoming dependence edges from all of the preceding nodes in the block. However, in practice, we can expect the number of variables used as an r-value in an expression to be bounded by some constant, effectively yielding $O(n)$ time for building the data dependence graph.

Building the Post-Graph

Building a post-graph from a pre-graph requires the knowledge of interprocedural summary information. Algorithms A.2a and A.2b show an algorithm to build the post-graph with interprocedural information. Algorithm A.2a consists of two main loops. During the first main loop, the routine examines the GUSED set of each sub-graph node, obtained as summary information by interprocedural analysis, in the order of control flow in *cblock*. For each variable in the GUSED set of a sub-graph node, the routine creates a new data dependence edge and modifies the graph accordingly.

During the second main loop, the routine examines the data dependence edges out of each sub-graph node and the GDEFINED set of the sub-graph node in the reverse order of control flow in *cblock*. The routine does two jobs during this loop. First, it takes care of the data dependence and linking edges for

each variable that was assumed to be defined by the sub-graph node but that are not in the GDEFINED set. Second, it takes care of the opposite case: the variables that were not assumed to be defined by a sub-graph node but that are in the GDEFINED set. GdNotAssumed in Algorithm A.2b handles this last case.

```
BuildPostGraph (cblock: control block);
    begin
        /* When this routine terminates, GU and
         * GD will contain GUSED(cblock) and
         * GDEFINED(cblock), respectively */
        GU ← USED(cblock);
        GD ← DEFINED(cblock);
        foreach sub-graph node subnode in the order of
          control flow in cblock
            foreach global variable gvar1 in GUSED(subnode)
                if there is a linking edge of gvar1 into subnode then
                    replace the edge with a data dependence edge;
                else
                    GU ← GU ∪ gvar1;
                    create a data dependence edge out of the GU entry
                      for gvar1 into subnode;
                end if;
            end for
        end for
        /* First, handle variables assumed to be written by subnode
         * but that are not in the GDEFINED set. */
        foreach sub-graph node subnode in the reverse order
          of control flow in cblock

            foreach data-dep or linking edge of a global variable gvar2
              out of subnode
                if gvar2 is not in GDEFINED(subnode) then
                  /* by-pass subnode */
                    disconnect any edges of gvar2 going
                      out of subnode and re-connect their tails to
                      the node out of which the dependence edge
                      of gvar2 into subnode comes;
                    disconnect any linking edge of gvar2 coming
                      into subnode;
                end if
            end for
            /* Second, handle variables not
             * assumed to be written by subnode
             * but that are in the GDEFINED set. */
            foreach global variable gvar3 in GDEFINED(subnode)

                if there is no data dependence edge of gvar3
                  out of subnode then
                    /* We have to modify the data
                     * dependence graph of this block */
                    GdNotAssumed (gvar3, GU, GD, subnode);
                end if
            end for
        end for
        foreach entry guentry in GU
            delete linking edge coming out of guentry
              and going into a sub-graph node;
            if no edge is connected to guentry then
                remove guentry from GU;
```

```
            end if
            end for
            GUSED(cblock) ← GU;
            GDEFINED(cblock) ← GD;
end BuildPostGraph;
```

Algorithm A.2a. An Algorithm to Add Interprocedural Summary Information to the Data Dependence Graph

```
GdNotAssumed (gvar3: variable; GU, GD: set of variable;
  subnode: sub-graph node);
    begin
        if gvar3 is not in GD then
            /* i.e., gvar3 has not been yet written in this block */
            GD ← GD ∪ gvar3;
            create a data dependence edge out of subnode into
              GD entry for gvar3;
            if gvar3 is not in GU then
                GU ← GU ∪ gvar3;
            end if
            create a linking edge out of GU entry for gvar3
              into sub-node;
        else
            if gvar3 is not in GU or
              GU entry for gvar3 is not connected
              with any linking edge then
                /* This is the first sub-graph node we meet in
                 * this phase that defines gvar3 */
                GU ← GU ∪ gvar3;
                create a linking edge out of GU entry for
                  gvar3 into subnode;

            else
                locate lastnode by following the linking
                  edge out of GU entry for gvar3;
                if there is no data dependence edge of gvar3
                  between subnode and lastnode then
                    create a linking edge out of subnode into lastnode;
                end if
                delete the linking edge going into lastnode for gvar3;
                create a linking edge out of GU entry for
                  gvar3 into subnode;
            end if
        end if
        /* Now, check if any node after subnode uses gvar3.
         * If so, the data dependence edges
         * from gvar3 in GU to such nodes must
         * be cut and reconnected to the subnode. */
        foreach data dependence edge u-edge out of gvar3 in GU

            u-node ← the node into which edge u-edge is connected;
            if u-node is after subnode then
                cut the tail of u-edge and reconnect it to subnode
            end if
        end for
end GdNotAssumed;
```

