Slicing Concurrent Java Programs: Issues and Solutions

Thesis

submitted in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

by

Mangala Gowri Nanda

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY

2001

Approval Sheet

The thesis entitled

Slicing Concurrent Java Programs:

Issues and Solutions

by

Mangala Gowri Nanda

is approved for the degree of

Doctor of Philosophy.

Examiners

Dr. Frank Tip, IBM, T. J. Watson Research Center

Prof. Rajib Mall, Department of Computer Science, IIT Kharagpur

Guide

Prof. S. Ramesh

Chairman

Prof. S. Biswas

Date : November 5, 2001

To Dinesh

ACKNOWLEDGEMENTS

I would like to thank everyone who supported and assisted me during my doctoral studies at the Indian Institute of Technology, Bombay. First and foremost, I would like to thank my advisor, Professor S. Ramesh for his encouragement and guidance during my Ph.D. program. He allowed me to pursue my own research interests, while making sure I stayed on track. He was always available to discuss various problems and provide feedback. I would like to thank Professor S. Biswas, Professor G. Sivakumar and Professor A. Sanyal for serving on my thesis committee and for their advice and support. I would also like to thank Dr. Sundeep Oberoi of Tata Infotech Ltd. for helping me to get started on my Ph.D. and his valuable encouragement and support in the initial stages.

I would like to gratefully acknowledge the IBM, India Research Laboratory for providing financial support during the course of my graduate program. In particular, I would like to acknowledge the support I received from Alok Aggarwal, Manoj Kumar and Sugata Ghosal. I received much encouragement from many people at IRL and I will always cherish my friendship with Rema Ananthanarayanan, Parul Mittal, Manika Kapoor, Lakshmi Ramchandran, Bibha Tripathi and Johara Shahabuddin. Thanks also to Pradeep Dubey, Pradeep Varma, Ashish Kundu and Rajkishore Barik, who reviewed some parts of my work and to Sudipta Maitra, Sujatha Dharmarajan, Sajita Vijayan and Bharthi Chawla who made my stay at IRL most comfortable.

I spent some pleasant Friday afternoons discussing various algorithms with Professor S. Arun-Kumar and Professor Sanjiva Prasad of the Indian Institute of Technology, New Delhi. I would like to thank them for their advice and guidance.

Many people have contributed to this thesis in one way or another. I thank Sachin Chitnis for introducing me to the subject of slicing and many interesting discussions that helped me choose my research topic. I thank Usharani, Purandar Bhaduri and Vasudev Kamath for their constant encouragement and many philosophical discussions and advice that helped me through some of the more difficult phases of my Ph.D. Purandar also read many drafts of my thesis and his comments, his support and his advise have indeed been very useful.

A very special thanks to Mr. Chandran at the Department of Computer Science and Engineering, IIT Bombay, for all that he did for me.

I thank all my family members, my husband Dinesh, my children Agastya and Samvida, Dinesh's parents Chander Kanta and J. N. Nanda, my parents Jayalakshmi and P. V. Indiresan, my sister Nitya, her husband Raghu and children Soumya, Sumana and Nandini, and my brother Atri, his wife Rohini and daughter Shruti for their love, patience and understanding over the long years of my Ph.D.

I thank my parents-in-law who were always there when I needed them. I thank them for their patience and support.

I thank my parents for their love and guidance. Their constant encouragement, support and advice has seen me through many difficult stages. I also thank my mother for proof reading my thesis and many constructive comments. I thank Atri for his continuous support, advice and guidance right through my Ph.D. I thank Nitya for always being there when I needed her to run errands, take care of my children, provide dinner for the entire family or just some moral support.

To my children, I owe an eternal debt of gratitude for their patience and understanding when I worked late and they had to fend for themselves. They shared their computer with me, suffered endless days of junk food and accepted gracefully my inability to spend more time with them.

I thank Dinesh for his love, encouragement and complete support. In the final analysis, it was his belief in me that kept me going. If it had not been for him, I would never have either begun or finished this Ph.D.

Contents

List of Figures

List of Tables

ABSTRACT

Slicing is a program transformation that is useful in debugging, testing, program maintenance, and applications that involve understanding program behavior. The *slice* of a program with respect to a program point p and a variable x, consists of all statements and predicates of the program that might affect the value of x at p.

The program dependence graph (PDG) is a representation of programs where the nodes, representing assignments and predicates in the program, are connected by control and data dependence edges. For sequential intraprocedural programs, the slice can be defined with respect to a node in the PDG as the set of all the nodes on which the given node is directly or transitively dependent. Thus, given the PDG, the slice can be computed by a simple reachability algorithm.

In contrast, mere reachability produces imprecise slices in interprocedural programs as not all paths in an interprocedural program are *realizable* paths. A *realizable* path is one that corresponds to a legal call/return sequence wherein a procedure returns to the site from where it was invoked.

Concurrent programs have additional dependences called *interference dependence*. A node S_1 is interference dependent on a node S_2 if S_2 defines a variable v, S_1 uses the variable v, and S_1 and S_2 execute in parallel. In concurrent intraprocedural programs also a simple reachability algorithm produces imprecise slices since interference dependence is not transitive.

Slicing concurrent inter-procedural programs raises the problems of both realizable paths and intransitivity of interference dependence.

This dissertation identifies the various issues in slicing concurrent intraprocedural and concurrent interprocedural programs and then gives solutions to these problems. We show how to generate *context-sensitive* slices for concurrent interprocedural programs. We have implemented the algorithms on the concurrent language, Java, and tested them on several benchmark programs.

Chapter 1

INTRODUCTION

The race is on to find chips of the future. "A sub-microscopic quantum world could dominate IT technology in the 21st century" (Financial Times, December 1, 1999). *"Scientists believe that quantum technology will take over when classical electronics reaches the physical limits of miniaturization. In a conventional computer, digital information is stored in the form of a "bit" composed of either a 0 (off) or 1 (on). Quantum bits of data, known as a qubits, are both 0 and 1 at the same time, as a result of superposition. This means that when several qubits are linked together, a quantum device can carry out many operations at the same time. As a result eight qubits, for example, would allow 256 parallel calculations. Because of their parallel computing ability, even relatively small quantum computers could, at least in theory, carry out billions of times more operations than any supercomputer based on conventional electronics. Whether it will be possible in practice to build such a device is still the subject of heated debate. What is more, as IBM's Tom Thesis points out, writing the programs for such a device could be just as much of a challenge.*

Meanwhile explicitly concurrent programs have already entered the programming mainstream.

The interest in concurrent programming has grown dramatically in recent years. Ada, Modula II and Java have built-in features for concurrency. Other languages such as C, C++ and FORTRAN have been extended in a variety of ways in order to support concurrent programming. The added complexity of expressing concurrency has made analysis, understanding and debugging concurrent programs an order of magnitude harder than analysis of sequential programs.

Dynamic analysis of concurrent programs have inherent problems. Any attempt to observe the behavior of a program may contribute to the difficulty of reproducing the erroneous behavior [62]. Any additional print or debugging statements may modify a crucial race, lowering the probability that the interesting behavior occurs. This has been referred to as the "Heisenberg Uncertainty" principle applied to software or the "Probe Effect".

Under such circumstances, static analyses can be very useful. Static analysis is useful in detecting two classes of errors in concurrent programs: *synchronization* errors and *data-usage* errors. Synchronization errors include such things as deadlock and wait forever. Data-usage errors include the usual sequential data usage errors, such as reading an uninitialized variable, and concurrent data-usage errors typified by two processes simultaneously updating a shared variable. **Static slicing** is a program analysis tool that is useful for detecting the second class of errors.

Static slicing is a program transformation that extracts a subset of a program based on some specified criterion. The transformed program is not only much smaller in size than the original, it also makes explicit the dependencies between the statements of the program and helps in understanding and debugging the program. It is useful in identifying the parts of the code that need to be tested for a specific criterion and has been used as an aid in program maintenance, reverse engineering and verification.

Why Java ? In 1995 Sun introduced a new programming language called Java. Sun describes Java as a "simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded and dynamic language". Java has become the standard language for sending platform independent programs over the Internet and executing them as part of Web pages.

With the advent of programming languages such as Java, concurrent programming languages are becoming more ubiquitous and more programmers are using concurrent constructs explicitly – an area that used to be the domain of operating system programmers. An area that is increasingly in need of tools that aid in understanding and debugging programs.

Research Outline When slicing a program, there are two issues – *correctness* and *precision*. Given an algorithm that computes a slice S of a program for a given criterion, p, we define the correctness of the slice as

Definition 1 *The computed slice S is correct if $\mathcal{S}(p) \subseteq S$.*

and we define precision as

Definition 2 *Given two correct slices S_1 and S_2 for a slicing criterion p, S_1 is more precise than S_2 if $S_1 \subseteq S_2$*

Ideally, one would like to compute the correct slice that is the most precise. But this is, in general, not computable. Hence, we try to compute a slice that is correct and as precise as possible.

Our work in slicing concurrent Java programs examines how synchronization, communication, and non-determinism along with traditional control and data dependence affect values at certain program points.

Since the Java concurrency model is quite complex, we first examine a much simpler model of concurrency based on shared memory, where parallelism is expressed by the classical `cobegin-coend` construct and there is no explicit synchronization. This helps us to identify some of the issues in static slicing of concurrent programs. We give solutions to these problems, starting with intraprocedural analysis and then extending the solutions to interprocedural analysis. The Java concurrency model has, in addition to shared memory, explicit synchronization expressed by the `wait/notify` construct, and monitors. Also Java's `fork/join` parallelism differs from the `cobegin-coend` parallelism. We identify issues specific to the analysis of concurrent Java programs and give solutions for some of them.

1.1 Major Contributions of the Dissertation

A slice is *correct* if it includes, at least, all the nodes that affect the criterion. Given two correct slices, one slice is more precise than the other if it contains less nodes. In the extreme, the complete program is always a correct slice, albeit possibly imprecise. This major emphasis of this dissertation is development of correct *and* precise slicing algorithms for concurrent intraprocedural and concurrent interprocedural programs.

1.1.1 Slicing Concurrent Intraprocedural Programs

For the basic shared-memory concurrency model based on `cobegin-coend` parallelism, we give a precise intraprocedural slicing algorithm. We show that when slicing concurrent programs, it is necessary to differentiate between loop independent and loop carried data dependences and show how to construct the concurrent program dependence graph. Unlike previous solutions [53], our algorithm gives precise slices even in the presence of nested threads and threads nested within loops. Though the complexity of this algorithm is exponential on the number of threads, a number of optimizations are suggested. Using these optimizations, we are able to get near linear behavior for many practical concurrent programs.

1.1.2 Detecting Interprocedural Realizable Paths

Determining whether there is a realizable path between any *two* nodes in the control flow graph of an interprocedural program is possible using standard techniques of data flow analysis for interprocedural programs [18, 19]. We introduce the notation $Reach(i, j)$ to be **true** if there is a realizable path from i to j and **false** otherwise. In intraprocedural programs, the *Reach* function is transitive. $Reach(a, b)$ **and** $Reach(b, c)$ implies $Reach(a, c)$. In interprocedural programs this is no longer true.

Given an interprocedural control flow graph, G, and a set of nodes n_1, n_2, $\cdots$, n_k belonging to G we give an algorithm to determine whether there is a realizable path $n_1 \xrightarrow{+} n_2 \xrightarrow{+} \cdots \xrightarrow{+} n_k$.

This realizable path problem differs from other realizable paths problems in literature. In *context-sensitive* interprocedural data flow analysis, realizable paths are identified (using a technique such as the matched parenthesis technique described by Reps [75]) and then data flow analysis is performed along only valid paths. Our realizable path problem is the *converse* of this situation. Here we are given a set of nodes and we wish to identify whether they form a part of a realizable path. A naive solution would generate all possible realizable paths in the graph (using a technique such as the matched-parenthesis technique) and then check if the given set of nodes forms a subsequence of any one of the realizable paths.

Our solution uses the interval-based approach and is a practical, incremental algorithm.

1.1.3 Slicing Concurrent Interprocedural Programs

For the basic shared-memory concurrency model of concurrent programs, we give a correct but context-insensitive interprocedural slicing algorithm which is very efficient and a correct and context-sensitive algorithm that has a worst case exponential complexity. The analysis handles nested threads with high precision but threads nested within loops are handled conservatively.

1.1.4 Data Dependence Analysis of Concurrent Java Programs

For slicing Java programs, we need to determine the data dependence edges. In Java all objects are allocated from a garbage-collected heap. Determining data dependence in the presence of heap allocated pointer variables is comparatively complicated as different access paths may refer to the same memory location. For intra-procedural data dependence analysis we give a new context-sensitive and path-sensitive solution to combined escape analysis, pointer analysis and data dependence analysis which uses a *sparse* representation. Our algorithm also builds the Static Single Assignment form of each procedure and generates the SSA def-use edges. For inter-procedural analysis we show how to use escape analysis to summarize the points-to information of a procedure so that it can be reused at each call site. Finally, we extend the analysis to handle programs with threads, monitors and synchronization.

1.1.5 Slicing Concurrent Java Programs

There are many issues in slicing Java programs pertaining to the `fork/join` concurrency construct, `wait/notify` synchronization and monitors. We discuss these issues and give solutions to some. We show how to slice concurrent Java programs. Although the complexity of the algorithm is exponential in the number of threads, we give some optimizations which improve the performance.

1.1.6 Implementation

We have implemented and tested the algorithms on several student programs as well as standard Java benchmark programs and give performance statistics. As intraprocedural slicing requires inlining of procedures, the intraprocedural slicing algorithm was tested on comparatively small programs without recursion. The algorithm has been implemented for Java source code. The interprocedural slicing algorithm has been implemented for Java bytecode and has been tested on some very large programs.

1.2 Related Work

Much work has been done in the area of static slicing both for sequential programs [86, 12] as well as for concurrent programs. For concurrent programs most work has been done in the area of intraprocedural slicing of concurrent programs. As analysis of concurrent programs tends to have exponential complexity, most solutions trade precision for speed and give imprecise solutions [23, 46]. Krinke [53] was the first to attempt a precise solution to static slicing of concurrent programs that use shared memory. However, his solution is not only impractical in terms of speed but is applicable only to a very restricted model of concurrency.

Slicing intraprocedural programs has a limitation in that it is often impractical to slice realistic programs. Procedures need to be inlined causing the control flow graph to grow potentially without bounds. Few solutions exist for interprocedural slicing of concurrent programs. Millet and Teitelbaum [64] give an algorithm for slicing Promela. Zhao [92] has given a solution for slicing concurrent Java programs, but his solution is both incorrect and imprecise.

1.3 Organization of the Dissertation

Chapter 2 gives some basic background information that makes it easier to read this dissertation. In this chapter we give a brief description of program representations for various models of programs, static slicing, the Static Single Assignment form, DJ graphs and connection graphs.

Chapter 3 gives a solution to intraprocedural slicing of concurrent programs that follow the basic shared-memory concurrency model.

Chapter 4 gives an incremental algorithm to determine whether a given set of nodes belonging to an interprocedural thread form a realizable path in that thread. This algorithm is used to calculate context-sensitive slices.

In Chapter 5, we explain the issues of correctness and precision in computing slices. We give a context-*in*sensitive algorithm which is correct and fast but not very precise. Then we give a context-*sensitive* slicing algorithm which is both correct and precise but not very efficient.

The remaining chapters deal with analysis and static slicing of concurrent Java programs. Chapter 6 gives a combined escape analysis, pointer analysis and data dependence analysis algorithm for sequential Java programs which is efficient in both time and space. Chapter 7 shows how to determine the interference dependences in a Java program with threads, monitors and synchronization. Chapter 8 explores the various issues in extending the slicing algorithm of the basic shared-memory concurrency model to the Java concurrency model. Though the complexity of the context-sensitive algorithm is exponential we show that, with certain optimizations, it is very practical and can be used to slice large Java benchmark programs with great precision. Chapter 9 gives implementation details.

Chapter 10 concludes this dissertation.

Chapter 2

BACKGROUND

In this chapter we give a brief description of some relevant background material used in this dissertation. We cover concurrency models, program representation, static slicing, the Static Single Assignment form, and connection graphs.

2.1 The Control Flow Graph

The basic program representation is a control flow graph. The *Control Flow Graph* (CFG) of a sequential intraprocedural program is a directed graph in which each node represents a statement or predicate in the program. The root of the CFG is the unique *Entry* node and it has no predecessors. There is a unique *Exit* node which has no successors. Every node is reachable from the *Entry* node and the *Exit* node is reachable from every node. This representation is modified in several ways to represent sequential interprocedural programs, concurrent intraprocedural programs and concurrent interprocedural programs.

2.2 Concurrent Programming Models

A concurrent program consists of a number of independently executing threads of control called *tasks*, *processes* or *threads* (in the rest of the dissertation, we use tasks, processes and threads interchangeably). Tasks may execute the same or different streams of instructions. They may operate in isolation or they may communicate, through various mechanisms, with other tasks.

Task communications fall into two major categories: *synchronous* and *asyn-*

chronous. Synchronous communication requires communication partners to execute communication statements at the same time. Two popular synchronous mechanisms are *remote procedure call (RPC)* and *rendezvous*. RPC is a procedure call in that the call site is in one task and the body of the procedure is executed in another task. Rendezvous refers to a pair of tasks that synchronize, potentially pass data, and then continue execution. From the perspective of each task, the communication statement is synchronous as the tasks block until the partner reaches the rendezvous point.

Asynchronous communication allows communication partners to independently execute communication statements as long as sends precede receives. Two popular asynchronous mechanisms are *message passing* and *shared memory*. In message passing, tasks may independently send and receive data. The sender and receiver do not synchronize as a result of the data transfer. Shared memory can be viewed as a form of message passing where sends are writes to shared memory and receives are reads from shared memory.

Techniques for analyzing programs based on synchronous communication [34, 35, 20, 61, 91] are different from those applied to programs based on asynchronous communication [27, 51, 52].

Since the focus of this dissertation is to find solutions to static slicing of concurrent Java, in the rest of the dissertation, we concentrate on the concurrent programming models based on shared memory.

2.2.1 A Basic Shared Memory Concurrency Model

We present here the program model of a concurrent imperative language with shared memory and interleaving semantics. We view concurrent programs semantically as abbreviations of usually much larger nondeterministic programs, which result from a product construction between parallel components [27]. The worst case size of the nondeterministic program grows exponentially in the number of parallel components of the corresponding concurrent program.

The Intraprocedural Model

Parallelism is expressed by the classical `cobegin-coend` statements whose component threads execute in parallel on a shared memory with atomic reads and writes. Threads share the same address space and execute concurrently with each other with

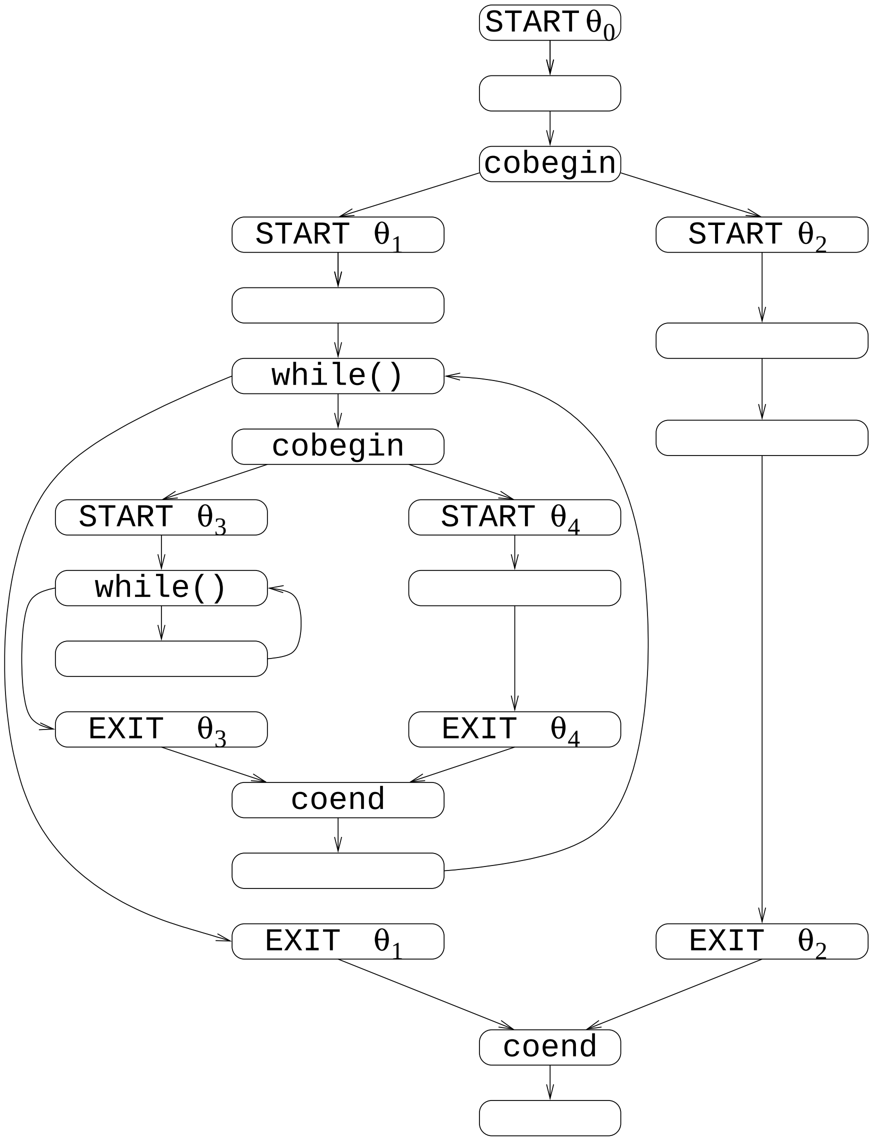


Figure 2.1: The control flow graph for the basic intraprocedural concurrency model

complete interleaving semantics. Threads synchronize at the corresponding `coend` statement. There is no other synchronization between the threads. Threads may be nested, but there may not be jumps leading into a thread or out of a thread. Each thread forms a Single Entry Single Exit region.

The Program Representation The program is represented by a *threaded control flow graph* (TCFG). Each thread is treated like a complete procedure and is represented by a complete control flow graph, where the nodes represent the statements in the program and the edges represent the control flow. The CFG of each thread has a distinguished `START` node and a distinguished `EXIT` node. The `cobegin` statement is represented by a special node with control flow edges from the `cobegin` node to the `START` node of each of the component threads. Similarly, there are control flow edges from the `EXIT` node of each thread to the corresponding `coend` node. Threads may be nested within each other and complete threads may be nested within loop constructs. An example is shown in Figure 2.1.

The Interprocedural Model

The interprocedural model is an extension of the intraprocedural model where each thread may call procedures. There is no limit on the call depth of the procedures and the procedures may even be recursive. However, no threads may be generated from within a recursive procedure. This ensures that no thread may recursively generate itself.

The Program Representation The program representation is an extension of the intraprocedural representation with provision for procedures. The procedures are represented by the standard control flow graph representation. The CFG of each procedure has a distinguished *Entry* node and an *Exit* node. There is a *call* edge from the call site to the *Entry* node of the procedure and a `Return` edge from the *Exit* node of the procedure to the statement that lexically succeeds the call statement. Each thread begins with a distinguished `START` node and terminates at an `EXIT` node. An example is shown in Figure 2.2.

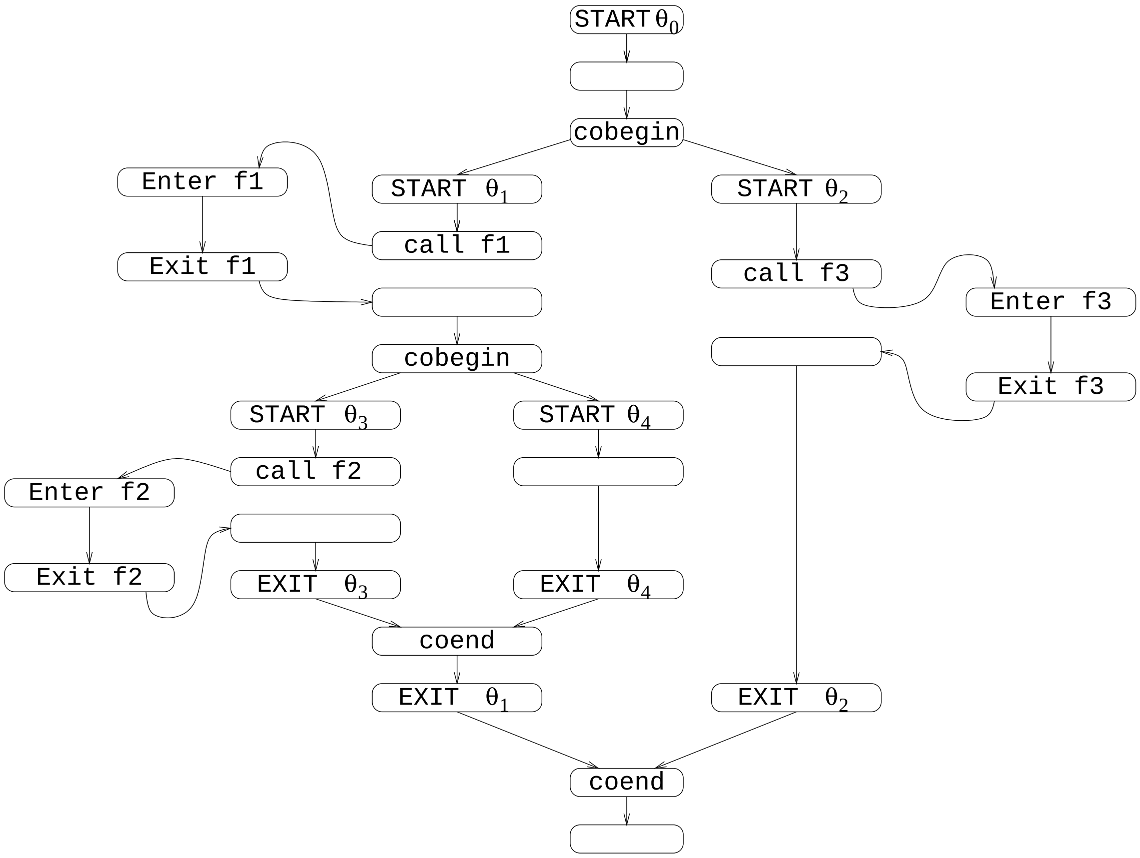


Figure 2.2: The control flow graph for the basic interprocedural concurrency model

2.2.2 The Java Concurrency Model

A key feature of Java is that it is translated into an intermediate language, called Java bytecode. Java bytecode programs are then executed by a Java Virtual Machine (JVM). The execution of Java programs can be described as follows [32]:

- **Phase 1:** The Java program is translated into Java bytecode. A Java program consists of a set of class definitions. For every class defined in the Java program, a *class file* is generated, which contains the bytecode, along with additional information such as signatures and access rights.

- **Phase 2:** A class file is loaded and one of its methods is called. The class file is verified and then the static fields are initialized. At the time of initialization, static code is also executed.

- **Phase 3:** A method is called which is not defined in this class but belongs to another class. If the other class has already been loaded, the access permissions are checked. If the class had not yet been loaded, it is loaded and verified.

Although the Java environment comes with a default programming language, the Java language is not a fundamental component of the underlying JVM technology. Any programming language which can be mapped onto the JVM can be used to develop applications that can execute on and interface with the Java environment [17, 29].

Java's Program Model

The Execution Model Java's execution model is based on a run-time stack, with one stack created per thread. When a method is called, its parameters and local variables are stored on the calling thread's stack. Parameter passing is thus always "call by value", with the value of the parameter copied to the formal parameter at the call point.

The Thread Model Concurrency is modeled with *threads*. Threads are objects of classes that specify thread types. A Java application must contain a `main()` method, which serves as the *main* thread of execution. This is the only thread when the program starts. Although the object containing this method does not have to extend the `Thread` class, it is a separate thread of control. Execution of all other threads is started by calling their `start()` methods. The `start()` method calls the `run()` method, but the `run()` method is never called explicitly.

The Memory Model All objects are explicitly allocated storage from the heap and the objects are manipulated indirectly through implicit references to the heap allocated storage. Automatic garbage collection is performed as a background thread.

The Communication Model Java uses shared memory as the basic model of communication. Communication across threads may be achieved in several ways. A low-level mechanism is to specify the modifier `volatile` on a field that is being accessed by different threads. This inhibits optimizations and ensures that each access to the field is to its current value. A second mechanism is the `synchronized` statement or method. Associated with each object is a conceptual lock that is held by the thread currently executing one of the object's synchronized methods. Analogously, there is a lock for the class itself if it has any static synchronized methods. Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread. It is possible to change the priority of a thread[1].

If a thread θ_1 attempts to call a synchronized method for an object that is locked by another thread, then θ_1 is suspended. When the lock is released, the highest priority thread suspended on the lock is awakened and given the lock as an atomic action. If a synchronized method needs to wait for a condition, then the Java idiom is

```
while ( !condition ) wait ();
```

During the execution of the `wait()` method, the thread releases the lock and becomes inactive. Inactive threads may be awakened only by some other thread executing one of the `notify()` and `notifyAll()` methods of the lock object.

The Program Representation

In Java, threads may be created in any procedure. It is also possible that if a thread is generated from a recursive procedure, that it recursively generates itself. The

[1] In this thesis, we have not researched the effect of priorities on the slcing algorithm

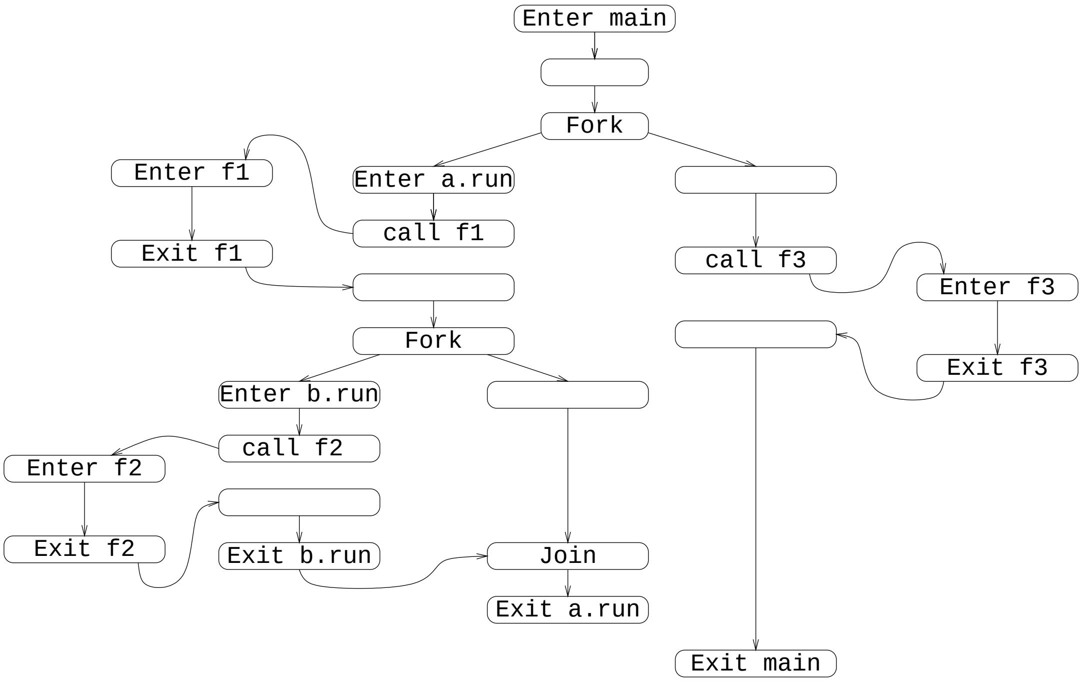


Figure 2.3: The control flow graph for the Java concurrency model

program representation is similar to the basic shared-memory concurrency model. The program begins with the special `main` procedure. Each thread begins with the special `run` method. In the control flow graph, at the call site to the `run` method, we create a special *Fork* node. The *Fork* node has two successors, the *Entry* node of the `run` procedure and the lexical successor of the call site. If a parent thread calls the `join` method on a child thread, we create an edge from the *Exit* node of the corresponding `run` method to the *Join* node. For example, in Figure 2.3, `main` forks a thread that starts with the method `a.run`. This thread in turn forks another thread that starts with `b.run`. The thread started by `a.run` calls `join` on the child thread and hence there is an edge from `Exit b.run` to the *Join* node in `a.run`. But there is no *Join* edge from `a.run` to `main`. This demonstrates the difference between the basic shared-memory concurrency model and the Java concurrency model.

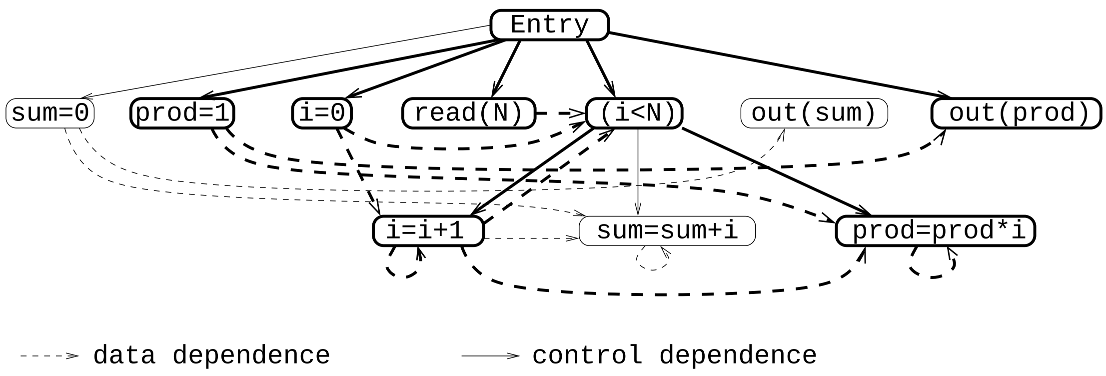


Figure 2.4: The Program Dependence Graph

2.3 Static Slicing

The *slice* of a program with respect to a program point p and a variable x, consists of all statements and predicates of the program that might affect the value of x at point p [88]. Slicing is a program transformation that has successfully been used in debugging [2], testing [44], program maintenance [40], reverse engineering [9], Y2K compliance transformations [65] and applications that involve understanding program behavior.

2.3.1 Slicing Sequential Intraprocedural Programs

Intraprocedural slices can be efficiently computed using a *Program Dependence Graph* (PDG) [70]. The PDG is a graph in which the nodes are the statements and predicate expressions of the CFG of the program and the edges are data dependence and control dependence edges.

Data Dependence A node S_j is data dependent on a node S_i, if

- S_i defines some variable x,
- S_j uses the variable x, and
- there exists a path from S_i to S_j without intervening definitions of x.

begin	**begin**
$sum = 0$	
$prod = 1$	$prod = 1$
$i = 0$	$i = 0$
read (N)	**read** (N)
while $i < N$ **do**	**while** $i < N$ **do**
$i = i + 1$	$i = i + 1$
$sum = sum + i$	
$prod = prod * i$	$prod = prod * i$
od	**od**
output (sum)	
output $(prod)$	**output** $(prod)$
end	**end**

Figure 2.5: The Intraprocedural Slicing Example

Control Dependence A node S_j *postdominates* a node S_i *iff* $S_i \neq S_j$ and S_j is on every path from S_i to *Exit*. A node S_j postdominates a branch of a predicate S_i *iff* S_j is the successor of S_i in that branch or S_j postdominates the successor of S_i in that branch.

A node S_j is control dependent on a predicate S_i *iff* S_j postdominates a branch of S_i but S_j does not postdominate S_i. A node can be directly control dependent on itself. Note that a node with only one successor can never be the source of a control dependence edge.

The Algorithm Given the program dependence graph G and a node s in G, the slice of s is the set of nodes on which s has a transitive data or control dependence. A simple worklist algorithm [49] is given in Figure 2.6. The sets of edges E_{dd} and E_{cd} represent the data dependence edges and control dependence edges, respectively. An example program is shown in Figure 2.5 along with its slice with respect to the statement `output(prod)`. Figure 2.4 shows the corresponding PDG. The nodes and edges in the slice are shown with thicker lines.

```
procedure MarkVertices (G, s)
declare
    G: a program dependence graph
    s: a node in G
    Worklist: a set of nodes in G
    v, w: nodes in G
begin
    Worklist = {s}
    while Worklist ≠ ∅ do
        select and remove node w from Worklist
        Mark w
        for all unmarked nodes v|(v, w) ∈ E_dd ∪ E_cd do
            Worklist = Worklist ∪ v
        endfor
    endwhile
end
```

Figure 2.6: The Intraprocedural Slicing Algorithm

2.3.2 Slicing Sequential Interprocedural Programs

Interprocedural slices are computed using a *System Dependence Graph* (SDG) [49]. The SDG is an an interprocedural graph which is an extension of the program dependence graph. An SDG is a collection of procedure dependence graphs connected by interprocedural control and data dependence edges. A procedure dependence graph is similar to the program dependence graph with additional nodes representing call statements, parameter passing and transitive flow dependencies. A call statement is represented by a *call* node. Parameter passing is represented using four kinds of *parameter* nodes.

- on the calling side, parameter passing is represented by *ActualIn* and *ActualOut* nodes which are control dependent on the call vertex.
- in the called procedure, parameter passing is represented by *FormalIn* and *FormalOut* nodes which are control dependent on the call vertex.

FormalIn nodes are created for every parameter and global variable that may be used or modified by the procedure and *FormalOut* nodes are created for every parameter and global variable that may be modified by the procedure. An *ActualIn* node is created for each *FormalIn* node and an *ActualOut* node is created for each *FormalOut* node. Interprocedural data flow analysis is used to determine the parameter nodes included for each procedure [8, 7].

Transitive dependence edges, called *Summary* edges, are added from *ActualIn* nodes to *ActualOut* nodes to represent transitive flow dependencies due to called procedures. A summary edge is added if a path of control, data and summary edges exists in the called procedure from the corresponding *FormalIn* node to the corresponding *FormalOut* node. Note that the addition of a summary edge in procedure Q may complete a path from a *FormalIn* node to a *FormalOut* node in Q's PDG, which in turn may enable the addition of further summary edges in procedures that call Q. Procedure dependence graphs are connected to form an SDG using three new kinds of edges:

- a *call* edge is added from each call site to the corresponding procedure-entry node.
- a *parameter-in* edge is added from each *ActualIn* node at the call site to the corresponding *FormalIn* in the called procedure.
- a *parameter-out* edge is added from each *FormalOut* node in the called procedure to the corresponding *ActualOut* node at the call site.

The sets of edges E_{dd}, E_{cd}, E_s, E_c, E_{pi} and E_{po} represent the data dependence edges, control dependence edges, summary edges, call edges, parameter-in edges, and parameter-out edges respectively. Figure 2.7 shows an example program and its SDG.

The Algorithm The interprocedural slice with respect to a node s is computed using two passes over the SDG.

The traversal in pass 1 starts from s and goes backwards (from target to source) along data dependence, control dependence, summary, call, and parameter-in edges, but *not* along parameter-out edges. Informally, if s is in procedure P then pass 1 identifies nodes that reach s and are either in P itself or in procedures that (directly or transitively) call P. ParameterOut edges are not followed but the source of the parameter-out edges are collected in a separate worklist. Hence the traversal in pass 1 does not descend into procedures called by P. Summary edges from *ActualIn* to *ActualOut* nodes permit the discovery of nodes that can reach s only through a

```
procedure Main
  sum=0
  i=1
  while i < 11 do
    call A(sum, i)
  od
  print ( sum )
end

procedure A(x, y)
  call Add(x, y)
  call Inc(y)
return

procedure Add(b, c)
  a=a+b
return

procedure Inc(z)
  call Add(z, 1)
return
```

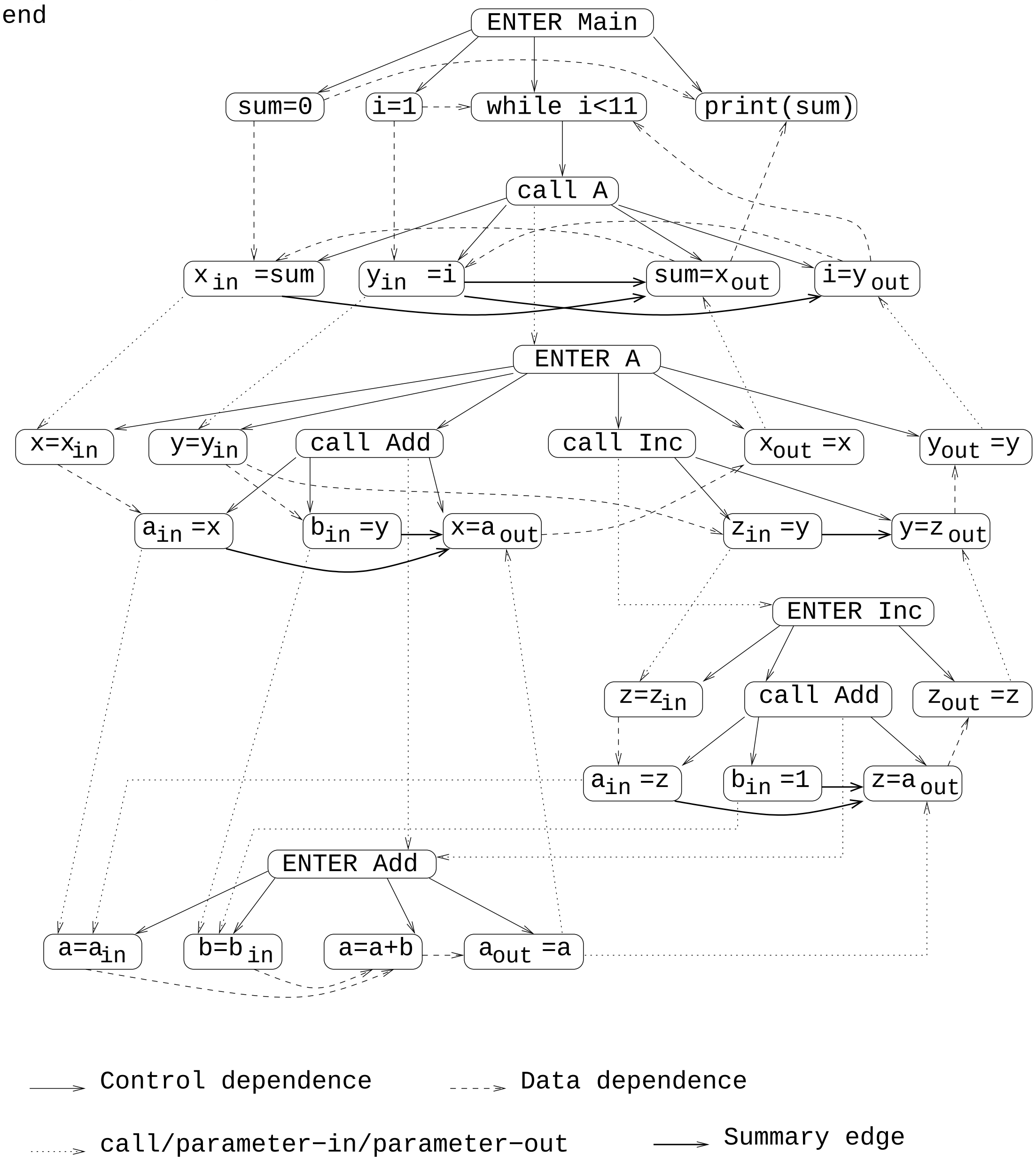


Figure 2.7: The System Dependence Graph

```
procedure MarkVertices (G, s)
declare
    G: a system dependence graph
    s: The slicing criterion, a node in G
    S: a set of nodes in G
    Worklist: a set of nodes in G
    v, w: nodes in G
begin
    S = ∅
    Worklist = {s}

    /* Phase 1: Slice without descending into called procedures */
    while Worklist ≠ ∅ do
        select and remove node w from Worklist
        Mark w
        for all unmarked nodes v|(v, w) ∈ E_dd ∪ E_cd ∪ E_s ∪ E_c ∪ E_pi do
            Worklist = Worklist ∪ v
        endfor
        for all unmarked nodes v|(v, w) ∈ E_po do
            S = S ∪ v
        endfor
    endwhile

    /* Phase 2: Slice called procedures without ascending to call sites */
    Worklist = S
    while Worklist ≠ ∅ do
        select and remove node w from Worklist
        Mark w
        for all unmarked nodes v|(v, w) ∈ E_dd ∪ E_cd ∪ E_s ∪ E_po do
            Worklist = Worklist ∪ v
        endfor
    endwhile
end
```

Figure 2.8: The 2-Phase Interprocedural Slicing Algorithm

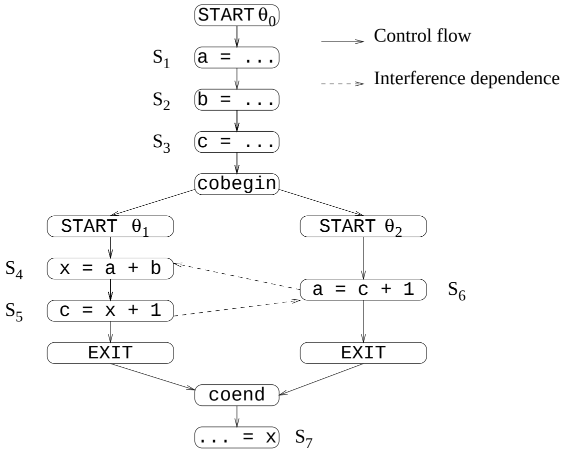


Figure 2.9: A threaded control flow graph

called procedure. Summary edges permit moving across a call site without having to descend into the called procedure. Thus there is no need to keep track of the calling context explicitly to ensure that only legal execution paths are traversed.

Pass 2 identifies nodes in called procedures that induce the summary edges used to move across call sites in pass 1. These are the nodes that can reach s from procedures called by P or from procedures called by procedures that (transitively) call P. The traversal in pass 2 starts from all nodes reached in pass 1 by parameter-out edges and goes backwards along data dependence, control dependence, summary and parameter-out edges, but *not* along call or parameter-in edges. The traversal in this pass does not "ascend" into calling procedures. The summary edges from $ActualIn$ nodes to $ActualOut$ nodes make such ascents unnecessary.

The result of the interprocedural slice consists of the set of nodes encountered during pass 1 and pass 2 and the set of edges induced by this set of nodes. A worklist algorithm is given in Figure 2.8.

2.3.3 Slicing Concurrent Intraprocedural Programs

Consider the basic shared-memory model of concurrent programs. *Threaded PDG* (TPDG) is an extension of PDGs to concurrent programs [53]. TPDGs capture, besides data and control dependencies, additional dependencies arising out of *interference* between concurrent execution of threads. Interference dependence is defined as : *A node S_1 is interference dependent on a node S_2 if S_2 defines a variable v, S_1*

uses the variable v, and S_1 and S_2 execute in parallel.

As stated by Krinke [53], interference dependence is fundamentally different from data or control dependence as it is intransitive. In Figure 2.9, S_4 is interference dependent on S_6 and S_6 is interference dependent on S_5. Yet S_4 cannot be dependent on S_5 in any execution of the program. Hence a slicing algorithm that computes a simple transitive closure on the TPDG could generate an imprecise slice.

Krinke was the first to propose a solution to this problem. He showed that the reachability algorithm can be extended to compute precise slices as follows[53]: The slicing algorithm maintains a tuple of nodes which keeps track of the last node visited in each thread. The slicing algorithm starts at the slicing criterion s and traverses backwards along data, control and interference dependence edges. When following backwards an interference dependence edge $i \xrightarrow{id} j$, from a node j in a thread θ_j to a node i in a thread θ_i, it crosses from the thread, θ_j to the thread θ_i. In this case it remembers that the last node visited by the algorithm in θ_j was j. When the algorithm re-enters the thread θ_j along some other interference dependence edge to some node k in θ_j, it checks if there is a path from k to j. For example, in Figure 2.9 to find the slice of S_4 in θ_1, the algorithm follows the interference dependence edge to S_6 in θ_2. S_6 is interference dependent on S_5 in θ_1 but there is no path from S_5 to S_4 and so the algorithm correctly rejects S_5 from the slice.

His algorithm, however, gives imprecise slices in the presence of nested threads and when complete threads are nested within loops.

2.3.4 Slicing Concurrent Interprocedural Programs

The *threaded SDG (TSDG)* is a natural extension of the SDG for sequential programs with edges added for interference dependence and synchronization. The complete set of edges includes the data, control, interference, synchronization, call, summary, parameter-in and parameter-out dependence edges which are denoted by E_{dd}, E_{cd}, E_{id}, E_{syn}, E_c, E_s, E_{pi} and E_{po} respectively. Zhao [93] applies a simple two-phase algorithm to the TSDG which, as we will show in this dissertation, gives incorrect, imprecise and context insensitive slices.

2.4 The Static Single Assignment Form

First we introduce some standard terminology.

Dominators

- In a control flow graph, a node S_x *dominates* a node S_y if all paths from $Entry$ to S_y must pass through S_x. We write S_x **dom** S_y if S_x dominates S_y and S_x !**dom** S_y if S_x does not dominate S_y.

- S_x *strictly dominates* S_y if S_x dominates S_y and $S_x \neq S_y$. We write S_x **sdom** S_y if S_x strictly dominates S_y and S_x !**sdom** S_y if S_x does not strictly dominate S_y.

- S_x is the *immediate dominator* of S_y if S_x strictly dominates S_y and all other nodes which strictly dominate S_y also dominate S_x. We write $S_x =$ **idom** (S_y) if S_x is the immediate dominator of S_y.

The dominance relation is reflexive and transitive and can be represented by a tree called the **dominator tree** [3, 59]. $S_x \rightarrow S_y$ is an edge in the tree if and only if $S_x =$ **idom** (S_y).

Dominance Frontier

- S_z is in the *dominance frontier* of S_y if S_y does not strictly dominate S_z, but S_y does dominate some predecessor of S_z.

That is, if there is a path from S_y to S_z, but there is also a path from $Entry$ to S_z which does not pass through S_y, then S_z is in the dominance frontier of S_y.

The Iterated Dominance Frontier

Let $DF(x)$ be the dominance frontier of the node x and let $DFS(S)$ be the dominance frontier of a set of nodes S. Then

- $DFS(S) = \cup_{x \in S} DF(x)$

The *iterated dominance frontier* $IDF(S)$ for a set of nodes S is the limit of the increasing sequence:

- $IDF_1(S) = DFS(S)$

- $IDF_{i+1}(S) = DFS(S \cup IDF_i(S))$

```
x = 0
y = 0
z = 0
if ( P ) then
    y = y + 1
endif

x = y
z = 2 * y - 1
```

(a)

$$
\begin{array}{l}
\mathtt{x_0 = 0} \\
\mathtt{y_0 = 0} \\
\mathtt{z_0 = 0} \\
\mathtt{if\ (\ P\ )\ then} \\
\quad \mathtt{y_1 = y_0 + 1} \\
\mathtt{endif} \\
\mathtt{y_2 = \phi\ (\ y_0\ ,\ y_1\ )} \\
\mathtt{x = y} \\
\mathtt{z = 2\ *\ y\ -\ 1}
\end{array}
$$

(b)

```
x = 0

y = 0

z = 0

if ( P ) then

    y = y + 1

endif

y = φ ( y , y )

x = y

z = 2 * y - 1
```

(c)

Figure 2.10: Static Single Assignment Form and Def-use chains

The Static Single Assignment Form

The static single assignment form has become a powerful framework with which to analyze and optimize code. Figure 2.10 shows a program fragment translated into SSA form. After a program has been converted into the SSA form [31], it has two distinguishing properties:

1. Every use of a variable in the program has exactly one reaching definition, and
2. At confluence points in the CFG, merge functions called *ϕ-functions* are introduced. A ϕ-function for a variable merges the values of the variables from distinct incoming control flow paths (in which a definition occurs along at least one of these paths), and has one argument for each control flow predecessor.