Algorithm A.2b. An Algorithm to Add Interprocedural Summary Information to the Data Dependence Graph (continued)

We can get interprocedural summary information in time $O(p^2+pn)$ steps by the algorithm described in [17] for a program with p subroutines and n call sites.[1] For reasonably sized GUSED and GDEFINED sets, as with the pre-graph algorithm, we expect $O(n)$ time steps in building the post-graph (given the interprocedural information).

1. If bit vectors for interprocedural analysis were small enough to fit in a word, the complexity would be $O(p+n)$.

REFERENCES

[1] H. Agrawal and E. Spafford, "An Execution Backtracking Approach to Program Debugging," *SERC-TR-22-P*, Software Engineering Research Center, Purdue University, (August 1988).

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

[3] J. Ichbiah, et al., "Reference Manual for the Ada Programming Language," *Technical Report, United States Department of Defense*, (July 1980).

[4] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *IBM T. J. Watson Research Center Tech Rep RC 13115*, (1987).

[5] J. R. Allen and K. Kennedy, "PFC: A Program to Convert FORTRAN to Parallel Form," *TR 82-6, Dept. of Math. Sciences*, Rice University, Houston, Texas, (March 1982).

[6] 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).

[7] T. R. Allen and D. A. Padua, "Debugging Fortran on a Shared Memory Machine," *Proc. of the 1987 International Conf. on Parallel Processing*, pp. 721-727 (August 1987).

[8] F. Baiardi, N. De Francesco, and G. Vaglini, "Development of a Debugger for a Concurrent Language," *IEEE Trans. on Software Engineering* **SE-12**(4) pp. 547-553 (April 1986).

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

[10] Utpal Banerjee, "Data Dependence in Ordinary Programs," *M.S. Thesis*, University of Illinois at Urbana-Champaign, (1976).

[11] J.P. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables," *Proc. of the SIGPLAN 79 Symposium on Principles of Programming Languages*, pp. 29-41 San Antonio, TX, (January 1979).

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

[13] P. C. Bates and J. C. Wileden, "EDL: A Basis for Distributed System Debugging Tools," *Proc. of the 15th Hawaii Int'l Conf. on Systems Science*, pp. 86-93 (1982).

[14] P. C. Bates and J. C. Wileden, "High Level Debugging of Distributed Systems: the Behavioral Abstraction Approach," *J. Systems and Software* **4**(3) pp. 255-264 (December 1983).

[15] B. Bruegge and P. Hibbard, "Generalized Path Expressions: A High Level Debugging Mechanism," *Proc. of the SIGSOFT/SIGPLAN Symp. on High-Level Debugging*, pp. 34-44 Pacific Grove, Calif., (August 1983).

[16] D. R. Cheriton and W. Zwaenepoel, "The Distributed V kernel and its Performance for Diskless Workstations," *Proc. of the 9th SOSP, Operating Systems Review* **17**(5) pp. 129-140 (November 1983).

[17] K. Cooper and K. Kennedy, "Interprocedural Side-Effect Analysis in Linear Time," *Proc. of the SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation*, pp. 57-66 Atlanta, Georgia, (June 22-24, 1988).

[18] 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).

[19] R. Curtis and L. Wittie, "BUGNET: A Debugging System for Parallel Programming Environment," *Proc. of the 3rd International Conf. on Distributed Computing Systems*, pp. 394-399 Hollywood, FL, (October 1982).

[20] E. W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Comm. of ACM* **18**(8) pp. 453-457 (1975).

[21] E. A. Emerson and J. Y. Halpern, ""Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic ," *Journal of the ACM* **33**(1) pp. 151-178 (January 1986).

[22] S. Feldman and C. Brown, "IGOR: A System For Program Debugging via Reversible Execution," *Proc. of ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices* **24**(1) pp. 112-123 (January 1989).