Two key algorithms which convert a program into SSA form are the *placement* of ϕ-functions to satisfy the second property, and *renaming* of variables to satisfy the first property. For every variable, a ϕ-function is placed at each point in the program where distinct definitions merge. The ϕ-function itself is regarded as an additional definition, and the algorithm continues until a fixed point is reached. The placement of ϕ-functions is performed typically by computing the iterated dominance frontier of the set of nodes where the variable is defined [31, 83] although there are other methods to compute the location of the ϕ-functions [50, 16]. Next, each variable definition is uniquely renamed, and each use is replaced by its one reaching definition. All ϕ-functions receive a unique name since they are considered definitions, while their arguments are renamed to the definition that reaches the corresponding control flow predecessor.

Although the traditional SSA form renames variables uniquely at every definition point, it is not really practical to add new names to the symbol table for all assignments. Thus the common implementations actually provide def-use links for each new definition (see Figure 2.10(c)). Since each use is the head of exactly one link, the semantics of SSA are preserved.

The advantage of the SSA form is that the number of ϕ-functions is always linear [31] even though the dominance frontier may be quadratic in the number of nodes. In addition, the SSA form ensures that the number of def-use links is linear in the size of the graph, whereas in the standard representation [3], the def-use links can easily be quadratic in the size of the graph.

2.5 DJ Graphs

On the dominator tree we define $SubTree(S_x)$ to be the dominator sub-tree rooted at S_x. That is, $SubTree(S_x)$ is the set of nodes dominated by S_x. With each node in the dominator tree we associate a level. The root of the tree is at level 0 ($START.level = 0$) and the level number increases as the depth of the tree. The dominator tree edges are termed D-edges.

A DJ graph is a dominator tree augmented with J-edges.

- An edge $S_x \rightarrow S_y$ in a CFG is named a J-edge if S_x !**sdom** S_y. Further S_y is named a `join` node.

It is simple to locate the J-edges. In the CFG every edge $S_x \rightarrow S_y$ is a J-edge if $S_x \neq \mathbf{idom}(S_y)$. A J-edge $S_x \rightarrow S_y$ is termed a *back* J-edge if S_y **dom** S_x, else it is called a *cross* J-edge.

The DJ graph is linear in the size of the CFG, can be constructed in linear time and has a number of interesting properties [83]. Given the DJ graph the algorithm for computing the dominance frontier of a node is

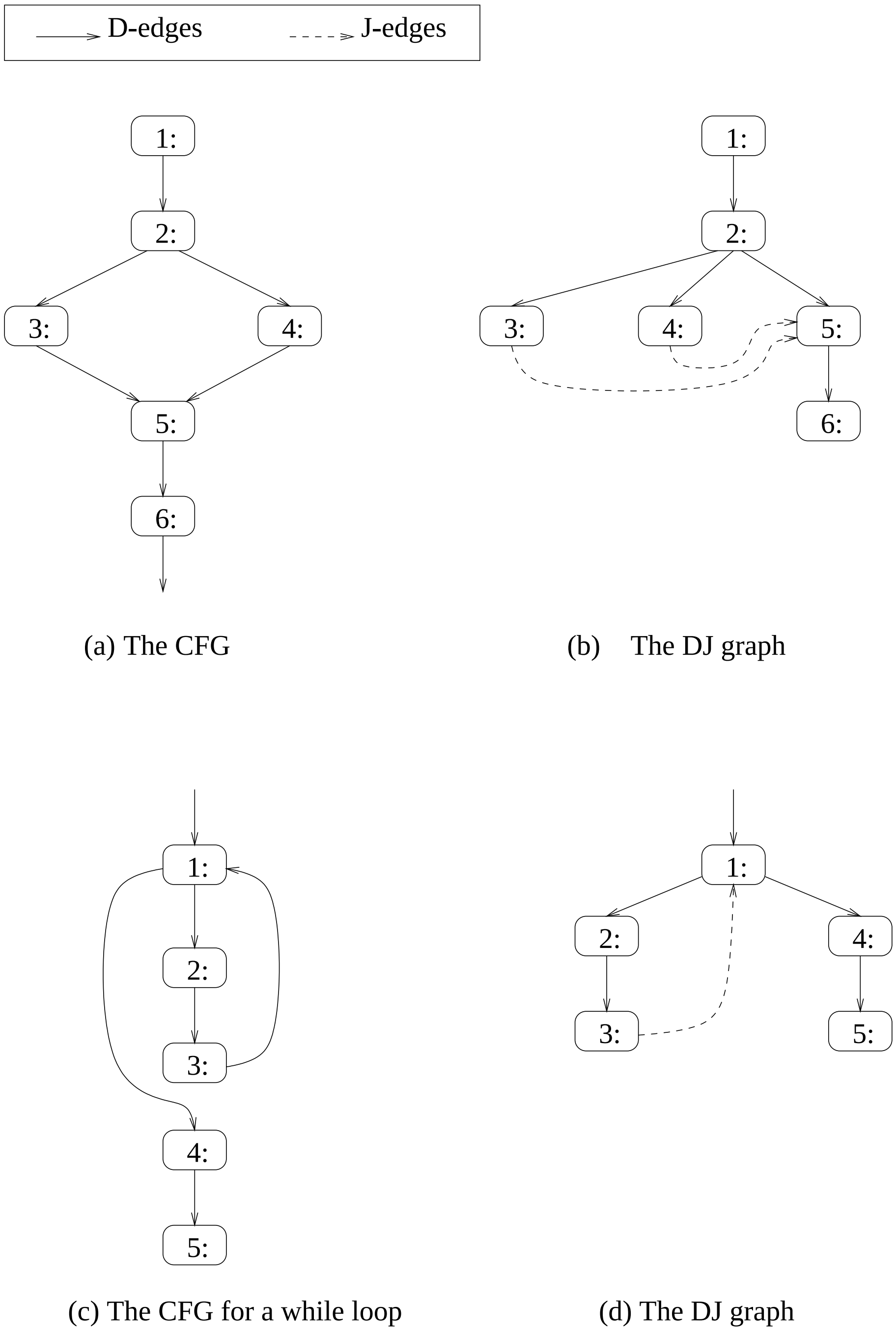


Figure 2.11: The DJ graph

DomFrontier(x)

$DF_x = \emptyset$

for each $y \in SubTree(x)$ **do**

if ($(y \rightarrow z == Jedge)$ **and** $(z.level \leq x.level)$)

$DF_x = DF_x \cup z$

The DJ graph for a branch and a loop are shown in Figure 2.11. In Figure 2.11(b), S_3 does not strictly dominate S_5, but it does dominate S_3 which is a predecessor of S_5. (There is a path from S_3 to S_5, but there is a path to S_5 which does not go through S_3). Hence S_5 is in the dominance frontier of S_3. Similarly, the dominance frontier of the node S_4 consists of the node S_5. In Figure 2.11(d), the dominance frontier of the nodes S_2 and S_3 consists of the node S_1. In Figure 2.11(b), the J-edges are *cross* J-edges and in Figure 2.11(d), the J-edge is a *back* J-edge.

2.6 Connection Graphs

The Connection Graph [25] is an abstract graph representation that models pointer analysis and is used for escape analysis. A connection graph is a directed graph $CG = (N_o \cup N_r \cup N_f \cup N_g, E_r \cup E_d \cup E_f)$, where

- N_o represents the set of objects. One object is created for each `new` statement.
- N_r represents the set of reference variables in the program.
- N_f represents the set of non-static field nodes.
- N_g represents the set of static field nodes.
- E_r is the set of points-to edges. If $x \xrightarrow{P} y \in E_r$, then $x \in N_r \cup N_f \cup N_g$ and $y \in N_o$.
- E_d is the set of deferred edges. If $x \xrightarrow{D} y \in E_d$, then $x, y \in N_r \cup N_f \cup N_g$.
- E_f is the set of field edges. If $x \xrightarrow{F} y \in E_f$, then $x \in N_o$ and $y \in N_f \cup N_g$.

Deferred edges are used to model assignments that merely copy references from one variable to another. Given a reference node $m \in N_r \cup N_f \cup N_g$, the set of object nodes $O \subseteq N_o$ that it (immediately) points-to can be determined by traversing the

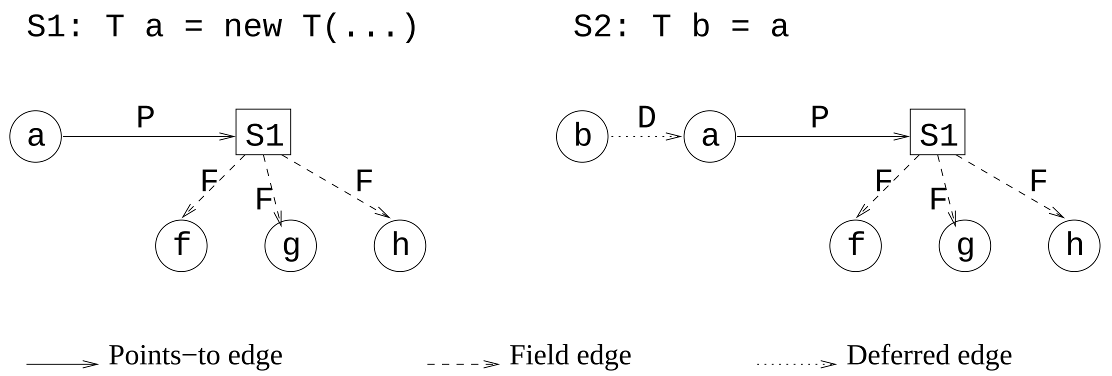


Figure 2.12: A Simple Connection Graph

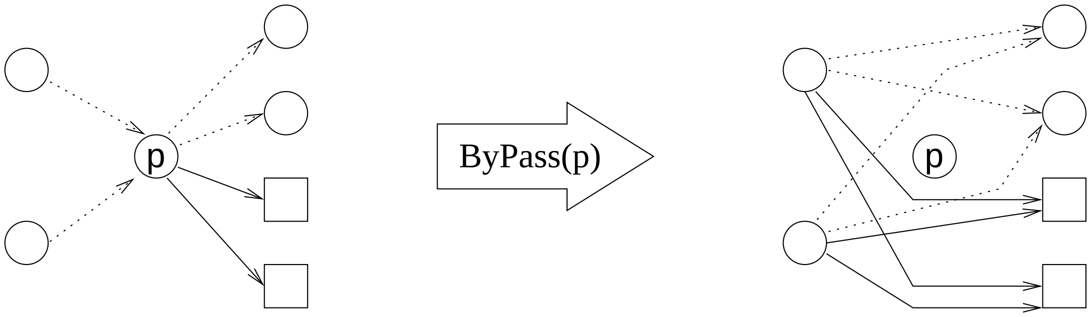


Figure 2.13: Illustrating the $ByPass(p)$ function

deferred edges from m upto the first points-to edge in the path. The destination node of the points-to edge will be in O. A simple connection graph is shown in Figure 2.12.

Intraprocedural Analysis There are two types of updates on points-to information: weak updates leave existing points-to edges in place when adding new edges, and strong updates remove (kill) existing edges when inserting new edges.

The $ByPass(p)$ function applied to a node $p \in N_r \cup N_r$ redirects the incoming deferred edges of p to the successor nodes of p and removes the outgoing edges from p as shown in Figure 2.13. The $ByPass(p)$ function performs a strong update.

A merge between two connection graphs $C_1 = (N_1, E_1)$ and $C_2 = (N_2, E_2)$ is the union of the two graphs: $C_1 \wedge C_2 = (N_1 \cup N_2, E_1 \cup E_2)$.

Intraprocedural analysis iteratively applies a transfer function to four basic types of statements that affect intraprocedural escape analysis.

p = new T() Create a new object node O, apply $ByPass(p)$ and add a new points-to edge from p to O.

p = q Apply $ByPass(p)$, then add a deferred edge from p to q.

p.f = q Let $U = PointsTo(p)$. If $U = \emptyset$, create a phantom node O_{ph} and insert a points-to edge from p to O_{ph}. During interprocedural analysis, the phantom nodes are mapped to actual nodes created by the calling procedure. Let $V = \{v | u \xrightarrow{F} v$ and $u \in U$ and v represents the field f. Add edges $\{v \xrightarrow{D} q | v \in V\}$ to the connection graph. The analysis cannot apply $ByPass(p.f)$ and hence can only perform weak updates on field variables.

p = q.f Let $U = \{u | q \xrightarrow{+P} u\}$, $V = \{v | u \xrightarrow{F} v\}$ and $u \in U$ and v represents the field f. If U is empty, create a phantom node and add it to U. Apply $ByPass(p)$ and add the edges in $\{p \xrightarrow{D} v | v \in V\}$ to the connection graph.

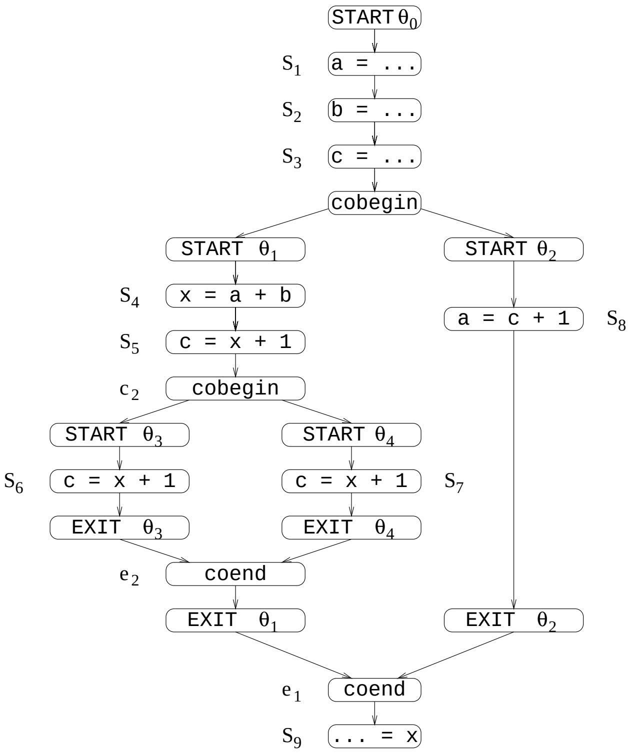
Chapter 3

SLICING CONCURRENT INTRAPROCEDURAL PROGRAMS

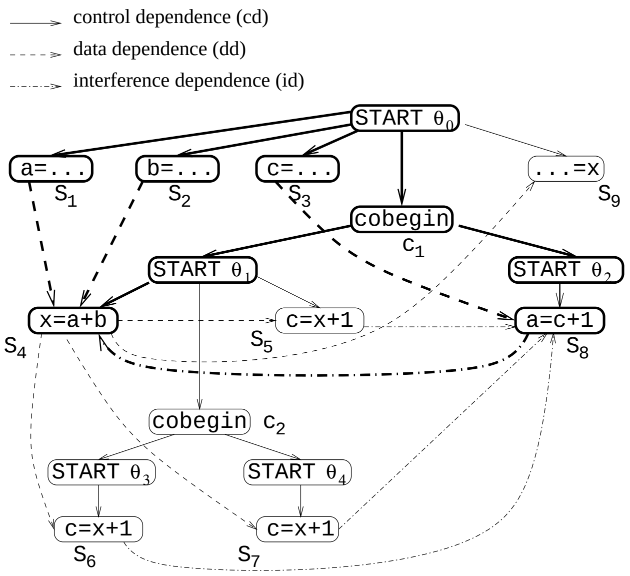
In this chapter we describe an intraprocedural slicing algorithm for the basic shared-memory concurrency model with `cobegin-coend` parallelism and no explicit synchronization. We first discuss the work done by previous authors in this area. We show why previous methods do not give precise solutions in the presence of nested threads and loops and describe our solution that correctly and efficiently computes slices [68]. Though the complexity of this algorithm is exponential on the number of threads, a number of optimizations are suggested. Using these optimizations, we are able to get near linear behavior for many practical concurrent programs.

3.1 Introduction

For sequential programs, the definition of a slice is given in terms of the standard program dependence graph [70] representation of programs where the nodes, representing assignments and conditions in the program, are connected by control and data dependence edges. The PDG of a program can be obtained from its CFG by removing control flow edges and adding the data and control dependence edges. The slice can be defined with respect to a given node in the PDG as the set of all the nodes on which the given node is directly or transitively dependent. Thus given the PDG, the slice can be computed by a simple reachability algorithm [70].



(a) A threaded control flow graph (TCFG)



(b) The corresponding threaded program dependence graph (TPDG)

Figure 3.1: Representation of threaded programs

In concurrent programs, data dependence and control dependence is calculated using standard techniques of data flow analysis for concurrent programs with shared memory and interleaving semantics [51]. In addition, the `START` node of each thread is control dependent on the corresponding `cobegin` node and the `coend` node is control dependent on the corresponding `cobegin` node. Threaded PDG (TPDG) is an extension of PDGs to concurrent programs [53]. TPDGs capture, besides data and control dependencies between nodes in a single thread, additional dependencies arising out of *interference* between concurrent execution of threads. Interference dependence is given by

Definition 3 *A node S_1 is interference dependent on a node S_2 if S_2 defines a variable v, S_1 uses the variable v and S_1 and S_2 execute in parallel.*

Interference dependence is essentially a data dependence from one node to another, where the two nodes belong to parallel executing threads. These dependencies are modeled by adding an interference dependence edge from the node, S_2, *defining* a variable to another node S_1, *using* the same variable where S_1 and S_2 may execute in parallel. The concurrent program in Figure 3.1(a) has five threads. The main thread θ_0 contains two top level threads θ_1 and θ_2 starting at S_4 and S_8 respectively. θ_1, in turn, generates threads θ_3 and θ_4. The corresponding TPDG is shown in Figure 3.1(b). Unlike the TPDG used by Krinke, we do not keep the control flow edges in our TPDG.

Previous Work The slicing criterion for a concurrent program is defined by the triple $\langle t, p, V \rangle$, where t is a thread in the program, p is a program point in t and V is a subset of the program's variables. Given a TPDG, the slicing criterion is clearly identified as a node in the TPDG. Cheng [23] and Zhao [93] have proposed slicing algorithms for concurrent programs which start at the slicing criterion and traverse backwards along data dependence, interference dependence and control dependence edges. Their algorithm is a direct extension of the slicing algorithm for sequential programs. As observed by Krinke [53], this leads to inaccurate slices. For example, in Figure 3.1(b), S_4 is interference dependent on S_8 and S_8 is interference dependent on S_5. Yet it is not possible for S_4 to be dependent on S_5 in any execution of the program. The reason for the inaccuracy is because interference dependence is fundamentally different from data or control dependence. The composition of control or data dependence is always transitive: a dependence from x to y and

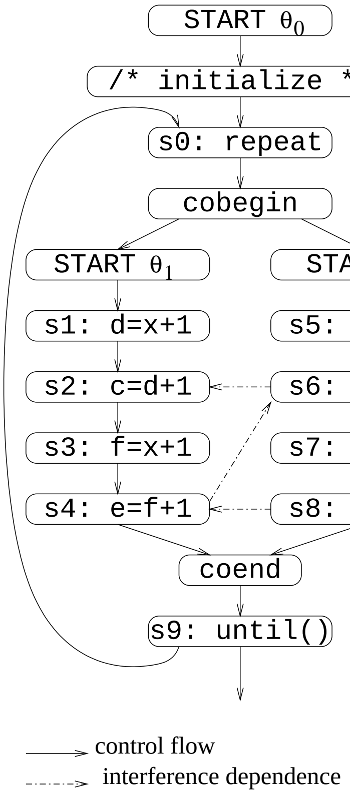


Figure 3.2: A thread nested within a loop

a dependence from y to z, implies a (transitive) dependence from x to z. But interference dependence is intransitive.

In order to solve this problem Krinke, tries to capture the concept of a proper execution path. He introduced the notion of a *threaded witness* which he defines as follows: a *threaded witness* in a TCFG, G, is an ordered sequence of nodes $\langle n_1, n_2, \ldots, n_k\rangle$ in G such that for any two nodes n_i and $n_j, i < j$ in the sequence, if n_i and n_j belong to the same thread then there is a path in G from n_i to n_j. However, this definition is not strong enough to capture all possible program constructs. In Figure 3.1, for example, there is a program with nested threads. According to Krinke's definition $\langle S_5, S_8, S_4\rangle$ is *not* a valid execution sequence (or threaded witness). However, according to his definition, $\langle S_6, S_5, S_3\rangle$ *is* a valid threaded witness. As a result his slicing algorithm would (correctly) exclude S_5 but would include both S_6 and S_7 in the slice of S_4 (which is imprecise). A more precise slice is $\{S_1, S_2, S_3, S_4, S_8\}$, shown in thicker lines in Figure 3.1(b), which is computed by the algorithm proposed in this chapter.

Further, consider the program in Figure 3.2. S_2 is interference dependent on S_6 which is interference dependent on S_4. But it is not possible for S_2 to be transitively dependent on S_4 as the definition at S_4 is killed by the definition at S_5 and the definition at S_6 is killed by the definition at S_1. However, by Krinke's definition $\langle S_4, S_6, S_2 \rangle$ is a valid threaded witness as there is a path from S_4 to S_2 induced by the loop and hence his algorithm would (*imprecisely*) include both S_4 and S_8 in the slice of S_2.

This chapter is an attempt to define and compute more accurate slices than the earlier works. Its contributions are

1. A precise definition of slicing that is applicable to concurrent programs with arbitrary nesting of parallelism and loops.
2. A slicing algorithm and its proof of correctness.
3. A number of optimizations on the basic slicing algorithm and case studies to illustrate the effectiveness of these optimizations.

The organization of this chapter is as follows: In Section 3.2 we describe an extended threaded program dependence graph. In Section 3.3 we give a formal definition of the slice of a concurrent program based on the extended threaded program dependence graph representation. Then we describe the slicing algorithm and give examples on nested threads and loops. In Section 3.4 we discuss the complexity and describe several optimizations with the help of which our slicer runs in near linear time on many standard concurrent programs. Some experimental results are included in the same section. Section 3.7 concludes the chapter. In the chapter appendix we give the algorithm for constructing the extended threaded program dependence graph and a proof for the slicing algorithm.

3.2 The Threaded Program Dependence Graph

We use an extended threaded program dependence graph (TPDG) which is very similar to the TPDG described by Krinke [53]. The basic difference is that Krinke does not differentiate between loop-carried data dependence edges and loop-independent data dependence edges or interference dependence edges, but we do.

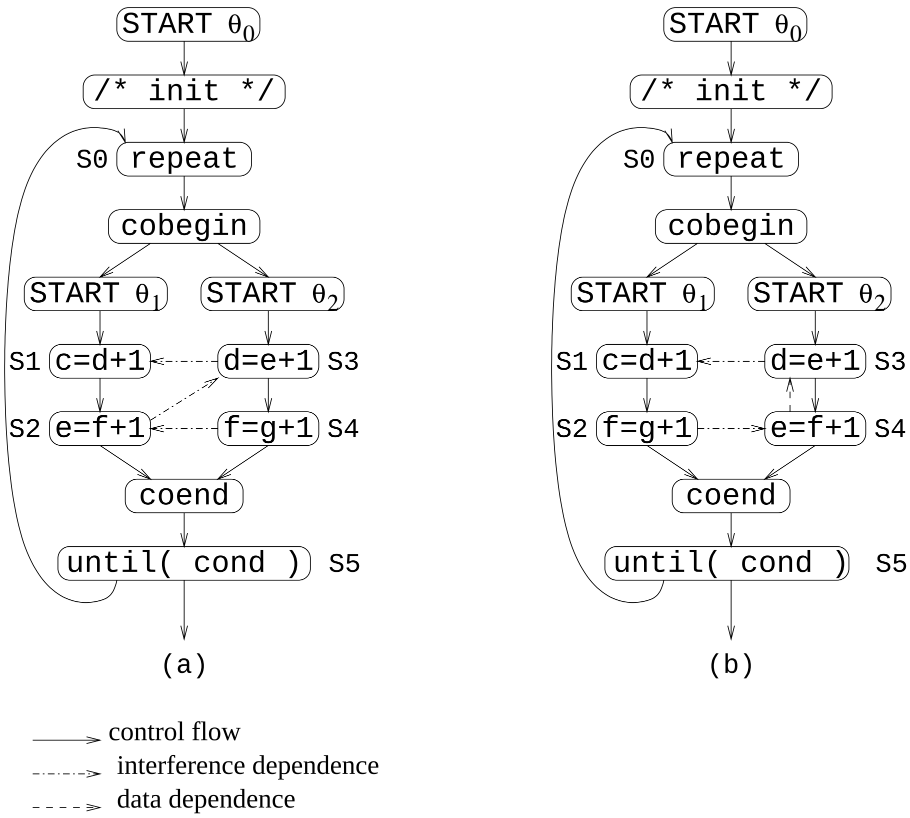


Figure 3.3: Loops

Consider the program in Figure 3.3(a). It is clear that there is an interference dependence from S_3 to S_1. However, if we unfold the loop (Figure 3.4), we see that there is *also* a loop-carried data dependence from S_3 to S_1. Similarly, in Figure 3.3(b), there is a loop-carried data dependence from S_4 to S_3. In concurrent programs we need to differentiate between loop-carried data dependence and loop-independent data dependence or interference dependence for two reasons

1. Although *lexically* two nodes S_i in thread θ_i and S_j in thread θ_j may execute in parallel, a loop-carried dependence from S_i to S_j refers to sequentially executing instances of the two nodes. We need two edges between the same two nodes, one representing interference dependence and the other representing loop-carried data dependence.

2. Consider two threads θ_i and θ_j that do not execute in parallel and are embedded in the same loop, l. Let S_i be a node in θ_i and let S_j be a node in θ_j.

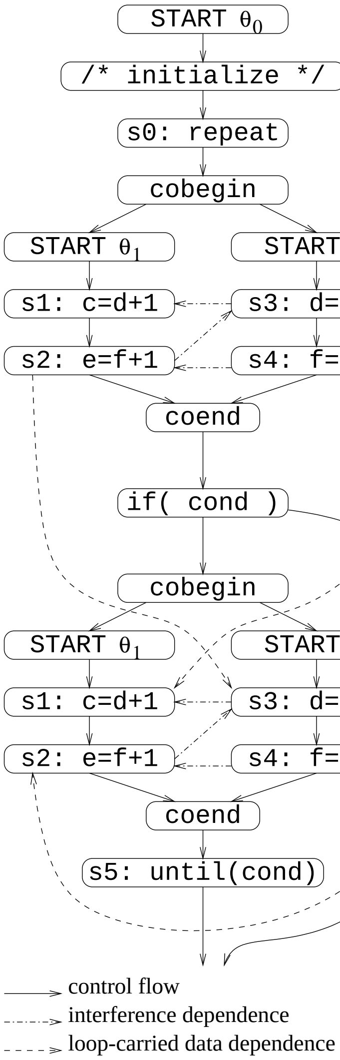


Figure 3.4: Loop-carried data dependences

There are two possibilities

(a) θ_i and θ_j are disjoint. Lexically, let θ_j execute before θ_i. However, a loop-carried dependence from S_i to S_j refers to an instance of θ_i that completes before an instance of θ_j.

(b) Let θ_i be nested within θ_j. Lexically, both threads execute simultaneously. However, here also, a loop-carried dependence from S_i to S_j refers to an instance of θ_i that executes *and terminates* before the next instance of θ_j.

We need to differentiate between ordinary data dependence and loop-carried data dependence to simulate the *exit* of the defining thread.

It is easy to determine the loop-carried dependences. In our algorithm we assume that we are dealing with structured programs with properly nested loops and threads. The algorithm is given in Section 3.8.1. Our TPDG is a multigraph where for every pair of nodes S_i and S_j such that S_i is interference dependent on S_j, there are two edges from S_j to S_i if S_i is *also* loop-carried data dependent on S_j. One edge is marked as an interference dependence edge and the other is marked as a loop-carried data dependence edge. For example, in Figure 3.3(a) the corresponding TPDG would have an interference dependence edge from S_3 to S_1 and a loop-carried data dependence edge marked with the loop whose header is S_0. In addition, all data dependence edges that are actually loop-carried data dependence edges are so marked if the loop has a thread nested within it. For example, in Figure 3.3(b), the corresponding TPDG would have no data dependence edge from S_4 to S_3, but it would have a loop-carried data dependence edge from S_4 to S_3 annotated with the loop whose header is S_0. Loop-carried data dependence edges that do not cross thread boundaries are marked as ordinary data dependence edges. All loop-carried data dependence edges are annotated with the name of the loop that induced the dependence. We consider only the outermost loop that preserves a particular loop-carried dependence as this preserves all the paths in the inner loop too.

3.3 Slicing Concurrent Programs

First we introduce a few notations and functions.

Let G be the TCFG of an arbitrary program. Let θ_0, θ_1, $\cdots$ θ_n denote the (names of) threads in G with θ_0 being the main thread. Given a node x in G, let $\theta(x)$ return the innermost thread enclosing x. Given two threads θ_i, θ_j, define

$$||(\theta_i, \theta_j) = \begin{cases} \textbf{true} & \text{if } \theta_i \text{ and } \theta_j \text{ execute in parallel} \\ \textbf{false} & \text{otherwise} \end{cases}$$

A *region* R of G is a subgraph whose nodes are in G and such that an edge (m, n) of G is in R if and only if m and n are both in R. A node m is an *entry node* of R if there is an edge (v, m) of G such that v is not in R. A node n is an *exit node* of R if there is an edge (n, v) of G such that v is not in R. A *Single Entry Single Exit (SESE) region* is a region with a unique entry node and a unique exit node.

Let i and j be two nodes that execute in parallel, that is, $||(i, j)$ is true. Then, for the program model under consideration, the nearest common ancestor of i and j must be a `cobegin` node. This is true even in the case of nested threads. We define $CoAncestor(i, j)$ to be the nearest common ancestor of two parallel executing nodes i and j and $CoRegion(i, j)$ to be the SESE region whose entry node is $CoAncestor($ i, $j)$ and whose exit node is the corresponding `coend` node.

Let i, j be nodes and R be a region in the TCFG. Then

$$Path(i, j, R) = \begin{cases} \textbf{true} & \text{if there is a path from } i \text{ to } j \text{ in } R \\ \textbf{false} & \text{otherwise} \end{cases}$$

As already explained, a simple reachability algorithm generates imprecise slices in concurrent programs. The basic problem with concurrent programs is that the subgraph obtained by including all reachable nodes from the slicing criterion does not satisfy the constraints of execution of parallel programs. So an additional constraint modeling the execution of parallel programs has to be placed. A standard constraint used in the literature is the notion of execution traces.

Definition 4 *A trace in the TCFG, G, of a concurrent program, is an ordered set of nodes* $\langle n_1, \cdots, n_k \rangle$ *that forms a valid execution path in G.*

A trace in G can be defined recursively as follows:

- Let R be a SESE region in G without any nested `cobegin-coend` node; let n_s be the entry node and let n_e be the exit node of R. A trace in R is defined as any path in R that starts at n_s and terminates at n_e.

- Let R be a SESE region in G with one or more nested `cobegin-coend` nodes. Let R'_i be the SESE regions between a `cobegin` node and its corresponding `coend` node (the entry node is the `cobegin` node and the exit node is the `coend` node). A trace in R is defined as any trace in R generated by treating each R'_i as a single node n'_i and then replacing n'_i by a trace in R'_i.

- Let R' be a SESE region between a `cobegin` node and its corresponding `coend` node. R' consists of a set of SESE regions, with one region for each thread generated by the `cobegin` statement (the entry node of each region is the `BEGIN` θ_i node of the thread and the exit node is the `EXIT` θ_i node). A trace in R' is any interleaving of traces in each of the SESE regions in R'.

Definition 5 *The trace set of G, denoted by $Tr(G)$ is the set of all traces in G.*

Given the set of traces, $Tr(G)$ we define

Definition 6 *A* **trace witness** *in G is an ordered sequence of nodes $\langle n_1, n_2, \ldots, n_k \rangle$ in G such that it forms a subsequence of some trace in the set $Tr(G)$.*

Clearly, for a region R' between a `cobegin` node and its corresponding `coend` node, a trace witness is defined as the set of sequences of ordered nodes $\langle n_1, n_2, \ldots, n_k \rangle$ in R' such that for any two nodes n_i and $n_j, i < j$ in the sequence, either n_i and n_j execute in parallel **or** there is a path from n_i to n_j in R'. That is, $||(\theta(n_i), \theta(n_j))$ or $Path(n_i, n_j, R')$.

3.3.1 The Slicing Algorithm

Given the TCFG, G of a concurrent program, the corresponding TPDG has the following set of edges: E_{dd}, E_{cd} and E_{id} represent the set of data dependence, control dependence and interference dependence edges respectively and E^l_{dd} represents the set of loop-carried data dependence edges induced by the loop l.

Given the TPDG, the slicing criterion is any node in the TPDG.

Definition 7 *The* **slice** *of a TPDG w.r.t. the criterion p, denoted by, $\mathcal{S}(p)$, consists of all nodes q on which p transitively depends and there exists a trace witness in G for each such q. More formally,*

$$\begin{aligned}\mathcal{S}(p) = \{q \quad | \quad & \mathcal{P} = \langle n_1, \ldots, n_k \rangle, \\ & q = n_1 \xrightarrow{d_1} \ldots \xrightarrow{d_{k-1}} n_k = p, \\ & \textit{for } 1 \le i < k, \\ & d_i \in \{cd, dd, id\} \cup \{dd_l | l \textit{ a loop in } G\} \\ & \mathcal{P} \text{ is a trace witness in } \; G\}\end{aligned}$$

The algorithm borrows the notation of Krinke. It takes as input the slicing criterion and the TPDG. It is a standard work-list based iterative algorithm that incrementally computes the slice. It starts from the criterion node and traverses backwards along the data, loop-carried data, control and interference dependence edges, successively adding the nodes encountered to the partial slice computed. All nodes reached via only data or control dependence edges can be simply added to the slice as trace witnesses exist for these nodes (by definition of data and control dependence). Care has to be taken when a node is reached via an interference edge. In the example in Figure 3.1(b), if the node S_4 is the slicing criterion, we find that S_4 is interference dependent on S_8 which is interference dependent on S_5. Thus it is possible to reach a node via one or more interference nodes from the slicing criterion for which there does not exist a trace witness. This problem is further compounded in the presence of nested threads.

In the program in Figure 3.1(a), S_4 in θ_1 is interference dependent on S_8 in θ_2. S_8 is interference dependent on S_6 in θ_3. All three nodes lie in the region R' lying between the `cobegin` node c_1 and its corresponding `coend` node e_1. Of these, S_8 executes in parallel with both S_4 and S_6, but S_4 executes sequentially with S_6. For $\langle S_6, S_8, S_4 \rangle$ to be a valid trace witness there must be a path in R' from S_6 to S_4 (by the rule for trace in a `cobegin` statement). So when the algorithm visits the node S_4 in θ_1, this information must be noted by all threads that execute sequentially with θ_1.

To determine the existence of trace witness for the nodes, a tuple of nodes containing one node per thread is maintained. In a tuple, $T = [t_0, t_1, \cdots, t_n]$ the node, t_i, corresponding to the thread, θ_i, is either the last node reached (so far) in that thread or in a nonconcurrent thread. When a node S_i in thread θ_i is visited, this information is made visible to *all threads that do not execute in parallel with* θ_i. For example, if θ_i and θ_j are two threads that do not execute in parallel, then the new tuple generated will be $T = [t_0, \cdots, t_i = S_i, \cdots, t_j = S_i, \cdots, t_n]$. Later if the algorithm visits a node, S_j, in θ_j we check if there is a path from S_j to S_i. If yes, S_j is added to the slice else it is rejected.

The algorithm for computing the slice is given in Figure 3.5,[1] where we use the notation $T[i] = x$ to replace the i^{th} element of a tuple T by x and the notation $x = T[i]$ to extract into x the i^{th} element of the tuple T.

[1]For ease of understanding, we present here the simplest form of the algorithm before the application of the optimizations described in Section 3.4

The algorithm takes as input the slicing criterion, p and the TPDG. At line [1], the initial tuple is constructed, setting every component to p if the corresponding thread does not execute in parallel with $\theta(p)$ or to $\perp$ otherwise.

The the algorithm repeatedly extracts a node, x and its corresponding tuple T from the worklist (line[5]).

At line [6], the algorithm finds from the TPDG, every node y that is data or control dependent on x. The it modifies the tuple T such that $T[i]$ is set to y for every thread that does not execute in parallel with $\theta(y)$. If the pair (y, T) are a new pair, then it is added to the worklist.

At line [14], the algorithm finds from the TPDG, every node y that is interference dependent on x. The variable t holds the value of the node in T corresponding to $\theta(y)$. R is the closest enclosing region that contains both x and y. If t is $\perp$, then y can be added to the slice. Or if t does not belong to the region R this implies that t executes sequentially with both x and y and hence y may be added to the slice. Or if t belongs to R and there is a path in R such that the apth lies within R then y can be added to the slice.

At line [25], the algorithm finds from the TPDG, every node y that is loop carried data dependen on x such that the loop carried dependence crosses a thread boundary. For every thread, i, that lies within the loop the tuple T is modified by setting $T[i]$ set to the source of the backedge of the loop. Then for every thread i that does not execute in parallel with $\theta(y)$, $T[i]$ is set to y.

Data Dependence and Control Dependence Edges. Consider a data or control dependence edge from S_x to S_y. By definition of the dependence, S_x and S_y must execute in sequence. So handling data or control dependence is simple. In the tuple, T, we only need to set to S_x every element that does not execute in parallel with the thread $\theta(x)$.

Interference Dependence Edges. Let S_x and S_y be two nodes such that $||(S_x, S_y)$ is true. Let $c_1 = CoAncestor(S_x, S_y)$ and let $R = CoRegion(S_x, S_y)$. Consider an interference dependence edge from S_x to S_y. By definition of trace witness for a `cobegin-coend` region, $\langle S_x, S_y \rangle$ is a valid trace witness in R. Let S_z be some node that does not execute in parallel with S_x. We wish to find out if $\langle S_x, S_y, S_z \rangle$ is also a valid trace witness. If $S_z \in R$, then by the definition of a trace for a `cobegin-coend` region there must be a path in R from S_x to S_z for $\langle S_x, S_y, S_z \rangle$ to be a valid trace

Input: the slicing criterion p
Output: $\mathcal{S}$, a set of nodes of the TPDG

Initialize the worklist with an initial state tuple:

$C = (p, [t_0, \ldots, t_n])$ **s.t.** $t_i = \begin{cases} p & \text{if } ||(\theta(p), \theta_i) \text{ is } \textbf{false} \\ \perp & \text{else} \end{cases}$
worklist $w = \{C\}$
slice $\mathcal{S} = \{p\}$
repeat
    remove the next element $c = (x, T)$ from w
    /* *Examine all reaching edges:* */
    for all $y \mid (y, x) \in E_{dd} \cup E_{cd}$ **do**
        for all i s.t. $||(\theta_i, \theta(y))$ is **false do**
            $T[i] = y$
        $c' = (y, T)$
        if c' has not already been calculated **then**
            mark c' as calculated
            $w = w \cup \{c'\}$
            $\mathcal{S} = \mathcal{S} \cup \{y\}$
    for all $y \mid (y, x) \in E_{id}$ **do**
        $t = T[\theta(y)]$
        $R = CoRegion(x, y)$
        if $t == \perp$ **or** $t \notin R$ **or** $Path(y, t, R) ==$ **true then**
            for all i s.t. $||(\theta_i, \theta(y))$ is **false do**
                $T[i] = y$
            $c' = (y, T)$
            if c' has not been already calculated **then**
                mark c' as calculated
                $w = w \cup \{c'\}$
                $\mathcal{S} = \mathcal{S} \cup \{y\}$
    for all $y \mid (y, x) \in E^l_{dd}$ **do**
        for all i s.t. $\theta_i \in l$ **do**
            $T[i] =$ source of backedge of l
        for all i s.t. $||(\theta_i, \theta(y))$ is **false do**
            $T[i] = y$
        $c' = (y, T)$
        if c' has not already been calculated **then**
            mark c' as calculated
            $w = w \cup \{c'\}$
            $\mathcal{S} = \mathcal{S} \cup \{y\}$
until worklist w is empty.

Figure 3.5: The Intraprocedural Slicing Algorithm for Concurrent Programs

witness. If $S_z \notin R$, then $\langle S_x, S_y, S_z\rangle$ is a valid trace witness in G if there is a path from S_x to S_z in G.

To determine the validity of an interference dependence edge from S_y to S_x, we need to apply the test $Path(S_x, S_z, R)$, where R is the CoRegion of S_x and S_y.

Loop-carried dependence. As we had noted earlier, a loop-carried dependence that crosses thread boundaries implies that the old thread has exited and a new invocation of the thread now exists. In Figure 3.4, we see that the loop-carried data dependence edge from S_2 to S_3 implicitly assumes a (backward) path from S_2 to the `cobegin` node in θ_0 to S_0 to S_5, and then subsequently to S_3 in θ_2. S_5 is in θ_0 and obviously executes sequentially with all threads within the loop and hence all the threads need to be marked with this node. Subsequently, when the edge reaches its destination, S_3, it will overwrite all threads that execute sequentially with θ_2 to the value S_3.

Thus, when traversing a loop-carried data dependence edge the state of the tuple associated with each thread within the loop is set to the source of the backedge corresponding to the loop that induces the loop-carried data dependence (lines #26-27). Then at lines #28-29, the algorithm sets the tuple element corresponding to each thread that does not execute in parallel with the destination of the loop-carried data dependence edge.

Example 1 To illustrate the algorithm, we first consider the example with nested threads shown in Figure 3.6. There are seven threads in the program and let S_4 in θ_3 be the slicing criterion. The algorithm starts by initializing the worklist with the pair $(S_4, [S_4, S_4, \perp, S_4, \perp, S_4, S_4])$. Here S_4 is the node to be sliced and θ_3 can execute sequentially with θ_0, θ_1, θ_3, θ_5 and θ_6. First we consider data and control dependence edges that reach S_4. As S_4 is data dependent on S_1 we add the following pair to the worklist : $(S_1, [S_1, S_1, \perp, S_1, S_1, S_1, S_1])$. We mark every thread that may execute sequentially with θ_1 (including θ_1). Control dependence is handled similarly.

Now consider the interference dependence edge from S_7 to S_1. S_7 belongs to thread θ_2. The state of the θ_2 is $\perp$ in the tuple associated with S_1. This implies that the algorithm has not yet visited θ_2 or any other thread that executes sequentially with θ_2. So S_7 is added to the worklist as $(S_7, [S_7, S_1, S_7, S_1, S_1, S_1, S_1])$. The only other thread that executes sequentially with θ_2 is θ_0. The state of the other threads

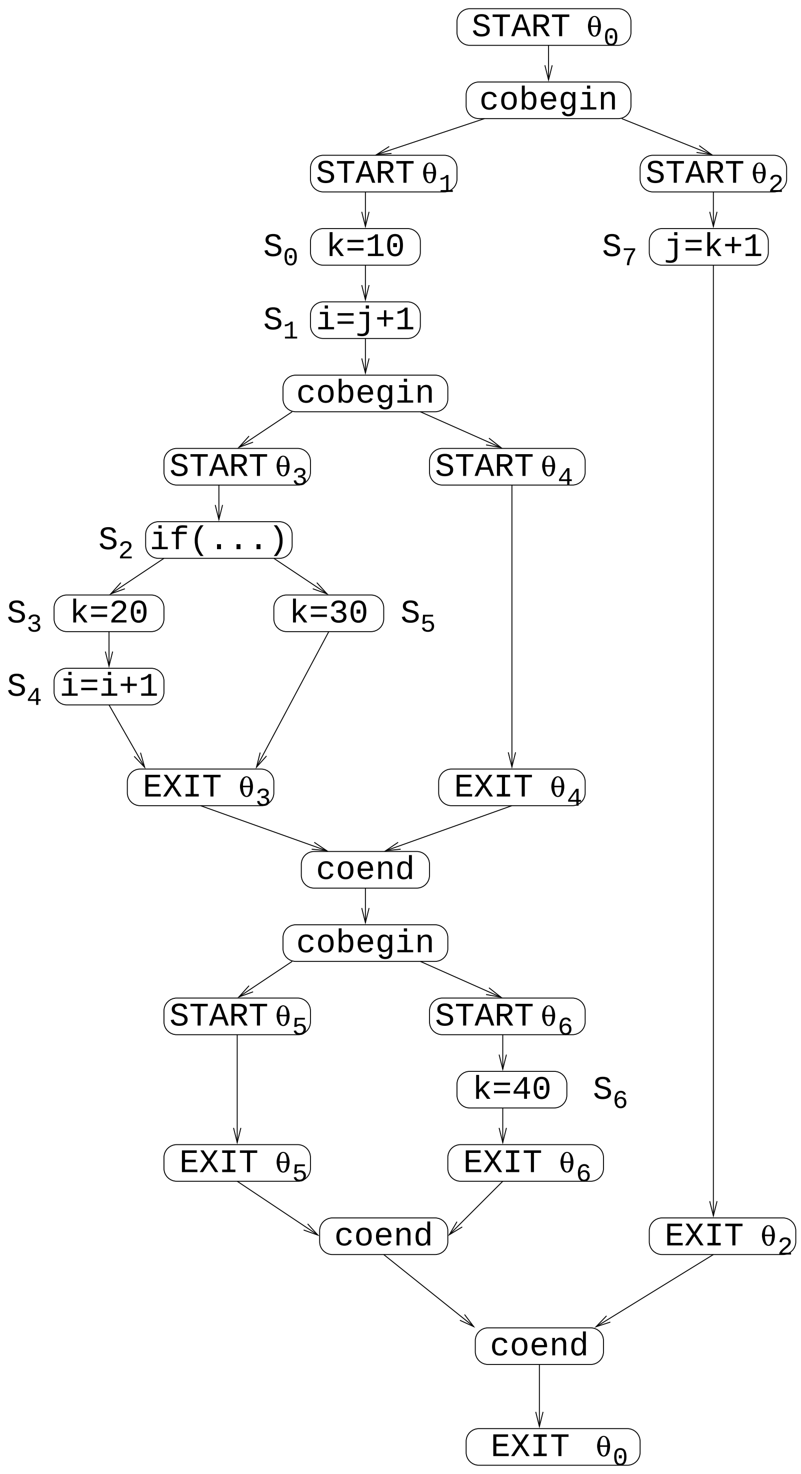


Figure 3.6: Nested threads

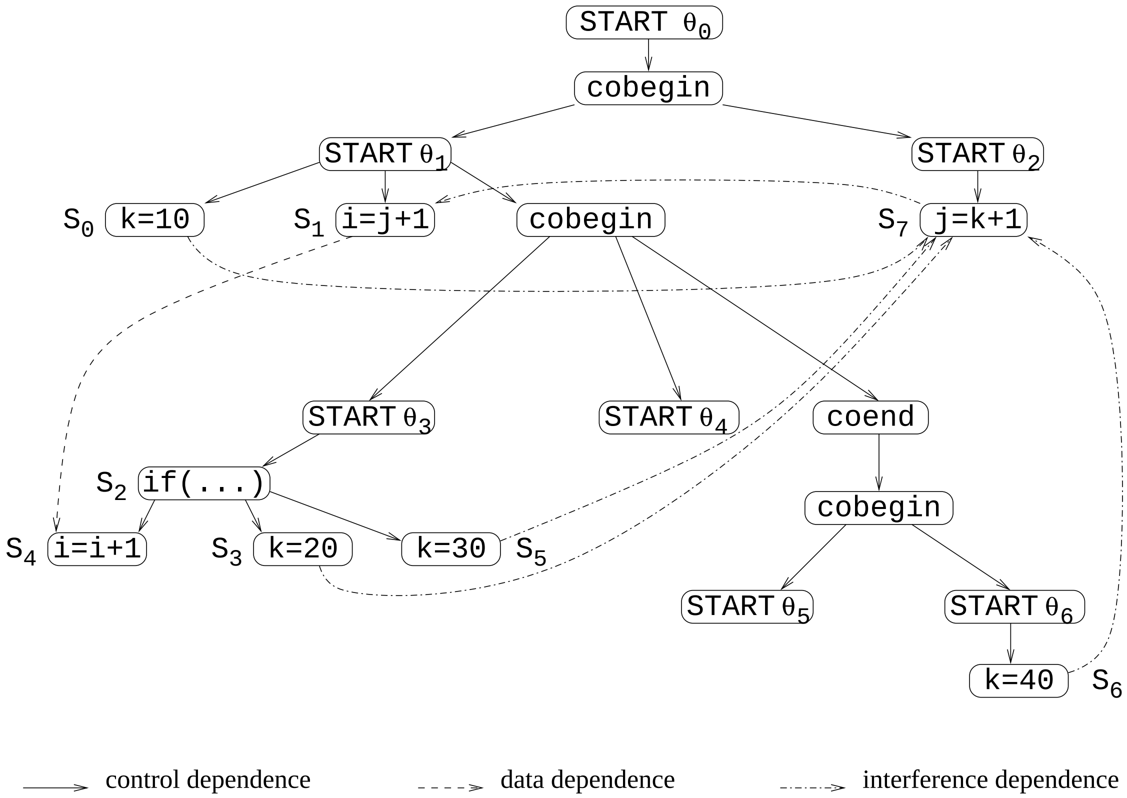


Figure 3.7: PDG for program in Figure 3.6

does not change. Further S_7 is interference dependent on S_0, S_3, S_5 and S_6. Let us consider each edge.

S_0: S_0 belongs to θ_1. The state of θ_1 in the tuple associated with S_7 is S_1. There is a path from S_0 to S_1 and hence S_0 is added to the slice.

S_3: S_3 belongs to θ_3. The state of θ_3 in the tuple associated with S_7 is S_1. There is no path from S_3 to S_1 and hence S_3 is *not* added to the slice. It is interesting to note that although S_4 seems to have a transitive dependence on S_3 and there is a path from S_3 to S_4, the value computed at S_3 cannot affect the value of i at S_4 since the path $S_3 \rightarrow S_7 \rightarrow S_1 \rightarrow S_4$ is not a valid path (S_1 cannot execute after S_3).

S_5: S_5 also belongs to θ_3 but there is no path from S_5 to S_1 and hence S_5 is not added to the slice.

S_6: S_6 belongs to θ_6 and the state of θ_6 in the tuple associated with S_7 is S_1. There is no path from S_6 to S_1 in the TCFG and hence S_6 is not added to the slice. That is, although $S_3 \rightarrow S_7 \rightarrow S_1 \rightarrow S_4$ is a valid path in the TPDG, it is not a valid path in the TCFG and hence is not a trace witness.

As a result, the slice of S_4 is computed to be $\{S_0, S_1, S_2, S_4, S_7\}$. The slice computed by Krinke's algorithm would have been $\{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7\}$. Three spurious nodes S_3, S_5 and S_6 are added to his slice.