[23] 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).

[24] P. K. Harter, Jr., D. M. Heimbigner, and R. King, "IDD: An Interactive Distributed Debugger," *Proc. of the 5th International Conf. on Distributed Computing Systems*, pp. 498-506 Denver, (May 1985).

[25] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM* **17**(10) pp. 549-557 (October 1974).

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

[27] 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).

[28] S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," *Proc. of the SIGPLAN 88 Symposium on Principles of Programming Languages*, pp. 133-145 San Diego, CA, (January 1988).

[29] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *Proc. of the ACM SIGPLAN 1988 Conf. on Prog. Lang. Design and Implementation*, Atlanta, Georgia, (June 1988).

[30] 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).

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

[32] B. Korel and J. Laski, "STAD — A System for Testing and Debugging: User Perspective," *Proc. of the Second Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, Canada, (July 19-21 1988).

[33] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pp. 207-218 Williamsburg, Va., (January 26-28 1981).

[34] 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).

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

[36] M. Lam, "Software Pipelining: An Effective Scheduling Technique For VLIW Machines," *Proc. of the ACM SIGPLAN 1988 Conf. on Prog. Lang. Design and Implementation*, pp. 318-328 Atlanta, Georgia, (June 1988).

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

[38] 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).

[39] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. on Computers* **C-36**(4) pp. 471-482 (April 1987).

[40] Y. Lichtenstein and E. Y. Shapiro, "Concurrent Algorithmic Debugging," *Proc. of ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices* **24**(1) pp. 248-260 (January 1989).

[41] B. P. Miller and J. D. Choi, "Breakpoints and Halting in Distributed Programs," *Proc. of the 8th International Conf. on Distributed Computing Systems*, pp. 316-323 San Jose, CA, (June 1988).

[42] 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).

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

[44] E. Myers, "A precise inter-procedural data flow algorithm," *Proc. of the SIGPLAN 81 Symposium on Principles of Programming Languages*, pp. 219-230 Williamsburg, VA, (January 1981).

[45] 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).

[46] D. Pan and M. Linton, "Supporting Reverse Execution for Parallel Programs," *Proc. of ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices* **24**(1) pp. 124-129 (January 1989).

[47] A. Pnueli, "The Temporal Logic of Programs," *Proc. of the 18th Annual Symposium on Foundations of Computer Science*, pp. 46-57 IEEE, (1977).

[48] B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys* **10**(2) pp. 123-165 (June 1978).

[49] N. Rescher and A. Urquhart, *Temporal Logic*, Springer Verlag (1971).

[50] C. Ruggieri and T. P. Murtagh, "Lifetime Analysis of Dynamically Allocated Objects," *Proc. of the SIGPLAN 88 Symposium on Principles of Programming Languages*, pp. 285-293 San Diego, CA, (January 1988).

[51] R. D. Schiffenbaur, "Interactive Debugging in a Distributed Programs," *M.S. Thesis, EECS Tech Report MIT/LCS/TR-264*, M.I.T., (August 1981).

[52] E. Schonberg, "On-The-Fly Detection of Access Anomalies," *Proc. of the ACM SIGPLAN 1989 Conf. on Prog. Lang. Design and Implementation*, pp. 285-297 Portland, Oregon, (June 1989).

[53] Robert Sedgewick, *Algorithms*, Addison-Wesley (1983).

[54] *Sequent C Compiler User's Manual*, Sequent Computer Systems, Inc. (1985).

[55] E. Y. Shapiro, "Algorithmic Program Debugging," Ph. D. Thesis (ACM Distinguished Dissertation, 1982), Yale University (May 1982).

[56] A. Silberschatz and J. L. Peterson, *Operating System Concepts (Alternate Edition)*, Addison Wesley (June 1988).

[57] R. Snodgrass, "Monitoring Distributed Systems: A Relational Approach," *Ph. D. Thesis*, Carnegie-Mellon University, (December 1982).

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

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

[60] M. Weiser, "Programmers Use Slices When Debugging," *Communications of the ACM* **25**(7)(July 1982).

[61] Michael J. Wolfe, "Optimizing Supercompilers for Supercomputers," *Ph.D. Thesis*, University of Illinois at Urbana-Champaign, (1982).

[62] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pp. 63-76 Austin, Texas, (November 1987).