Example 2. Consider the slice of the node S_2 in the example in Figure 3.2. S_2 is interference dependent on S_6 and S_6 is interference dependent on S_4. However, in the region defined by $CoRegion(S_4, S_6)$ there is *no* path from S_4 to S_2 (the loop edge lies *outside* the region under consideration) and so S_4 is not added to the slice. The only other nodes added to the slice are S_1 (S_2 is data dependent on S_1) and S_5 (S_6 is data dependent on S_5).

Example 3. In Figure 3.3(a), let S_1 be the slicing criterion. S_1 is interference dependent on S_3 and S_3 is interference dependent on S_2. Here also there is no path from S_2 to S_1 in the region defined by $CoRegion(S_3, S_2)$ and so S_2 is *not* added to the slice at this step. However, there is also a loop-carried data dependence edge from S_2 to S_3. Hence S_2 will be correctly added to the slice. The state of both θ_1

and θ_2 is set to S_5, then the state of θ_1 (and all threads that execute sequentially with it) is set to S_2. The resultant element in the worklist will be $(S_2, [S_2, S_2, S_5])$. Next S_2 is interference dependent on S_4 and the state of θ_2 is S_5. S_5 does not lie in $CoRegion(S_2, S_4)$ and hence S_4 is also added to the slice (there is a path from S_4 to S_5). This represents the valid sequence of execution $\langle \cdots \rightarrow S_4 \rightarrow S_2 \rightarrow S_3 \rightarrow S_2 \rightarrow \cdots \rangle$.

The algorithm calculates correct and precise slices even in the presence of arbitrarily nested threads and loops. Our algorithm ensures that paths over interference edges are always trace witnesses. Let p be the slicing criterion in a given TPDG; let S be the slice computed by the algorithm. Then the correctness of the algorithm can be stated as

Theorem 1 $\mathcal{S}(p) = S$

A sketch of the proof of this theorem is given in the Appendix.

3.4 Complexity and Optimizations

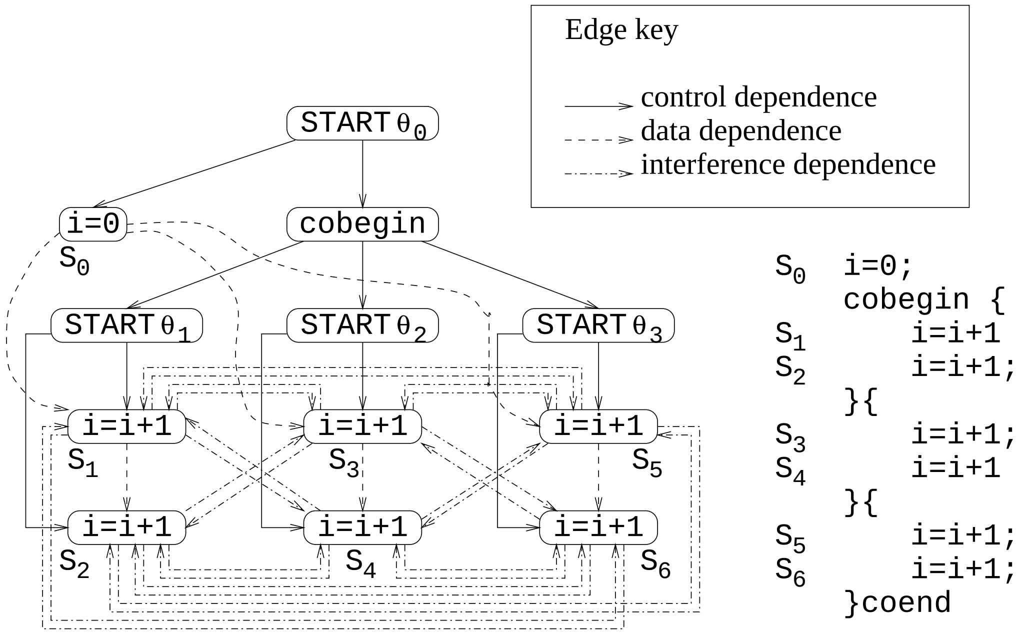


Figure 3.8: Another threaded PDG

The number of interference dependence edges can be quadratic in the number of nodes as shown in Figure 3.8 and in general, the number of times a node is added to the worklist can be exponential in the number of threads.

To slice S_2 in Figure 3.8, the algorithm would find that S_2 is data dependent on S_1 and insert S_1 with the tuple $[S_1, S_1, \perp, \perp]$. Next the algorithm may determine that S_2 is interference dependent on S_3 which is interference dependent on S_1. S_1 would be inserted into the worklist again with the new tuple $[S_1, S_1, S_3, \perp]$. Let the tuple $[t_0, t_1, t_2, t_3]$ be the tuple associated with S_1. The value of t_0 is always S_1; the value of t_1 is S_1; the value of t_2 could be $\perp$, S_3 or S_4 depending on the path taken; and the value of t_3 could be $\perp$, S_5 or S_6. Taking into account all the permutations the number of times S_1 may be added to the worklist is $1 * 1 * (n_2 + 1) * (n_3 + 1)$, where n_i represents the number of nodes in thread θ_i. In general, the worst case complexity of the algorithm is $O(N^t)$, where N is the number of nodes in the graph and t is the number of threads.

Although the algorithm has exponential complexity there are many optimizations that can be applied to make it practically efficient. We have implemented this algorithm with the various optimizations on a subset of Java where all thread communication is done with volatile shared variables.

Optimization 1. The first optimization involves the path reachability function $Path(i, j, R)$. There are two ways to implement $Path$. The first is to traverse the TCFG each time the function is called. The second is to summarize the $Path$ relation in bit vectors using data flow analysis. Since $Path$ needs to be applied to different regions at different times, the first option requires identifying the region, then isolating it, followed by an $O(N)$ traversal. The second option is much faster, requiring the lookup of a single bit but has the disadvantage of requiring large amounts of space. Since we need different bit vectors for different regions, this only compounds the space problem. Yet we opt for the second method with the following modification.

We use an interval-based approach [18] to generate the $Path$ bit vectors. This requires reducing each strongly connected region (SCR) in the TCFG to a single node. The resultant graph is acyclic and the $Path$ bit vectors can be found during a single topological order traversal of the reduced graph. Since every node within a strongly connected region is reachable from every other node in the same region, the bit vectors need to be only large enough to hold one bit for each SCR. Although we still need to maintain multiple bit vector sets, we found this very effective. This is not surprising, as a walk through any program shows that most programs spend most of their time inside loops.

Optimization 2. If a node S_i in θ_i is data or control dependent on node S_j in thread θ_j and $\theta_i \neq \theta_j$, then in the state tuple associated with S_j, we can reset the state of every thread θ_k (set the corresponding entry into $\perp$) for which the following conditions hold :

- $||(\theta_j, \theta_k)$ is false
- there is *no* thread θ_x such that $||(\theta_j, \theta_x)$ is true and $||(\theta_k, \theta_x)$ is true. That is, there is no thread that executes in parallel with both θ_j and θ_k.

This test is static for a given graph and can be precalculated and stored in small bit vectors with one bit per thread. In particular, the main thread does not execute parallel to any other thread. By this optimization, a node in this thread will not be inserted into the worklist more than once.

Optimization 3. Consider the program in Figure 3.8. In finding the slice of S_2 let us assume that the data dependence edge to S_1 is traced first. The state tuple associated with S_1 would be $[S_1, S_1, \perp, \perp]$. Since the state tuple associated with θ_2 is $\perp$, this implies that S_1 can reach any node in θ_2 without restriction. Next let us trace the path backward from S_2 to S_3 to S_1. This time the state tuple associated with S_1 would be $[S_1, S_1, S_3, \perp]$. Any path traced with this tuple will be a subset of the paths generated by the previous tuple.

Similarly, if S_1 were inserted into the worklist with the tuple $[S_1, S_1, S_4, \perp]$ first and later with a tuple of $[S_1,\ S_1,\ S_3,\ \perp]$ then the second tuple is redundant since there is a path from S_3 to S_4. This implies that any node that may be reached from S_4 may be reached from S_3 and hence this tuple can only generate a subset of the paths generated by the first tuple.

These two results can be summarized as follows : given that a node has been inserted into the worklist with a tuple $[t_0, t_1, \ldots, t_n]$, if it gets a new tuple $[t'_0, t'_1, \ldots, t'_n]$ where t_i is equal to t'_i for all i except $i = k$ and t_k is $\perp$ or there is a path from t'_k to t_k then the new tuple is redundant.

However, there is no guarantee that S_1 will be inserted with a less restricted tuple first. To ensure that such does happen, we give nodes reached by interference dependence edges a lower priority by maintaining a two level priority queue in the worklist. Nodes reached through data or control dependence are inserted into the high priority queue and nodes reached through interference are inserted into the low priority queue. Nodes from the low priority queue are extracted for slicing only if the high priority queue is empty.

3.5 Statistics

In practice with the first optimization, we found that we could get upto 50% increase in the speed. The second optimization gave as much as 100% to 150% increase in speed on our sample programs, depending on the number of threads in the program. This is not surprising as all paths must terminate in thread θ_0 and hence the number of times these nodes would have been inserted into the worklist is truly exponential in the number of threads. With the third optimization the "worst case" example of Figure 3.8 actually runs in linear time as each node is inserted exactly once into the worklist. The complexity of the algorithm is still exponential but in general, we have found that many practical problems run in near linear time.

We ran our slicer on 333 MHz Pentium II machine with 64 MB RAM running Linux. Without optimizations, the algorithm is obviously exponential and running more than 3 or 4 threads required more than half an hour of processor time. With the complete optimization, we could run a number of threads efficiently. We tried out the slicer on some classical programs including dining philosophers, producer consumer, reader writer, parallel shortest path.

The performance statistics for two of the programs are given in Figure 3.9. In the table, t represents the number of threads, N is the total number of nodes in the TCFG, $\mathcal{S}$ represents the number of nodes in the slice, w represents the number of elements inserted into the worklist and s is the time in seconds. Cases that took more than an half an hour to run have been marked with a '-'. As already explained, each node may be entered into the worklist more than once. A measure of the effectiveness of the optimizations is the maximum number of times a node is inserted into the worklist. This is represented by m in the table. O_0, O_1, O_2, and O_3 represent the program with no optimization, the first, the first two and all optimizations respectively.

Without optimization, the time taken increases exponentially and this is also

Figure 3.9: Statistics

	O_0	O_1	O_2	O_3
$t = 4$, $N = 313$, $\mathcal{S} = 57$				
s	0.17	0.07	0.05	0.0
w	2028	2028	1257	57
m	388	388	25	1
$t = 5$, $N = 409$, $\mathcal{S} = 83$				
s	124.56	113.08	51.5	0.0
w	72055	72055	46883	83
m	12589	12589	625	1
$t = 11$, $N = 985$, $\mathcal{S} = 239$				
s	-	-	-	0.05
w	-	-	-	239
m	-	-	-	1
$t = 21$, $N = 1945$, $\mathcal{S} = 499$				
s	-	-	-	0.3
w	-	-	-	499
m	-	-	-	1
$t = 31$, $N = 2905$, $\mathcal{S} = 759$				
s	-	-	-	0.89
w	-	-	-	759
m	-	-	-	1

(a) Dining Philosophers

	O_0	O_1	O_2	O_3
$t = 3$, $N = 310$, $\mathcal{S} = 158$				
s	0.25	0.13	0.04	0.01
w	4165	4165	2003	162
m	964	964	27	2
$t = 4$, $N = 454$, $\mathcal{S} = 262$				
s	345.02	335.62	28.22	0.01
w	159673	159673	7823	317
m	34371	34371	802	4
$t = 11$, $N = 1462$, $\mathcal{S} = 955$				
s	-	-	-	0.24
w	-	-	-	2858
m	-	-	-	18
$t = 21$, $N = 2902$, $\mathcal{S} = 1945$				
s	-	-	-	4.3
w	-	-	-	10908
m	-	-	-	38
$t = 31$, $N = 4342$, $\mathcal{S} = 2935$				
s	-	-	-	30.88
w	-	-	-	24158
m	-	-	-	58

(b) Producer Consumer

reflected in the values of w and m. In the dining philosophers problem, after all optimizations, the number of nodes handled by the worklist is exactly the number of nodes in the slice.

In the producer consumer problem we found that the number of nodes being inserted into the worklist was increasing linearly with the number of threads and hence the algorithm gave a performance of $O(Nt^2)$. This is much better than the exponential performance of the unoptimized algorithm.

3.6 Related Work

Static slicing is an established method of analyzing sequential programs, especially for program understanding, debugging and testing. But today even small programs use parallelism and a method to slice such programs is required.

Duesterwald *et al* [33] extended the concept of slicing to distributed programs. They presented a parallel algorithm to compute dynamic slices for distributed programs. They model a system with synchronous communication and no shared memory. They generate statically the control dependences of each distributed program and then dynamically generate the data and communication dependence for a specific set of input values. They present a distributed slicing algorithm where each process contributes by determining the local statements in the slice. Communication between processes takes place when a send or receive node is included in a local slice by some process.

Cheng [23] developed a static slicer for concurrent programs that supports synchronous communication. He extends the system dependence graph for sequential programs to include dependence edges to handle communication. He extends the slicing algorithm for sequential programs and presents a graph reachability solution. His solution does not model shared memory. Later, he combined this model with a slicing technique for object-oriented languages [57] to derive a static slicer for concurrent object-oriented languages, but did not extend the model to include shared memory. Zhao [92, 94] also presents a similar graph representation for concurrent object-oriented languages. Neither Cheng nor Zhao handle the issue of intransitivity of interference dependence.

Hatcliff *et al* [46] analyze Java programs with monitors and synchronization. They introduce several new dependencies including a *ready dependence*, where a node, S_i, is ready dependent on another node, S_j, if S_j's failure to complete could imply that S_i never executes. For example, a `wait` statement is ready dependent on the corresponding `notify` statement and all statements that may be reached from a `wait` statement are ready dependent on the `wait` statement. In our model, we would be required to include a ready dependence from a `coend` node to all statements that may be reached from the `coend` node. Our algorithm can be extended to include these dependencies. However, although Hatcliff *et al* introduce several new dependencies and can handle `wait/notify` synchronization, they also ignore the imprecision introduced by the intransitivity of interference dependence.

Krinke [53] gives a precise algorithm for concurrent programs with shared memory but his solution is imprecise in the presence of nested threads or threads nested within loops. None of the authors give implementation details or statistics.

3.7 Conclusion

In this chapter we have given a formal definition of the slice of a concurrent program. We have then given an algorithm that generates correct and precise slices. We show that the algorithm is exponential in complexity. However, with certain optimizations, we find that many practical concurrent problems execute in near linear time.

Limitations. In order to handle the mutual exclusion due to the `synchronized` clause of Java we used the techniques of Novillo *et al* [69] as follows: Each synchronized block of code was replaced by a single statement represented by the set of "upward exposed" REFs and "downward exposed" DEFs.

In our analysis, we have assumed that atomicity is at the level of a memory read or write. In programs with statement level atomicity or coarser atomicity data dependence may not be transitive. Consider the program

(θ_0) S_1: $x = 10$;
cobegin {
(θ_1) S_2: $x = x + 1$;
}{
(θ_2) S_3: $x = 20$;
} coend;
(θ_0) S_4: $y = x$;

Here S_4 is data dependent on S_2 and S_2 is data dependent on S_1. However, the value computed at S_1 is killed by S_3 and hence S_4 is not transitively dependent on S_1. One would expect the program to output a value of 20 if S_3 executes after S_2 or a value of 21 if S_2 executes after S_3. However, if the statements are not properly synchronized, the program may also output the unexpected value of 11 since Java does not providestatement level atomicity. However, if the statements are synchronized then S_4 cannot be transitively dependent on S_1. This is an interesting problem that our analysis does not handle yet.

3.8 Appendix

3.8.1 Calculating Loop Carried Data Dependence

We first review the concepts of loop-carried and loop-independent data dependence in the context of sequential programs. For scalar variables, a loop-independent data dependence from S_i to S_j is a a *forward* data dependence (*i.e.*, there is a path from S_i to S_j without any back-edges) and a loop-carried data dependence from S_i to S_j is a *backward* data dependence (the path from S_i to S_j includes a back-edge). For vectors there are standard techniques [5] that determine loop-carried and loop-independent data dependence. However, in this chapter, we do not further discuss vectors. The test for scalar variables needs to be revisited as there is no direct analogy with forward and backward dependences when the dependence is between parallel executing statements.

For our program model, data dependence is calculated by determining reaching definitions using the 'inside-out, outside-in' algorithm of Knoop *at al* [51, 87]. The algorithm uses bitvectors and at the end of the analysis the definitions reaching a node S_i are stored in $in(S_i)$, the definitions available after S_i executes are in $out(S_i)$. Let REF(S_i) and DEF(S_i) refer to the set of variables referenced or defined at S_i, respectively and KILL(S_i) refer to the set of definitions killed at S_i. A data dependence exists from S_i to S_j if $S_i \in in(S_j)$ and $\text{DEF}(S_i) \cap REF(S_j) \neq \emptyset$. A data dependence edge is drawn from S_i to S_j if $||(\theta(S_i), \theta(S_j))$ is false, else an interference dependence edge is is drawn from S_i to S_j.

Although the data flow analysis identifies all loop-carried dependences the analysis is unable to *distinguish* between interference dependence and a loop-carried data dependence. In the bitvectors the loop-carried data dependence is masked by the direct interference dependence. Hence it is impossible to tell from the bit vector information alone whether a loop-carried data dependence exists or not. In order to find the loop-carried data dependence edges, we essentially find all the definitions generated within the loop that reach the head of the loop and subsequently reach some *use* within the loop body. We give below an algorithm to identify loop-carried data dependence edges.

As we are dealing with a structured language, all loops and threads are either properly nested or disjoint. Loop carried data dependences that do not cross thread boundaries are treated as simple data dependences. Working from outer loops to inner loops, for each loop, l that contains a thread nested within we do the following:

- Let H be the loop header and let R be the region induced by the loop. We isolate the region by temporarily marking all entry and exit edges of R as 'defunct'.

- Find D the set of definitions generated within R. Let S be the source of the back edge of the loop. Calculate $D = D \cap out(S)$. Then D has the set of definitions generated within the loop that can re-enter the loop on a subsequent iteration.

- Do a depth first traversal of R and mark all back edges within the region as 'defunct'. The nodes in R can now be sorted in topological order.

- Apply the algorithm in Figure 3.10 to R, where $pred(n)$ is an immediate predecessor of n that has not been marked as 'defunct'. In the algorithm $p \xrightarrow{dd_{l'}} n$ refers to a loop carried data dependence edge associated with an outer loop and $p \xrightarrow{dd_l} n$ refers to a loop carried data dependence edge associated with the current loop.

- Restore the edges that were marked 'defunct'.

Input

H: the loop header
R: the region in the TCFG induced by the loop
S: the source of the back edge of the loop

Output

The loop carried data dependence edges

for each node $n \in R$ in topological order **do**
  if $n == H$ **then**
    $In'(n) = D$
  else if $n.type ==$ COEND **then**
    $In'(n) = \cap_{p \in pred(n)} Out'(p)$
  else /* all other nodes */
    $In'(n) = \cup_{p \in pred(n)} Out'(p)$
  for each $p \in In'(n)$ s.t. DEF(p) $\cap$ REF(n) $\neq \emptyset$ **do**
    if $||(n, p) ==$ **true then**
      if there is no edge $p \xrightarrow{dd_{l'}} n$ **then**
        Create $p \xrightarrow{dd_l} n$
    else /* **if** $||(n, p) ==$ **false** */
      if there is no edge $p \xrightarrow{dd_{l'}} n$ **then**
        Delete the existing edge $p \xrightarrow{dd} n$
        Create $p \xrightarrow{dd_l} n$
  endfor
  $Out'(n) = In'(n) - KILL(n)$
endfor

Figure 3.10: Algorithm to find Loop Carried Data Dependence

3.8.2 Proof for the Slicing Algorithm

Proof of $S \subseteq \mathcal{S}(p)$:
We need to show that for every node q added to the slice, there is a trace witness $\{q = q_0,\, q_1,\, q_2,\, \cdots,\, q_n = p\}$. Let $\theta(q) = \theta_i$. Suppose q is added to S during the n^{th} iteration of the loop in the algorithm. We prove by induction on n that q is in $\mathcal{S}(p)$.

Base: $n = 1$. Then q is either directly data dependent, loop-carried data dependent, control dependent or interference dependent on p. Then obviously, q is in $\mathcal{S}(p)$, since (q, p) is a trace witness.

Induction Step: Suppose $n = (k + 1)$ for some $k \geq 0$. Then q is directly data, loop-carried data, control or interference dependent on some node q_i that is already added to S in k or less steps. By induction hypothesis, q_i is in $\mathcal{S}(p)$ and q_i has a trace witness, say, σ. The following cases can be distinguished depending upon the type of dependency in $q \rightarrow q_i$:

- Case *cd* or *dd* or dd_l: Then by definition of data dependence, loop-carried data dependence and control dependence, $q.\sigma$, the sequence obtained by prefixing q to σ is a trace witness.
- Case *id*: Let (q_i, T) be the element in the worklist which when processed, the node q was added to $\mathcal{S}$ (in step 24 of the algorithm). There are two cases to be distinguished:
  1. $T[i] = \perp$: This means that every node in σ executes in parallel with q. Then $q.\sigma$ is a trace witness for q.
  2. $T[i] = q'$: Let $R = CoRegion(q,\, q_i)$. Here again there are two cases :
     (a) $q' \in R$. Then by the definition of a trace for `cobegin` statements there must be a path from q to q' within R. We ensure this with the test $Path(q, q', R)$.
     (b) $q' \notin R$. Clearly there is a path from q_i to q' and hence a trace witness exists.

Proof of $\mathcal{S}(p) \subseteq S$:
The proof requires the following fact: Every node added to the worklist is eventually removed from the worklist and each of its predecessors is considered for inclusion in

the slice. This follows from the fact that there is only a finite number of tuples and each tuple is added exactly once to the worklist.
Now we need to show that for every trace witness $\langle q = q_0,\ q_1,\ \ldots, q_n = p\rangle$, q_0 is added to the slice.
For a witness of size two: $q \rightarrow p$, it is easy to see that q will be added to the slice.

Consider the trace witness of length $n+1$ given by $\langle q_0 \rightarrow q_1 \rightarrow \ldots \rightarrow q_n = p\rangle$ By induction hypothesis, all the nodes $q_1, q_2, \ldots, p$ are in S.

It is clear that eventually q_1 will be removed from the worklist and q_0 will be considered for inclusion in the slice. q_0 will be added to the slice depending upon the type of edges between q_0 and q_1 as argued below:

- Case cd or dd or dd_l : Either q is already in the slice or it is trivially added to the slice (line #13, #34).
- Case id : Let Θ be the set of threads that do not execute in parallel with $\theta(q_0)$. There are two possibilities here
  1. q_0 is the first node to be added to the slice that belongs to any thread in Θ. Then the state tuple associated with q_1 will have $\perp$ for $\theta(q_0)$ and q_0 will be added to the slice (line #17).
  2. Let q_k be the last node visited by the algorithm that belongs to any thread in Θ. Then the state tuple associated with q_1 will have q_k for $\theta(q_0)$. Let $R = CoRegion(q_0,\ q_1)$.
     - If $q_k \in R$: By the definition of a trace witness, there is a path from q_0 to q_k and hence q_0 is added to the slice (line #17).
     - If $q_k \notin R$: q_0 is added to the slice (line #17).

Thus every trace witness is added to the slice.
Hence the theorem.

Chapter 4

INTERPROCEDURAL REALIZABLE PATHS

In Chapter 3 we used a procedure $Path(i, j, R)$ to determine whether there was a path from the node i to the node j. In interprocedural programs this reachability information is complicated by the presence of procedure calls. In this chapter we introduce the concept of a "realizable" path in an interprocedural program and give an algorithm to determine whether there is a realizable path from one node to another.

4.1 Introduction

An interprocedural program consists of a `main` procedure and a set of procedures that are called directly or transitively by the `main` procedure. Recursion is allowed where a procedure may be directly or indirectly invoked by itself. The static block structure of the program is represented by a tree, where the root represents the `main` procedure. Each procedure is represented by a complete control flow graph rooted at the procedure *Entry* node and terminating at the procedure *Return* node. A statement that invokes a procedure is referred to as a *call site*. An edge from the call site to the *Entry* node of the called procedure connects the two control flow graphs together. Procedure return is represented by an edge from the procedure's *Return* node to the statement that lexically follows the call site, and this is termed the return site.

A procedure may be invoked from multiple call sites. Consider the interpro-

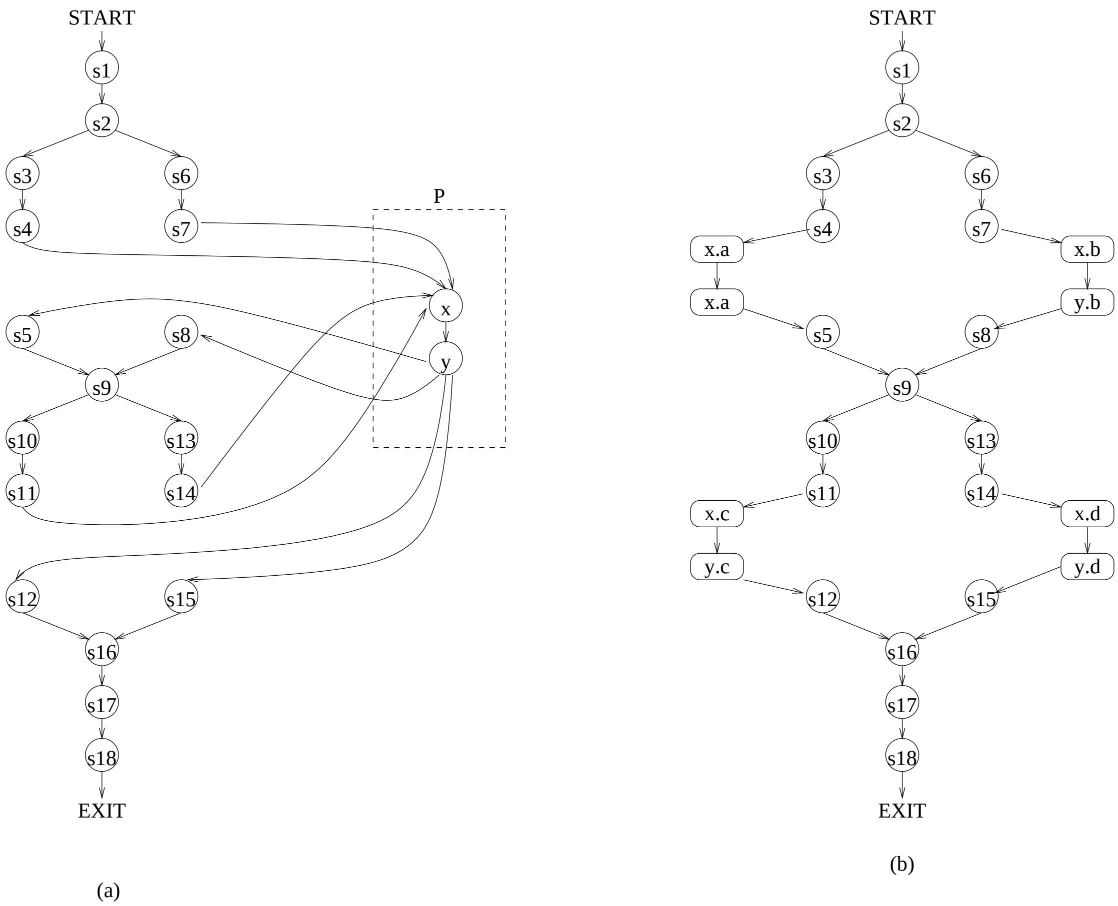


Figure 4.1: (a) A Program CFG and (b) The CFG with Procedures Inlined

cedural control flow graph in Figure 4.1(a). In this example the procedure P is called from four different call sites s_4, s_7, s_{11} and s_{14} in the main program and the corresponding return sites are s_5, s_8, s_{12} and s_{15}, respectively. A *realizable* path is one that corresponds to a legal call/return sequence wherein a procedure returns to the site from where it was invoked [75, 81]. Such paths are also termed 'valid' or 'feasible' paths in literature. In Figure 4.1 there is a realizable path from s_3 to s_5 but the path from s_3 to s_8 is not a realizable path as it represents a path where P is invoked from s_4 but returns to s_8.

Determining whether there is a realizable path between any *two* nodes in the CFG is possible using standard techniques of data flow analysis for interprocedural programs [18, 19]. We introduce the notation $Reach(i, j)$ to be **true** if there is a realizable path from i to j and **false** otherwise. For example $Reach(s_3, s_{12})$ and $Reach(s_3, s_{15})$ are true but $Reach(s_3, s_8)$ and $Reach(s_{10}, s_5)$ are false.

In intraprocedural programs, the *Reach* function is transitive. $Reach(a, b)$ **and** $Reach(b, c)$ implies $Reach(a, c)$. In interprocedural programs this is no longer true. For example, in Figure 4.1, $Reach(s_3, x)$ and $Reach(x, s_8)$ but not $Reach(s_3, s_8)$. Further, consider the realizable path $(s_4, x, y, s_5, s_9, s_{10}, s_{11}, x, y, s_{12})$. From this path we see that there is a realizable path from y to x. That is, $Reach(y, x)$ is true. Similarly, $Reach(x, s_9)$ is true and in this case $Reach(y, s_9)$ is also true. It might appear that the concatenation of the path from y to x and the path from x to s_9 may be a realizable path. Yet, there is no realizable path $y \xrightarrow{+} x \xrightarrow{+} s_9$. On the other hand, there is a realizable path $y \xrightarrow{+} x \xrightarrow{+} s_{16}$.

We are now ready to state the problem to be solved: *Given an interprocedural control flow graph, G, and a set of nodes n_1, n_2, $\cdots$, n_k belonging to G we wish to find out if there is a realizable path $n_1 \xrightarrow{+} n_2 \xrightarrow{+} \cdots \xrightarrow{+} n_k$.* We term this problem the "*Realizable Path*" problem.

♣ Note: This realizable path problem differs from other realizable path problems in literature. In *context-sensitive* interprocedural data flow analysis realizable paths are identified using some technique as the matched parenthesis described by Reps [75] and then data flow analysis is performed along only valid paths. Our realizable path problem is the converse of this situation. Here we are given a set of nodes and we wish to identify whether they form a part of a realizable path. A naive solution would generate all possible realizable paths in the graph using some technique such as the matched-parenthesis technique and then check if the given set of nodes forms a subsequence of any one of the realizable paths.

Another solution to the *Realizable Path* problem could be to inline all the procedures. In Figure 4.1 we have shown an interprocedural program with the procedures inlined. Then the problem reduces to the intraprocedural case and can be solved easily. This solution obviously requires unbounded space when there are recursive procedures. Another, not so obvious, problem with this solution is that a given node (for example, x in Figure 4.1) may occur in multiple locations in the expanded graph. To find a realizable path that includes x we need to trace every path through each location of x. In the presence of loops, the length of a path may be unbounded.

In this chapter we propose a solution based on interval analysis.

4.2 An Interval-based Approach

First we introduce some standard terminology. $G = (V, E, r)$ is an interprocedural control flow graph where V represents the set of vertices, E is the set of edges and r is the distinguished start node. A *region* R of G is an interprocedural CFG whose nodes are in G and such that an edge (m, n) of G is in R if and only if m and n are both in R. A node m is an *entry node* of R if there is an edge (v, m) of G such that v is not in R. A *strongly connected region* (SCR) is a region in which every node in it is reachable from every other node. The *interval order* of the nodes of G is the order in which they are visited by a reverse postorder traversal (*i.e.*, postorder traversal on the reverse graph, rooted at the `EXIT` node of the CFG). If G is **acyclic**, then interval order is a topological ordering [18]. Let $NUMBER(i)$ be the topologically ordered number for any node i. Then topological order has the interesting property that $NUMBER(i) < NUMBER(j)$ if i is a predecessor of j in the graph, G.

A *call graph* of a program is a graph where each procedure is uniquely represented by a single node and each call site by a unique edge. An edge (P, Q) in the call graph represents a call site in procedure P that invokes procedure Q.

Since every node in a SCR is reachable from every other node in it, it is obvious that we do not need to test for reachability for nodes within a SCR. This is the basis of our algorithm. We find the largest possible *interprocedural* SCRs in the CFG and collapse them into single nodes. The resultant graph is an acyclic graph that can be ordered in topological order. We use the program in Figure 4.2 as an example. Briefly, our analysis takes the following steps

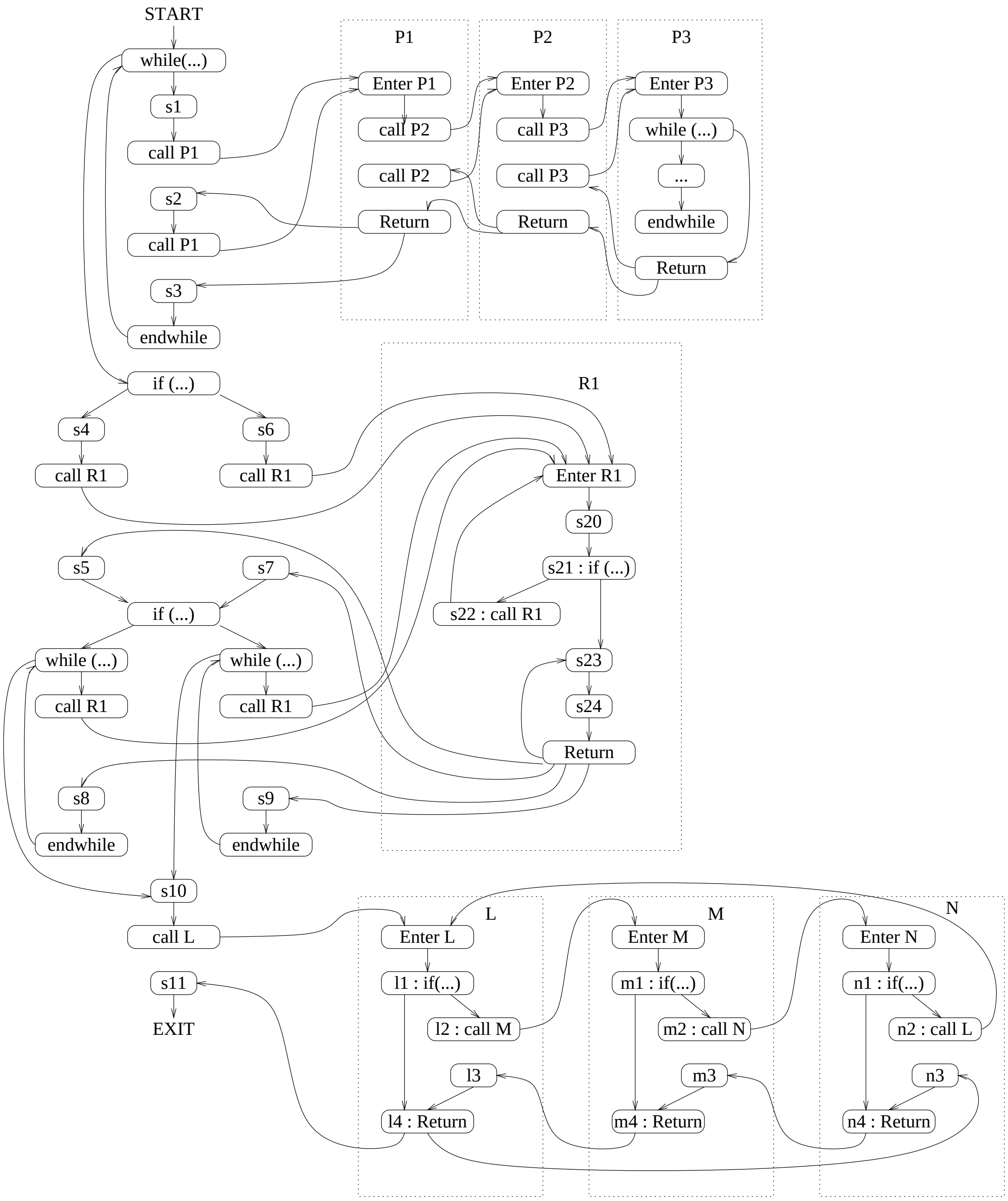
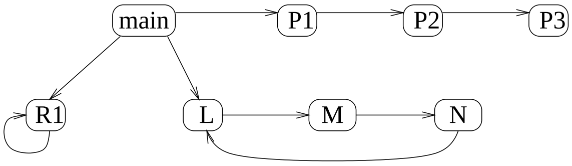
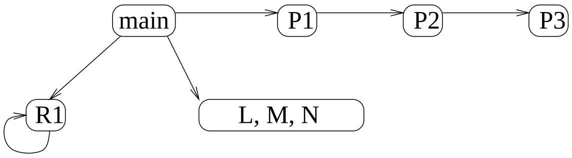


Figure 4.2: The CFG of an Interprocedural Program



(a)



(b)

Figure 4.3: (a) The call graph of the program and (b) the call graph after finding the *call* SCRs (cSCR)

1. First we construct the call graph of the program and find the SCRs in this graph. We term the SCRs in this graph the *call strongly connected components* or *cSCRs*. The cSCRs are shown in Figure 4.3.

2. Next we find the maximal SCRs in each non-recursive procedure and collapse them into a single node each using standard techniques [84, 47]. We call these *intraprocedural strongly connected components* or *iSCRs* for short. They are labelled as χ_i in all the figures. Figure 4.4 shows the iSCRs of the example program.

3. The cSCRs are used for locating recursive procedures. We find the largest possible SCRs induced by the recursive procedures as described in Section 4.2.1. These SCRs are also termed iSCRs.

4. Once the SCRs have been detected at the intraprocedural and call level we give an algorithm for integrating the iSCRs with the cSCRs to construct the final *interprocedural strongly connected components* or *ISCRs*. The ISCRs of the example program are shown in Figure 4.5. The ISCRs are numbered as K_n. The component iSCRs are also shown. The resultant ISCR graph is an interprocedural CFG which has no cycles, no recursion and in which each node represents an ISCR. Next we inline any procedure that has not been integrated into an ISCR.

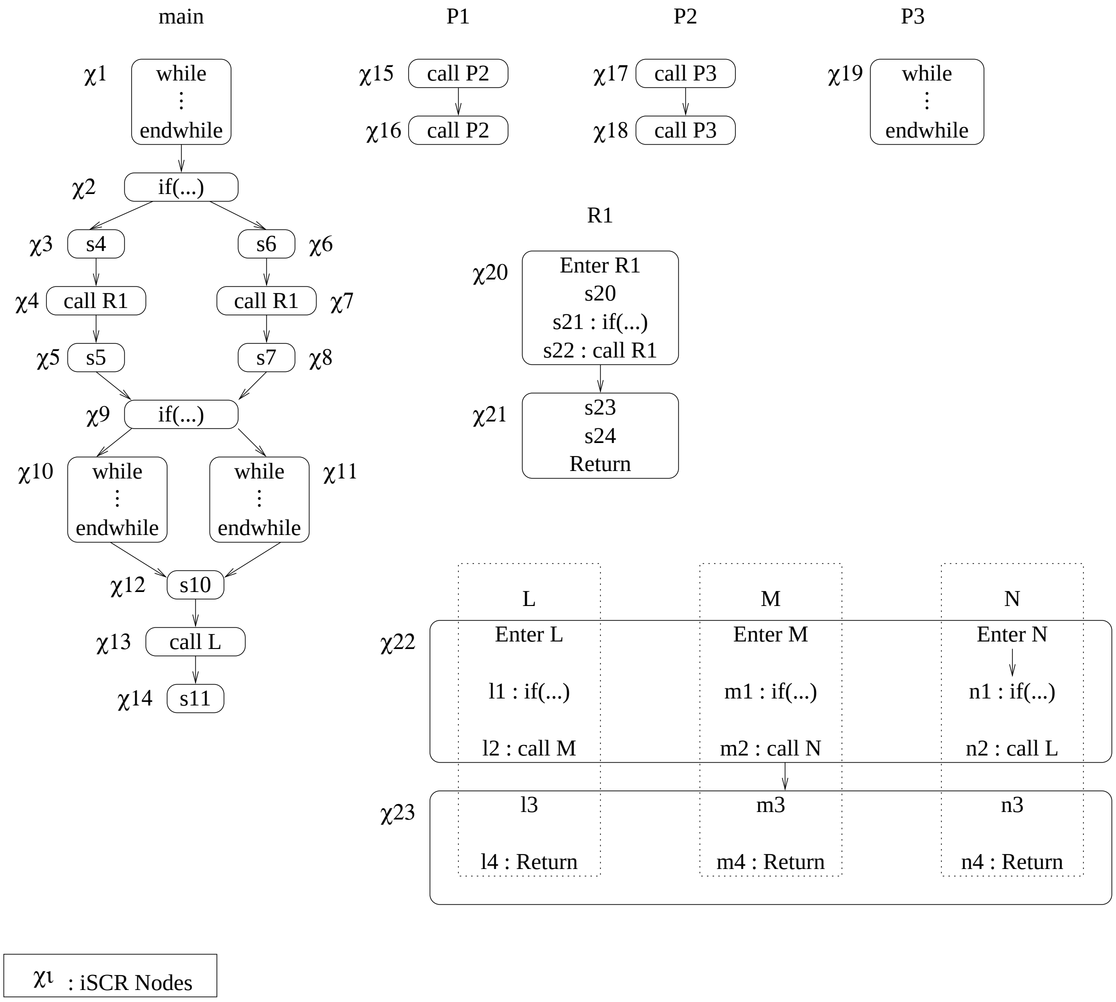


Figure 4.4: The *intraprocedural* and *recursion induced* strongly connected components (iSCRs)

5. On the inlined ISCR graph we calculate the Reach() function between every two ISCR nodes of the graph during a single traversal in topological order.

Once this set of information has been calculated we can arrive at an algorithm to calculate whether a given set of nodes form a realizable path.

4.2.1 SCRs in Recursive Procedures

First consider the case of a self recursive procedure, as for example, $R1$ in Figure 4.2. The edge from s_{22}(`call R1`) to the `Enter R1` vertex generates an iSCR containing all the statements between `Enter R1` and s_{22}. This is treated as a simple loop. Similarly, the edge from the `Return` node to s_{23} induces another iSCR and the edge $s_{21} \rightarrow s_{23}$ induces the connection between the two iSCRs. In Figure 4.4 these two iSCRs are labeled χ_{20} and χ_{21}. This is logically correct since every node in χ_{20} is reachable from every other node along some recursive path and similarly for χ_{21}. Also, every node in χ_{21} is reachable from every node in χ_{20}.

Next consider the multiway recursion in the procedures L, M and N. In the region induced by these procedures we calculate the SCRs. This region of the CFG includes the interprocedural call and return edges between these procedures *only*. For the purpose of determining the SCRs within this region we ignore the interprocedural call and return edges to and from other procedures as these edges do not belong to the given region. From the Figure 4.4 we see that there are two SCRs in the region labelled χ_{22} and χ_{23} induced by the two interprocedural loops Enter$L \rightarrow l1 \rightarrow l2 \rightarrow$ Enter$M \rightarrow m1 \rightarrow m2 \rightarrow$ Enter$N \rightarrow n1 \rightarrow n2 \rightarrow$ Enter$L$ and $l3 \rightarrow l4 \rightarrow n3 \rightarrow n4 \rightarrow m3 \rightarrow m4 \rightarrow l3$. There is an external edge from `call L` in the main procedure to χ_{22} and an external edge from χ_{23} to s_{11} in `main`.

The SCRs in a region induced by single or multiple recursion can be found by applying any standard algorithm to find the maximal loops in the region. An informal proof is given in Section 4.5.1.

4.2.2 Building the ISCR Graph

Now, in the CFG, each procedure (including `main`) consists of iSCRs connected by control flow edges. The procedures are also connected by call and return edges as in the original CFG. We now try to combine as many iSCRs (across all procedures) as possible into a single *interprocedural SCR* or ISCR. The algorithm is very simple. The algorithm walks through the collapsed CFG starting with the `main` procedure. For each multinode iSCR

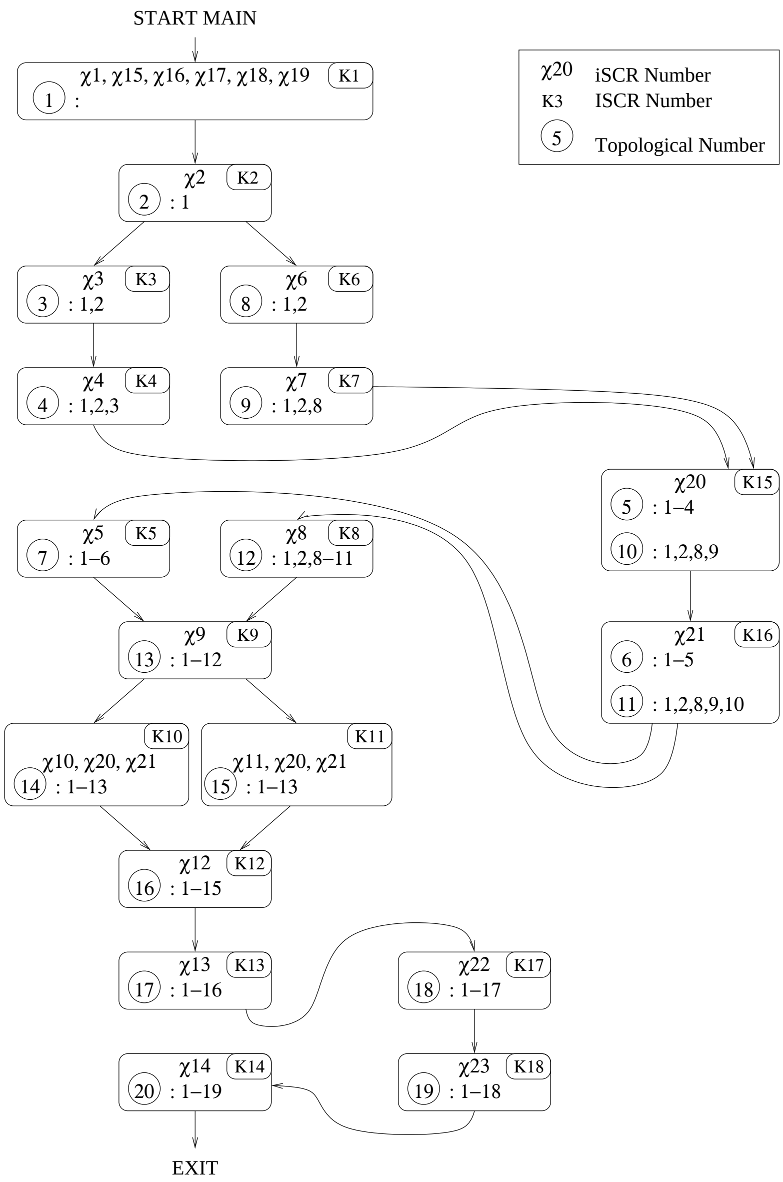


Figure 4.5: The Interprocedural strongly connected components (ISCRs)

that contains a call site ($P \to Q$), we delete the call edge and insert the list of iSCRs in Q into the parent ISCR. Then we traverse the call SCR graph from Q and insert all the iSCRs in the procedures that are called directly or transitively from Q. Since a procedure may be integrated from multiple locations, this process may have to be repeated from each location. To avoid this we use the cSCR graph to cache the list of nodes reachable transitively from a procedure. The algorithm works in two phases. In the first phase we walk through the cSCR graph, starting at the root and build the cache for each procedure call. The algorithm is given in Figure 4.6. In the second phase we start with the `main` procedure and walk through the iSCR graph and integrate as many iSCRs as possible into a single ISCR. Each iSCR has an attribute *multi* which is true if the iSCR has at least one internal edge and is false otherwise. The corresponding ISCR also inherits this attribute. A call statement may be part of a multi-node iSCR or it may be a single statement. If it is a single statement, then the corresponding ISCR generated from it is marked to be of type $CALL\_SITE$ and we introduce a dummy ISCR after it and label it to be of type $RETURN\_SITE$ with attribute *multi* set to false. If the call statement is part of a multi-node iSCR, then the called procedure is integrated into the current ISCR and the call edge will be deleted.

At the end of this phase the interprocedural SCRs are in place in the interprocedural SCR graph. The ISCR graph is shown in Figure 4.5. As we can see in the ISCR graph, several iSCRs may be combined into a single ISCR, as for example, χ_1, χ_{15}, χ_{16}, χ_{17}, χ_{18} and χ_{19} are all in the ISCR numbered K_1. Also, it is possible that a single iSCR may be tagged to several ISCRs, as for example, χ_{20} is tagged to ISCRs labelled K_{10}, K_{11} and K_{15}. In the example in Figure 4.5, the procedures $P1$, $P2$ and $P3$ have got integrated into the ISCR labelled K_1; procedures L, M and N have generated the two ISCRs K_{17} and K_{18}, but there is a call edge from K_{13} to K_{17} and a return edge from K_{18} to K_{14}. The case of $R1$ is interesting. $R1$ has been integrated into K_{10} and K_{11} so there is no call edge from K_{10} or from K_{11}. But there is a call edge from K_4 and K_7 to K_{15} and a corresponding return edge from K_{16} to K_5 and K_8, respectively.

At this stage we are assured that the resultant graph is acyclic and the ISCR nodes can be ordered in topological order after inlining all the remaining procedure calls that did not get integrated into an ISCR. We give an efficient method for inlining that does not require making actual copies of the procedures. Briefly, the algorithm walks through the graph in reverse postorder maintaining a stack of the call sites to ensure that only realizable paths are traversed. The nodes are numbered in reverse postorder. A procedure node that is visited more than once gets multiple numbers. So inlining is achieved by the simple expedient of using a single integer to number the position of the inlined procedure in the graph. The topological numbering is shown enclosed in circles in Figure 4.5. Each

```
procedure CacheTransitive (cnode)
    if cnode.mark == true then
        return
    cnode.mark = true
    cnode.cache = ∪_{P∈cnode} ∪_{iSCR∈P} iSCR
    for each successor x of cnode do
        CacheTransitive(x)
        cnode.cache = cnode.cache ∪ x.cache


procedure BuildISCRGraph (i_node)
    if i_node.mark == true then
        return
    i_node.mark = true
    ISCR = {i_node}
    ISCR.multi = i_node.multi
    if i_node.multi == true then
        for each call site x → P in i_node do
            ISCR = ISCR ∪ P.cache
            delete the edge x → P
    else /* single node */
        if i_node is a call site x → P then
            BuildISCRGraph (P)
            ISCR.type = CALL_SITE
            Introduce a dummy ISCR of type RETURN_SITE with attribute
                                                        multi = false
    for each s ∈ successor(i_node) do
        BuildISCRGraph (s)
```

Figure 4.6: Algorithm to build the ISCR graph

```
procedure GenerateTopologicalNumberSet(Inode)
    if (Inode.number > Inode.ENTRY.number) then
        return
    Inode.REACH = φ
    for each predecessor p of Inode do
        switch p.type
            case RETURN_SITE :
                PushStack(p.corres_call_site)
                GenerateTopologicalNumberSet ( p )
                Inode.REACH = Inode.REACH ∪ p ∪ p.REACH
            case CALL_SITE :
                if p == TopStack() then
                    PopStack()
                    GenerateTopologicalNumberSet ( p )
                    Inode.REACH = Inode.REACH ∪ p ∪ p.REACH
            default :
                GenerateTopologicalNumberSet ( p )
                Inode.REACH = Inode.REACH ∪ p ∪ p.REACH
        endswitch
    endfor
    Inode.number = count++
    if Inode.multi == true then
        Inode.REACH = Inode.REACH ∪ Inode.number
    Inode.top_set = Inode.top_set ∪ ⟨Inode.number, Inode.REACH⟩
    return
```

Figure 4.7: The algorithm to generate topological numbering

ISCR has one or more unique topological numbers associated with it. For example, the ISCR K_{15} has two numbers 5 and 10, corresponding to each call site whereas the ISCR K_{17} has only one number 18.

The algorithm for generating the topological order uses the following notation: Each node is of type CALL_SITE, RETURN_SITE or DEFAULT. "*Inode*" refers to an ISCR in the ISCR graph. Each ISCR has the following attributes:

- *Inode.number* holds the latest topological number received by the *Inode* and is initially set to zero;
- *Inode.REACH* is a bitvector that collects the Reach information from the *Inode*'s predecessors.
- The attribute *multi* indicates whether the ISCR has at least one internal edge. If it is true, then it implies that all nodes in the ISCR are reachable from each other.
- For every topological number there is a corresponding REACH bit vector. This pair is inserted into a list called the *Inode.top_set*. For example, in Figure 4.5 at χ_{20} there is a set of two topological numbers numbered 5 and 10 with the corresponding REACH information associated with each number.
- *Inode.ENTRY* points to the Entry ISCR of the corresponding procedure.

The algorithm starts with the `EXIT` node of the CFG and visits the nodes in reverse postorder. The variable *count* is a global variable initialized to 0. The numbering scheme allows a procedure node to be re-entered for each corresponding call site without looping endlessly or missing any call sites. The call stack ensures that procedures are entered and exited along realizable paths.

4.2.3 The Realizable Path Algorithm

The algorithm for determining the existence of a path is given in Figure 4.8. The algorithm is given in terms of iSCRs. As a node belongs to a unique iSCR, the algorithm can be easily extrapolated to a list of nodes. The procedure $GetTopologicalNumberSet(iSCR)$ does the following: It finds all the ISCRs that the iSCR belongs to and returns the set of topological numbers belonging to all these ISCRs. $GetTopologicalNumberSet(\chi_{20})$, for example, would find the ISCRs K_{10}, K_{11} and K_{15} and the corresponding set is $\{\langle 5, 5.REACH\rangle, \langle 10, 10.REACH\rangle, \langle 14, 14.REACH\rangle, \langle 15, 15.REACH\rangle\}$. In the procedure $RealizablePath(i, j, \mu)$, μ is a proper subset of all the topological numbers associated with j and represents the valid numbers associated with j at a particular position in the

procedure $ValidPath$
Input : An interprocedural SCR graph G', a set of iSCRs $\langle n_1, n_2, \cdots, n_k \rangle$
Output : **true** if the input nodes form a realizable path, **false** otherwise

```
    μ = GetTopologicalNumberSet(n_k)
    for i = k − 1 downto 1 do
        μ' = RealizablePath(n_i, n_{i+1}, μ)
        if μ' == ∅ then
            return false
        μ = μ'
    endfor
    return true
```

procedure $RealizablePath$(iSCR i, iSCR j, TopologicalNumberSet μ)

```
    newset = ∅
    ν = GetTopologicalNumberSet(i)
    for each topological number x ∈ ν do
        for each topological number y ∈ μ do
            if x.number ≤ y.number and x ∈ y.REACH then
                newset = newset ∪ x
                break
        endfor
    endfor
    return newset
```

Figure 4.8: The Realizable Path algorithm

path. The procedure locates every topological number associated with i that reaches at least one of the numbers in μ. If it returns $\emptyset$ it implies that there is no realizable path from i to j.

Example As an example we will walk through the algorithm to check if $\langle \chi_{21}, \chi_{20}, \chi_9 \rangle$ is a valid path. The nodes are processed in reverse order. First μ is initialized to the topological number set associated with χ_9, which is (13). The numbers associated with χ_{20} are $(5, 10, 14, 15)$. Of these, we need to check only 5 and 10 since 14 and 15 are larger than 13 and hence cannot precede 13. We have both $5 \in 13.REACH$ and $10 \in 13.REACH$. So now *newset* has $(5, 10)$, which is returned by *RealizablePath*. Next we check χ_{21} whose numbers are $(6, 11, 14, 15)$. Of these none are smaller than 5 and only 6 is smaller than 10 but $6 \notin 10.REACH$ and hence this is not a valid path.

The correctness of the algorithm follows from the correctness of the RealizablePath algorithm. The correctness of the RealizablePath algorithm can be stated as

Theorem 2 *For any two iSCR nodes i and j in the ISCR graph, $RealizablePath(i, j, \mu)$ returns all and only the topological numbers of i that have a path to some topological number of j in μ.*

Then the correctness of the ValidPath algorithm may be stated as

Theorem 3 *Given a set of iSCR nodes and the corresponding set of topological numbers $\langle \langle q_1, \mu_1 \rangle, \langle q_2, \mu_2 \rangle, \ldots, \langle q_n, \mu_n \rangle \rangle$, such that $\mu_n = GetTopologicalNumberSet(q_n)$ and $\mu_i = RealizablePath(q_i, q_{i+1}, \mu_{i+1})$, for $1 \leq i < n$, then there is a realizable path through $\langle q_1, q_2, \ldots, q_n \rangle$ if and only if $\mu_i \neq \emptyset$ for $1 \leq i \leq n$.*

Corollary 3 Given a set of iSCR nodes and the corresponding set of topological numbers $\langle \langle q_1, \mu_1 \rangle, \langle q_2, \mu_2 \rangle, \ldots, \langle q_n, \mu_n \rangle \rangle$, such that $\mu_n = GetTopologicalNumberSet(q_n)$ and $\mu_i = RealizablePath(q_i, q_{i+1}, \mu_{i+1})$, for $1 \leq i < n$, then $\langle q_0, q_1, q_2, \ldots, q_n \rangle$ is a realizable path if and only if $RealizablePath(q_0, q_1, \mu_1) \neq \emptyset$.

Thus, by Corollary 3, it is clear that to add a node to an existing realizable path, it is sufficient to check for a path from the new node to some topological number of the last node added to the path. It is not necessary to run ValidPath on the entire set of nodes in the path or even to keep track of all the previous nodes in the path. Thus the RealizablePath algorithm can be used to *incrementally* determine whether a set of nodes form a realizable path.

We give a proof of correctness of the RealizablePath algorithm in Section 4.5.2.

4.3 Complexity

Since we are using a form of procedure inlining, the space complexity can in the worst case grow exponentially. If there is a long call chain, of length y, where each procedure in the chain is called from x sites then a procedure at the bottom of the call chain may get inlined x^y times. Although we use only one integer to represent the inlined procedure the bit vectors may grow very large.

The steps in building the interprocedural SCR graph involve finding SCRs in each procedure and finding SCRs in the call graph. There are standard near-linear algorithms for doing this. Integrating the intraprocedural SCR graphs with the call SCR graph to generate the interprocedural SCR graph requires a single walk through the program and also takes linear time. The algorithm for inlining the procedures and calculating the REACH information has potentially exponential performance if there are very long chains of procedures that need to be integrated.

The time complexity of the *RealizablePath* algorithm depends on the number of bit vector operations performed in each step of the outermost loop. If the number of bits is exponential we may need to perform exponential bit vector operations. In addition, procedure inlining gives rise to exponential number of topological numbers stored in the bitvector at a node.

In general, this algorithm does not scale very well, not due to the time complexity but due to the space complexity. Both the build time and the running time of the algorithm are very good, but large programs tend to run out of memory when building the REACH bitvectors during the topological numbering in procedure *GenerateTopologicalNumberSet*. So, in practice, instead of creating bitvectors for each topological number, we only generate the numbers. In the *RealizablePath* algorithm instead of doing a single bitvector lookup in the step $x \in y.REACH$, we walk back along the ISCR graph from y until we reach x or a topological number that is smaller than x. If we reach x, then there is a valid path from x to y else there is not. As a result the memory requirement is very reasonable. It might appear that the time required might be very high but in practice the algorithm is very fast. We have tested it on several Java benchmark programs and the results are given in Section 9.2. The *Reach* algorithm is described in Section 4.3.1.

4.3.1 The Reach Algorithm

The algorithm, *Reach*, to determine whether there is a path from a topological number x to another number y is given in Figure 4.9. As the algorithm uses a graph of topological numbers whose size is exponential on the size of the original ISCR graph, it might appear that the complexity of the Reach algorithm is also exponential. However, with a little

```
procedure Reach ( Number from, Number to )
    if from == to then
        return true
    if from < to then
        return false
    if to is a return site then
        callnumber = Number of the corresponding call site (using dummy edge)
        if callnumber ≥ from then
            /* Skip the called procedure */
            return Reach ( from, callnumber )
        else
            /* Descend into the called procedure */
            prednumber = predecessor of to
            return Reach ( from, prednumber )
    else
        for each predecessor, prednumber of to do
            if Reach ( from, prednumber ) == true then
                return true
    return false
```

Figure 4.9: The Reach algorithm

thought the algorithm can be made linear in the size of the ISCR graph which is in the worst case $O(N)$, where N is the number of nodes in the graph.

Each edge of the ISCR graph maps to an edge from a unique topological number to another. So we delete the edges in the ISCR graph and create the edges between the corresponding topological numbers. For example in Figure 4.5, the ISCR numbered K_{15} has two topological numbers 5 and 10. We create an edge from the topological number 4 in K_4 to the topological number 5 in K_{15} and an edge from the topological number 9 in K_7 to the topological number 10 in K_{15} and delete the corresponding edges from the ISCR graph. Although K_{15} has two predecessors, the numbers 5 and 10 have only one predecessor. Thus we do not need to maintain the calling context explicitly.

A topological number may have multiple edges reaching it if it has multiple predecessors. The topological number 16 in K_{12} has two edges reaching it from the number 14 in K_{10} and the number 15 in K_{11}.

In addition, we keep a dummy edge from the number associated with a call site to the number associated with the corresponding return site. For example, we create a dummy edge from 9 to 12 and from 4 to 7 in Figure 4.5.

The algorithm starts from the larger number and walks back along the ISCR graph by following the edges associated with the topological numbers. As explained, we do not need to maintain any calling context. Let the source number be x and the destination number be y. When the algorithm reaches a return site, it uses the dummy edge to check if the call site number is smaller than x. If the call site number is smaller than x then it descends into the called procedure using the ISCR edge, else it can skip the called procedure and continue traversing from the call site. Thus the traversal enters a procedure only if x is in the called procedure or in a procedure called (transitively) from the called procedure, otherwise it keeps going up the ISCR graph. Thus the maximum number of ISCRs visited by the algorithm is at most the number of ISCRs in the ISCR graph which is $O(N)$.

4.4 Conclusion

In this chapter we have described a practical algorithm for determining whether a given set of nodes form a path in an interprocedural program. The algorithm is based on the interval-based approach of data flow analysis and although it has worst case exponential complexity, it was found to be practical when applied to standard Java benchmark programs. The implementation statistics are given in Chapter 9. In Chapter 5 we show how this algorithm is useful in the context of slicing of concurrent programs. We conjecture that it may have many more applications.

To the best of our knowledge, no one else has worked on a similar problem.

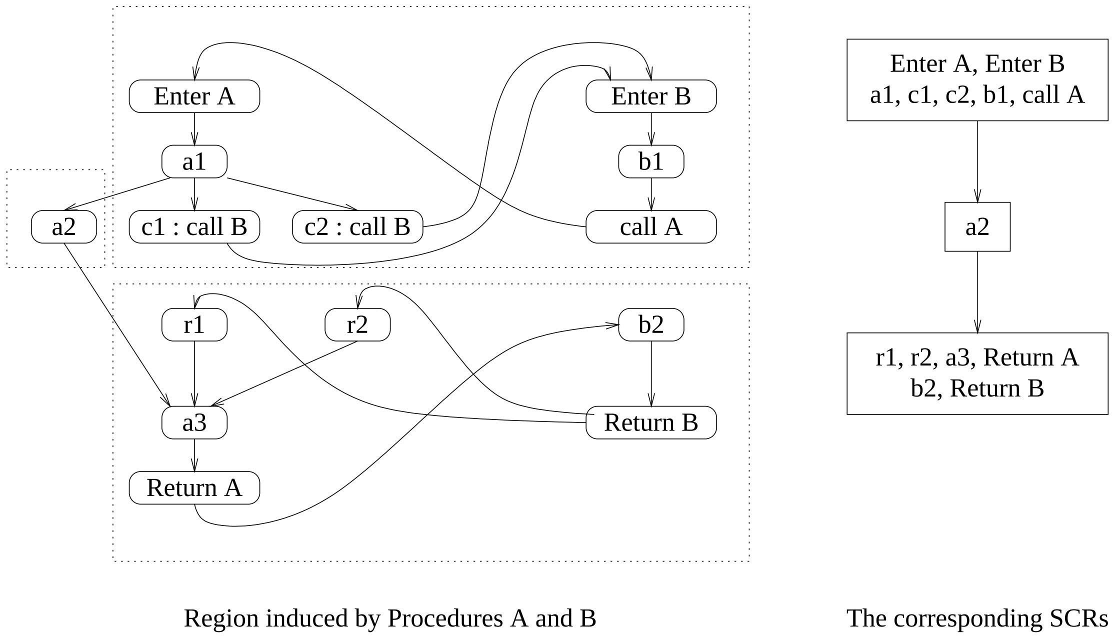


Figure 4.10: Region and SCRs induced by recursion

4.5 Appendix

4.5.1 SCRs in Recursive Procedures

Every recursive call essentially breaks a procedure into three regions which we term the *Recurse*, *Retreat* and *Base* regions. The *Recurse* region consists of all nodes from procedure entry to the call statement. The *Retreat* region includes all nodes from the corresponding return site to the procedure exit. The *Base* region includes all the remaining nodes/edges and represents the base case of recursion where there is a path from Entry to Exit without going through a recursive call. In single or multi-way recursion there must be at least one *Base* region otherwise the program will loop infinitely.

Intuitively, we see that every node in the *Recurse* region can be reached from every other node in the region by simply building up a sufficiently large stack and hence is an SCR. The same is true of the *Retreat* region. The *Base* region corresponds to an intraprocedural region and SCRs can be found in this region using standard algorithms. A procedure may have several recursive calls within it. If the *Recurse* or *Retreat* SCR induced by one recursive call intersects with the *Recurse* or *Retreat* SCR induced by another recursive call then the two can be merged into a single SCR.

We show that a standard intraprocedural algorithm for finding loops can be applied to find the interprocedural SCRs in recursive procedures. Let R be the region induced by the recursive procedures. The proof is in two steps:

1. We show that for every *Recurse* and *Retreat* region within R there is a loop in R.

2. The outermost loops in R representing maximal SCRs correspond to *valid* paths.

The first part of the proof is very simple. Consider a recursive call sequence $P_1 \rightarrow P_2 \rightarrow \cdots \rightarrow P_n \rightarrow P_1$. In each procedure P_i there is a path from EntryP_i to the call statement callP_{i+1} and a path from the return site r_i to the return node ReturnP_i. Hence clearly there is a cycle in the region R, $\langle$Entry$P_1 \rightarrow$ call$P_2 \rightarrow$ Entry$P_2 \rightarrow$ call$P_3 \cdots$ Entry$P_n \rightarrow$ call$P_1 \rightarrow$ Entry$P_1\rangle$. This corresponds to the *Recurse* SCR of R. Similarly, there is a loop $\langle r_1 \rightarrow$ Return$P_1 \rightarrow r_n \rightarrow$ Return$P_n \rightarrow \cdots \rightarrow$ Return$P_2 \rightarrow r_1\rangle$, which forms the *Retreat* SCR.

For the second part of the proof we need to show that the loops generated correspond to realizable paths. Without loss of generality, we assume that there are two procedures A and B, where A calls B and B calls A. If there is exactly one call statement in each of the procedures then all paths are realizable paths and all loops correspond to realizable paths. In order to get unrealizable paths there must be at least two calls to the same procedure. Let us assume that in A there are two calls to B, at c_1 and c_2. Then there are three possibilities, either c_1 and c_2 are on disjoint paths, or there is an intraprocedural path from c_1 to c_2 or there is an intraprocedural path from c_1 to c_2 and an intraprocedural path from c_2 to c_1 (*i.e.*, c_1 and c_2 are in the same loop). We will show that in each case either there is **no** interprocedural loop that includes unrealizable paths, or every loop with unrealizable paths is subsumed by an enclosing loop with realizable edges.

- Let c_1 and c_2 be on disjoint paths as shown in Figure 4.10. A loop with unrealizable edges must have a path where the call at c_1 returns to r_2 or a call from c_2 returns to r_1. However, by construction, we see that there is no loop that includes both a call vertex and a return vertex and hence all loops correspond to realizable paths only.

- Let there be a path from c_1 to c_2 as shown in Figure 4.11. The *Recurse* and *Retreat* regions induced by c_1 in A are $\langle$Enter A, $a1$, $c_1\rangle$ and $\langle r1$, $c2$, $r2$, $a2$, Return$A\rangle$ respectively. The interprocedural *Recurse* and *Retreat* SCRs induced by c_1 are $\langle$Enter A, $a1$, c_1, Enter$B$, $b1$, call$A\rangle$ and $\langle r1$, $c2$, Enter$B$, $b1$, call$A$, Enter$A$, $a1$, $a2$, Return$A$, $b2$, Return$B$, $r2$, $a2$, Return$A\rangle$ respectively. These are formed purely from realizable paths. There is clearly a large intersection between the two SCRs and hence they can be merged to form a single SCR that contains every node in the two procedures. Clearly, there is a realizable path from every node to every other

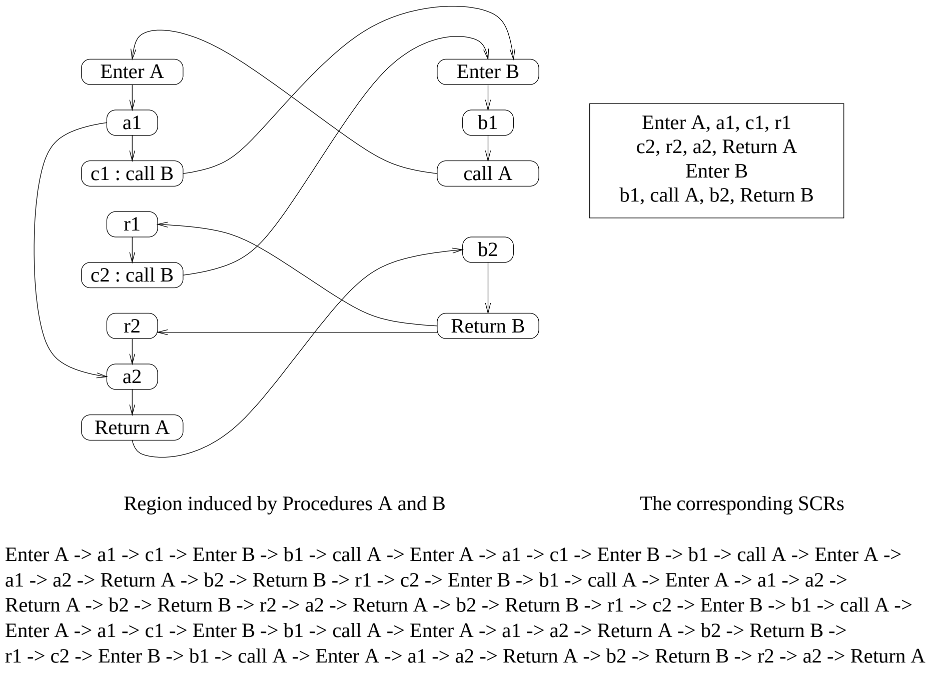


Figure 4.11: Region and SCRs induced by recursion

node (we show a realizable recursive path depicting this in the figure). Hence any nested loops formed from unrealizable paths will be subsumed by the larger SCR.

- The case where c_1 and c_2 are in a loop is identical to the previous case as a single edge from r_1 to c_2 is sufficient to generate an SCR with all the nodes.

All recursive structures are a combination of the above cases. As we are not interested in nested loops and are only looking for the outermost SCR, we do not mind if the algorithm generates intermediate loops that have unrealizable paths as they will eventually be subsumed by a larger realizable loop. Hence we can use any standard algorithm to generate the SCRs in the graph.

4.5.2 Proof of Correctness of Realizable Path Algorithm

By construction, every iSCR node i is part of at least one ISCR node in the ISCR graph. Each ISCR node has at least one topological number associated with it. This follows from the assumption that there is a path from $ENTRY$ to every node in the CFG and a path from every node to $EXIT$ in the CFG. The proof uses the following fact: for an iSCR node i, $GetTopologicalNumberSet(i)$ returns *all* the topological numbers of all the ISCR nodes in the ISCR graph to which i belongs.

By construction, every intraprocedural and interprocedural loop is collapsed into a single node. Therefore there are no loops in the ISCR graph. Also, by construction, the topological numbering inlines all the procedures in the ISCR graph. Hence we have the following two lemmas.

Lemma 1 *The ISCR graph has no loops.*

Lemma 2 *Topological numbering reduces the program to a single procedure with no loops.*

Lemma 3 *For any two topological number x_i and x_j in the ISCR graph $x_i \in x_j.REACH$ if and only if there is a path from x_i to x_j. Also, if there is a path from x_i to x_j, then $x_i \in x_j.REACH$.*

Proof This follows from Lemma 1 and the GenerateTopologicalNumberSet algorithm that traverses every path in the inlined ISCR graph and sets a bit at x_i for every topological number encountered from $START$ to x_i along every path from $START$ to x_i.

Proof of Theorem 2 The proof follows from Lemma 3 and the $RealizablePath(i, j, \mu)$ algorithm, which returns every topological number of i that belongs to $j.REACH$ for some $j \in \mu$ and rejects any topological number of i that does not belong to any $j.REACH$ for some $j \in \mu$.

Lemma 4 *Given any three topological numbers q_i, q_j and q_k, a path from q_i to q_j and a path from q_j to q_k implies there is a realizable path through $\langle q_i, q_j, q_k\rangle$.*

Proof By Lemma 2 and by property of intraprocedural paths.

Proof of Theorem 3 By Theorem 2, $RealizablePath(i, j, \mu)$ returns a non-empty set if and only if there is a path from some topological number of i to some topological number in μ. Hence there is a path from q_i to q_{i+1} if and only if $\mu_i = RealizablePath(q_i, q_{i+1}, \mu_{i+1})$ and $\mu_i \neq \emptyset$. If any μ_i is the empty set then there is no path from that q_i to q_{i+1} and there cannot be a realizable path. Conversely, if $\mu_i \neq \emptyset$ for $i \leq 1 \leq n$, then there is a path from every q_i to q_{i+1} and by Lemma 4 there is a realizable path through $\langle q_1, q_2, \ldots, q_n\rangle$.

Hence the theorem.

Proof of Corollary 3 By Theorem 3, there is a realizable path through $\langle q_1, q_2, \ldots, q_n\rangle$. By theorem 2 there is a path from q_0 to q_1 and by Lemma 4 there is a realizable path through $\langle q_0, q_1, q_2, \ldots, q_n\rangle$.

Thus, to add a new node to the realizable path, it is sufficient to apply RealizablePath to the last node added to the path and its corresponding set of topological numbers.

Chapter 5

SLICING CONCURRENT INTERPROCEDURAL PROGRAMS

In Chapter 3 we had given a solution to slicing of concurrent programs. In this chapter, we extend the algorithm to concurrent programs with procedures. We give a solution for the basic concurrency model with `cobegin-coend` parallelism and no explicit synchronization. First we give a context-insensitive solution which is correct and efficient though imprecise and then we give a context-sensitive solution which is both correct and precise.

5.1 Introduction

The interprocedural concurrent program is represented by an *interprocedural threaded control flow graph* (ITCFG) as shown in Figure 5.1. A *threaded system dependence graph* (TSDG) is an extension of the system dependence graph of sequential programs, with extra edges representing interference dependence. Briefly, each procedure is represented by a PDG, with a unique `ENTRY` and `EXIT` node. At every procedure there is a *FormalIn* node for each global variable and parameter of the procedure that may be modified or referenced by the procedure and a *FormalOut* node for each global variable and parameter that may be modified by the procedure. Each call statement is represented by a *call* node. At each call site, there is an *ActualIn* node for each FormalIn node and an *ActualOut* node for each FormalOut node. There is a *call* edge from the call node to the *Entry* node of the called procedure. *ParameterIn* edges are drawn from corresponding *ActualIn* nodes to FormalIn nodes, and *ParameterOut* edges from FormalOut nodes to *ActualOut* nodes. *Summary*

```
x=10
y=20
cobegin {
   a=5
   b=6
   call f2(a)
   call f1(a,b)
   print a
   call f2(b)
}{
   y=x+1
}

proc f1(c,d)
   x=c+1
   call f2(d)
   c=y+1

proc f2(e)
   x=e+1
```

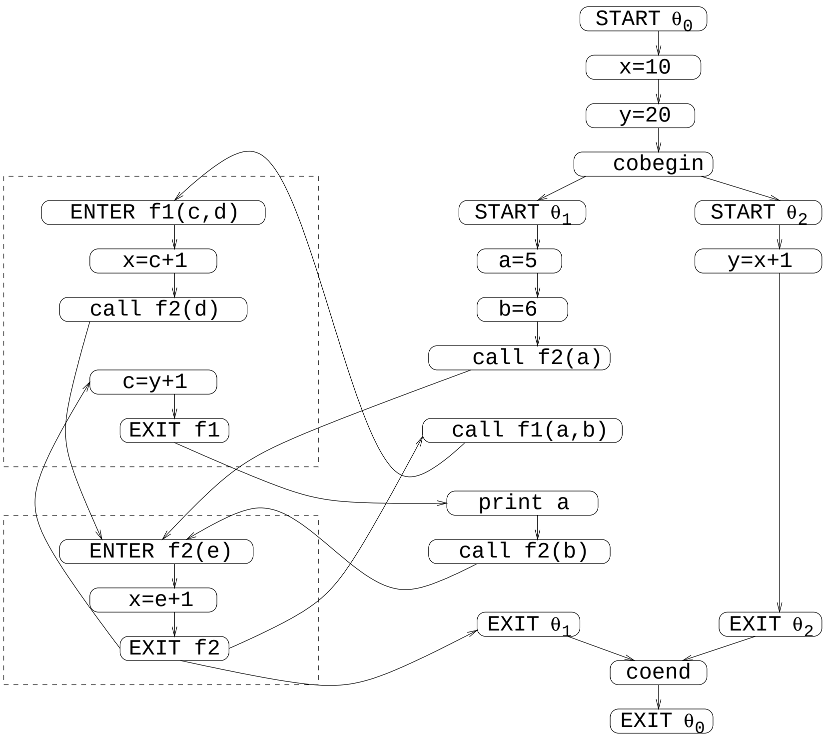


Figure 5.1: A threaded interprocedural control flow graph (ITCFG)

edges are drawn from *ActualIn* nodes to *ActualOut* nodes to represent transitive flow dependencies. The slicing criterion is defined as a node in the TSDG. The set of edges, E in the TSDG are the data dependence, control dependence, interference dependence, call, summary, parameter-in and parameter-out edges which are denoted by E_{dd}, E_{cd}, E_{id}, E_c, E_s, E_{pi} and E_{po} respectively. All edges are constructed using standard techniques [49, 58]. The TSDG corresponding to the TCFG of Figure 5.1 is given in Figure 5.3.

In sequential programs, the intraprocedural slicing algorithm produces imprecise slices when applied to interprocedural program. This is because not all paths in an interprocedural CFG are realizable paths. The two-phase slicing algorithm [49] solves the problem of context-sensitivity for sequential programs.

Similarly, in concurrent programs also, the intraprocedural slicing algorithm generates imprecise slices when applied to concurrent interprocedural programs. Further, a simple extension of the two-phase algorithm to concurrent programs produces not only imprecise but incorrect slices.

In this chapter, we identify the various issues in generating context-sensitive slices for concurrent interprocedural programs and then give solutions. We start with a context-insensitive solution which is correct and efficient though imprecise and then propose a context-sensitive algorithm which is both correct and precise.

5.2 Issues

There are two main issues here

1. **Correctness:** A slice is correct if it includes, at least, all the nodes that affect the criterion. A slice is *incorrect* if it excludes some nodes that should be in the slice.

2. **Precision:** Given two correct slices, one slice is more precise than the other if it contains less nodes.

In the extreme, the complete program is always a correct slice, albeit possibly imprecise. Ideally, one would like to compute the correct slice that is the most precise. But this is, in general, not computable. Hence, we try to compute a slice that is correct and as precise as possible. A simple two-phase algorithm with extensions for interference dependence edges as the one proposed by Zhao [93] is both incorrect and imprecise.

5.2.1 Correctness Issues

In the example in Figure 5.2 let us apply the two-phase algorithm to the node `print x`. In Phase 1, `print x` is data dependent on `x=x_out` which is parameter-out dependent on

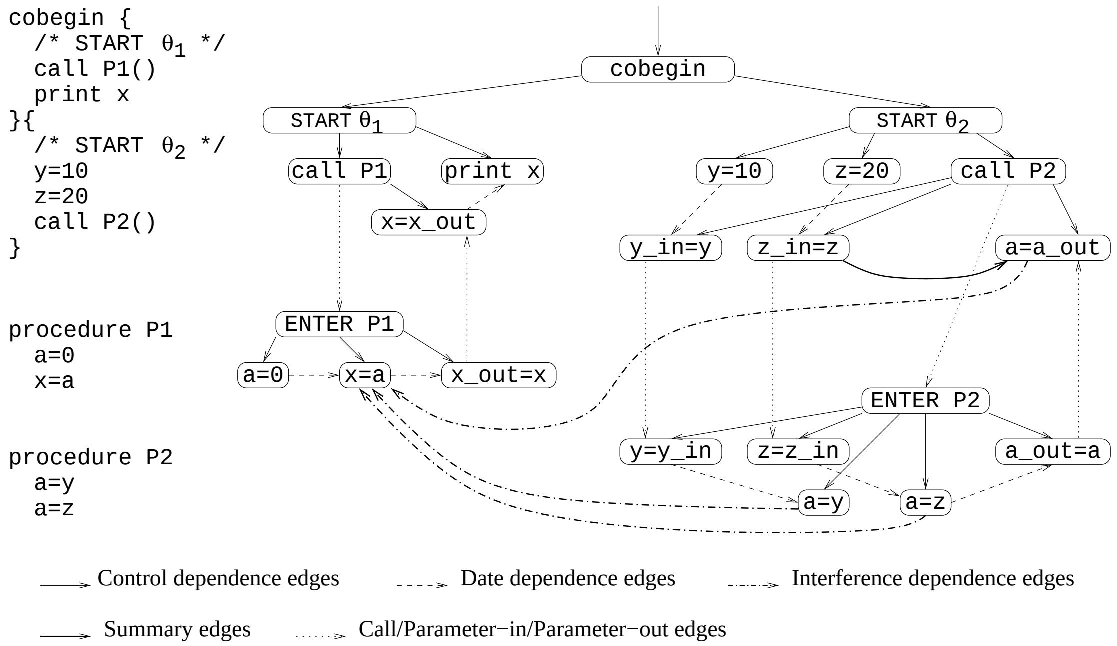


Figure 5.2: Example showing incorrectness of the two-phase algorithm

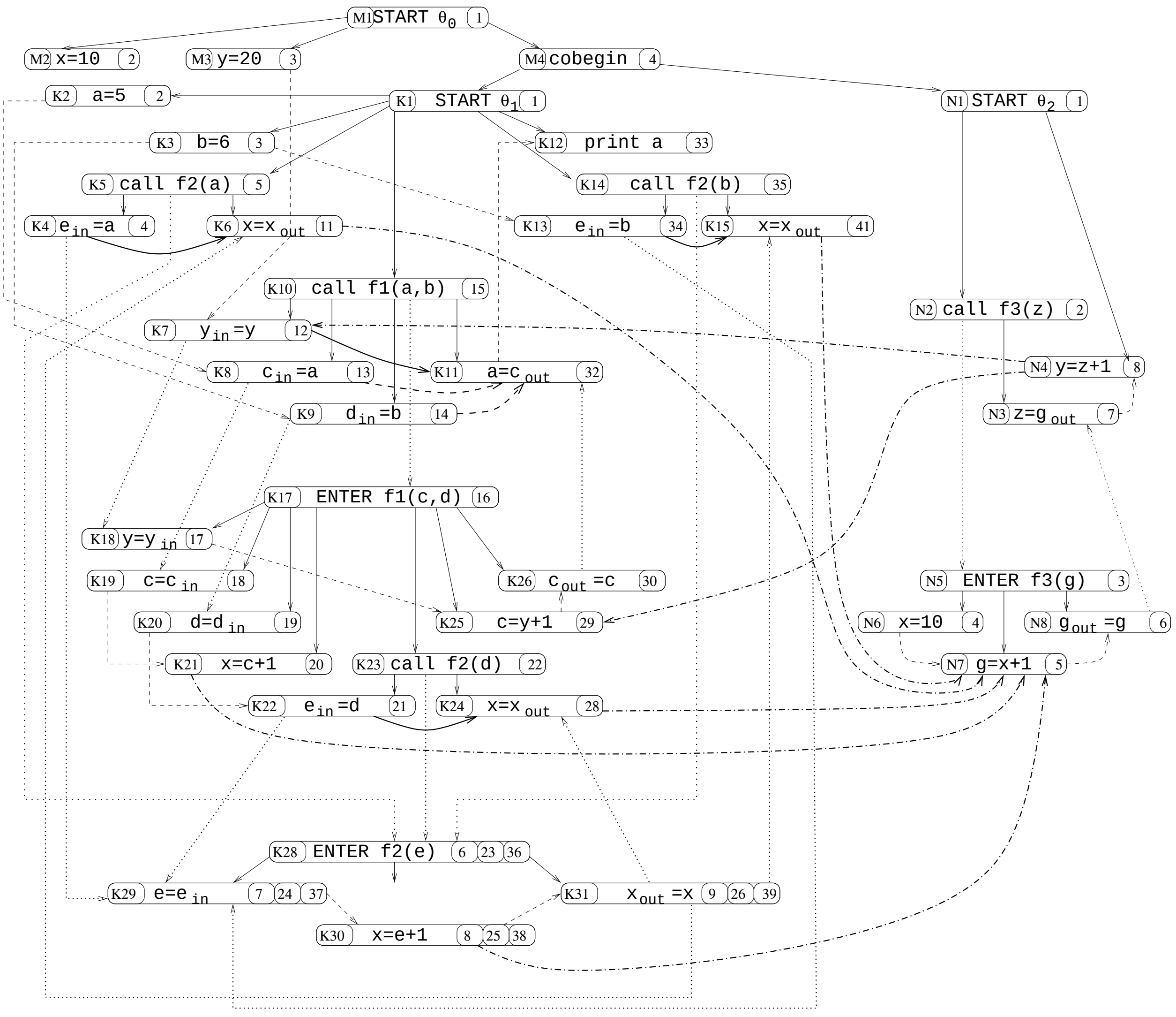
`x_out=x` in procedure `P1`.

In Phase 2, `x_out=x` is data dependent on `x=a`. `x=a` is data dependent on `a=0`, interference dependent on `a=y` and `a=z` in procedure `P2` in thread θ_2 and interference dependent on `a=a_out` in θ_2. Next, `a=z` is data dependent on `z=z_in` which is parameter-in dependent on `z_in=z`. Since the algorithm is now in Phase 2, `z_in=z` will not be added to the slice. However, `a=a_out` is summary dependent on `z_in=z` and hence `z_in=z` will eventually get added to the slice along with its transitive dependencies. Now consider the dependencies arising from `a=y`. `a=y` is data dependent on `y=y_in` which is parameter-in dependent on `y_in=y`. Since the algorithm is in Phase 2, `y_in=y` will not be added to the slice. As a result `y_in=y` and nodes that it is transitively dependent on (`y=10`) will never get added to the slice.

Hence the two-phase algorithm gives an incorrect slice.

Inter-thread Summary Edges Another problem with applying the two-phase algorithm directly is that the summary edges do not adequately represent transitive dependencies in the presence of threads since there may be transitive dependencies that cross thread boundaries.

Consider the slice of K_{12} (`print a`) in Figure 5.3. In Phase 1, we find that K_{12} is




```
x=10                proc f1(c,d)
y=20                   x=c+1
cobegin {              call f2(d)
    a=5                c=y+1
    b=6
    call f2(a)
    call f1(a,b)    proc f2(e)
    print a            x=e+1
    call f2(b)
}{
    call f3(z)      proc f3(g)
    y=z+1             x=10
}                     g=x+1
```

Mι : ISCR number in θ_0

Kι : ISCR number in θ_1

Nι : ISCR number in θ_2

ι : topological number

- control dependence
- data dependence
- summary
- interference dependence
- call/parameter−in/parameter−out

- inter−thread summary edges

Figure 5.3: Illustrating inter-thread summary edges

data dependent on K_{11} which is summary dependent on K_7 and parameter-out dependent on K_{26}. In Phase 2, we trace the following dependencies backwards: $K_{26} \leftarrow K_{25} \leftarrow N_4 \leftarrow N_3 \leftarrow N_8 \leftarrow N_7 \leftarrow K_{21} \leftarrow K_{19}$. K_{19} is parameter-in dependent on K_8 but since the algorithm is in Phase 2, K_8 will not get added to the slice and nor will any of the nodes that it depends on. Similarly, K_9 and its dependencies will also not get added to the slice.

Here the problem is that some transitive dependencies are missing. Procedure $f1$ has the following sequence of assignments: $x = c + 1$; $c = y + 1$. If we were to determine summary edges for thread θ_1 independently of other threads, we would not find any transitive dependence between the $FormalIn$ node $c = c_{in}$ and the $FormalOut$ node $c_{out} = c$. However, if we consider the inter-thread sequence of execution $c = c_{in}$; $x = c+1$; $g = x + 1$; $g_{out} = g$; $z = g_{out}$; $y = z + 1$; $c = y + 1$; $c_{out} = c$; then we see that there is a transitive dependence from the $FormalIn$ node $c = c_{in}$ to the $FormalOut$ node $c_{out} = c$. That is, there should be a summary edge from $c_{in} = a$ to $a = c_{out}$ at the call to $f1$ in θ_1. Similarly, there should also be a summary edge and from $d_{in} = b$ to $a = c_{out}$. These have been marked as *inter-thread summary edges* in Figure 5.3. These transitive dependencies cannot be detected by the standard [49] intra-thread summary edge algorithm.

5.2.2 Precision Issues

Even in the absence of inter-thread summary edges, Zhao's algorithm remains imprecise and context-insensitive. In Figure 5.4 we show a program and the corresponding TSDG is in Figure 5.5. In Figure 5.5, m_6 is parameter-out dependent on m_{21} in procedure `f1`. Only the call at m_5 should be added to the slice, but a simple two-phase algorithm picks up the call to `f1` at m_{11} also ($m_{21} \leftarrow m_{19} \leftarrow n_3 \leftarrow m_{13} \leftarrow m_{11}$).

It might appear that a naive marriage of Krinke's algorithm and Zhao's algorithm would generate precise context-sensitive slices. However, this is not the case because unlike intraprocedural programs, paths in interprocedural programs are not transitive.

Intransitivity of Interprocedural Paths Determining whether there is a realizable path between any *two* nodes in the interprocedural control flow graph of a *sequential* program is possible using standard techniques of data flow analysis. The relation $Reach(i, j)$ is **true** if there is a realizable path from i to j and **false** otherwise. For example, in Figure 5.4, $Reach(m_8, m_{20})$ and $Reach(m_{20}, m_{19})$ are true but $Reach(m_{11}, m_8)$ is false. In intraprocedural programs, the composition of $Reach$ is a realizable path. That is, $Reach(a, b) \circ Reach(b, c)$ implies a realizable path $\langle a, b, c \rangle$. In interprocedural programs the composition of $Reach$ may not be a realizable path. For example, $Reach(m_8, m_{20}) \circ Reach(m_{20}, m_{19})$ does not imply that there is a realizable path through

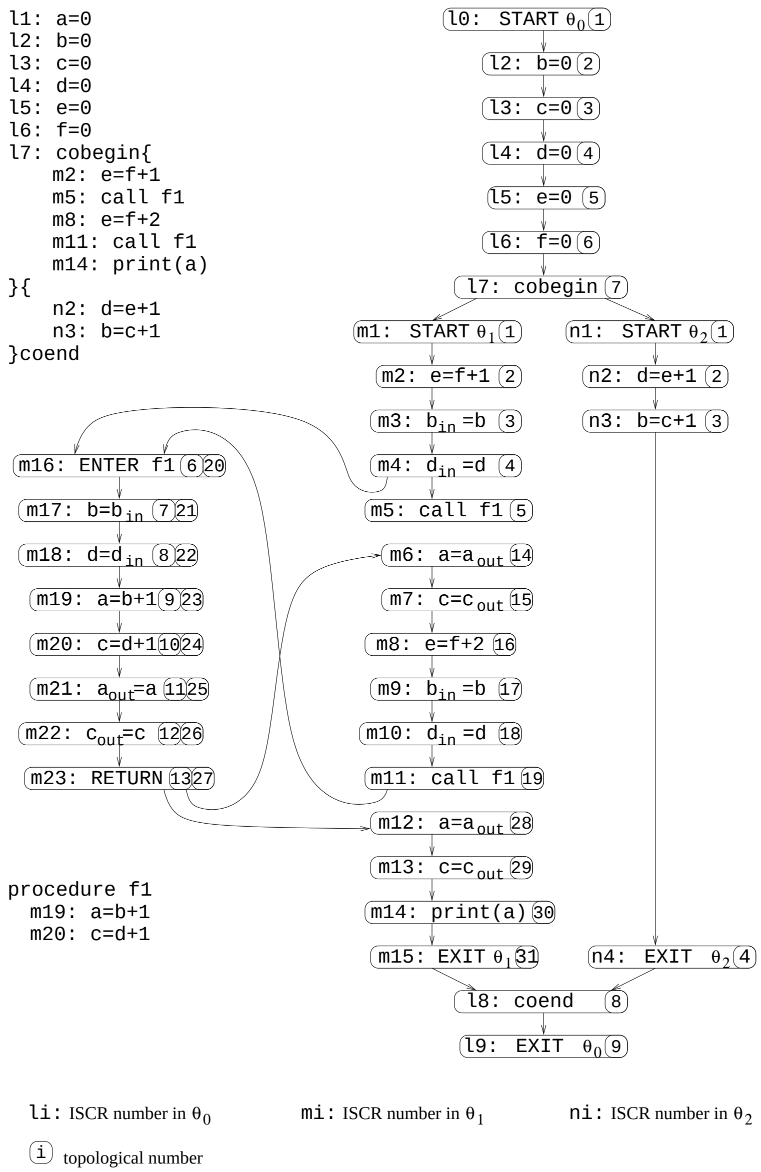


Figure 5.4: A threaded interprocedural CFG with topological numbers

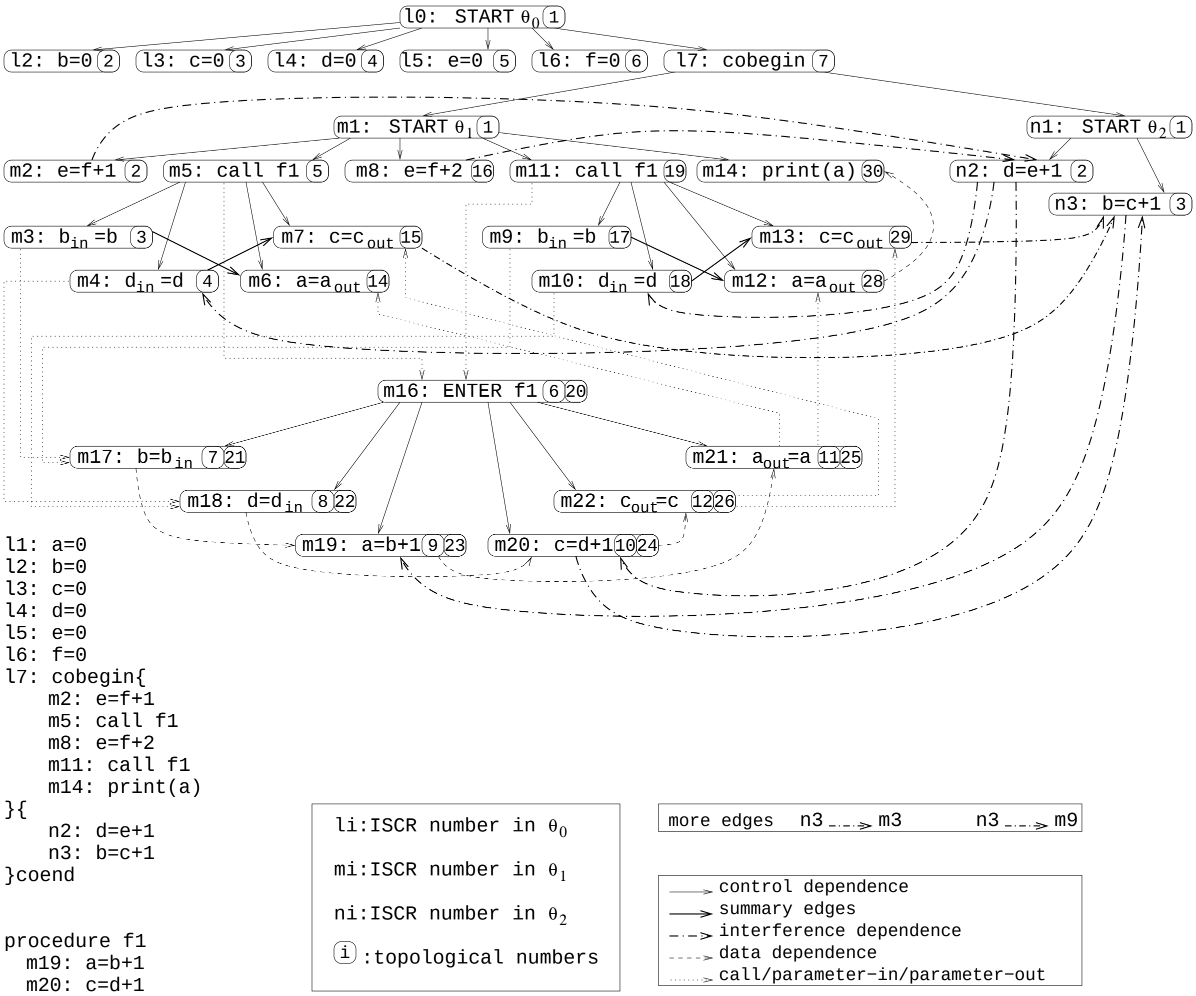


Figure 5.5: An example depicting intransitivity of paths

$\langle m_8, m_{20}, m_{19} \rangle$.

As a consequence, a simple extension of Krinke's algorithm to concurrent interprocedural programs generates context-insensitive slices. To illustrate this issue, Consider a simple extension of the two-phase algorithm where we keep track of the last node visited in each thread. The slice of m_{14} would trace the following dependencies: In Phase 1, $m_{14} \leftarrow m_{12} \leftarrow m_{21}$. In Phase 2, $m_{21} \leftarrow m_{19} \leftarrow n_3$. Since n_3 is in θ_2 the algorithm remembers that the last node visited in θ_1 was m_{19}. Next, $n_3 \leftarrow m_{20}$. When the algorithm re-enters θ_1 it finds that there is a realizable path from m_{20} to m_{19} (Figure 5.4) and so m_{20} is added to the slice.

Note that the path from m_{20} to m_{19} is an interprocedural path that corresponds to an execution of m_{20} from the call at m_5 followed by an execution of m_{19} from the call at m_{11}. However, the path information has no way of keeping track of the calling contexts associated with each node.

Further, $m_{20} \leftarrow n_2$. On re-entering θ_2, the algorithm checks that there is a path from n_2 to n_3. Now, n_2 is interference dependent on m_2 and m_8. Both have a path to m_{20} and so both will be added to the slice. But adding m_8 violates the realizable path condition. To understand why, observe that the path traced by the algorithm within θ_1 is as follows : $m_8 \rightarrow m_{20} \rightarrow m_{19} \rightarrow m_{21} \rightarrow m_{12} \rightarrow m_{14}$ and the path within θ_2 is $n_2 \rightarrow n_3$. The subsequence of nodes in θ_2 is $\langle n_2, n_3 \rangle$ which form a valid path in θ_2. The subsequence of nodes in θ_1 is $\langle m_8, m_{20}, m_{19}, m_{21}, m_{12}, m_{14} \rangle$. In θ_1, Although there is a *realizable* path between every pair of nodes in the sequence, yet the *composition* of the realizable paths is not a realizable path in θ_1.

5.2.3 Our Solution

We start with a simple model of concurrency where there are no nested threads and no threads nested within a loop. We first solve the problem of correctness. We observe that a node that may be reached via an interference dependence edge needs to be reprocessed in Phase 1, even if it has been reached in Phase 2 or previously been processed in Phase 2. On this premise we device a slicing algorithm which is correct.

This solution, however, produces context-insensitive slices. Then we extend the context-insensitive algorithm to a context-sensitive solution by ensuring that the nodes sliced in each thread lie on a realizable path in that thread (Section 5.4). This is accomplished by applying the Realizable Path algorithm described in Section 4.2.3. In Sections 5.5 and 5.6 we extend the algorithm to nested threads and threads nested within loops.

5.3 A Context-Insensitive Slicing Algorithm

In this section we give a *correct* slicing algorithm which is not very precise but is correct and fast. First we make a general observation about two-phase slicing of sequential programs (or a single thread of a concurrent program). In a sequential program

- For a node that is sliced in Phase 1, every node that it is dependent on gets added to the slice.
- For a node that is sliced in Phase 2, only the relevant *subset* of the calling contexts are added to the slice.

We had shown that the two-phase algorithm gives incorrect results since the summary edges do not capture inter-thread transitive relations. Rather than design an algorithm to compute inter-thread transitive dependencies that would allow us to use the standard two-phase algorithm, we design a slicing algorithm that uses the intra-thread summary edges to maintain *some* context-sensitivity within a thread and compute the inter-thread dependencies on the fly.

The intuition behind the slicing algorithm for concurrent interprocedural programs is simple

- A node that is reached via an interference dependence edge needs to be sliced in Phase 1, since we need to find all the nodes that it is dependent on. So whether the interference dependence edge is traversed in Phase 1 or Phase 2, the node must be sliced in Phase 1.

  This (as we will prove later) also solves the problem of transitive dependencies that cross threads.

  Consider the example in Figure 5.2; the interference dependence of node `x=a` on `a=y` is discovered only during a Phase 2 traversal. However, unless `a=y` is sliced in Phase 1, all its calling contexts will not get included, resulting in an incorrect slice.

- Consider a node S_i in θ_i which is interference dependent on a node S_j in θ_j. Let us slice θ_j, starting with S_j as the slicing criterion in Phase 1. During the slice, let a node S_k be sliced in Phase 2. Further, let S_i be interference dependent (or transitively dependent on) S_k. Now although S_k has already been sliced in Phase 2, S_k needs to be sliced again in Phase 1, since Phase 2 generates only a subset of all the nodes that S_k is dependent on.

5.3.1 The Algorithm

The slicing algorithm essentially puts a loop around the traditional two-phase algorithm [49]. The purpose of the loop is to ensure that nodes reached via an interference dependence edge are sliced in Phase 1. Our slicing algorithm uses three lists. Each time a node is reached through an interference dependence edge, it is added to the outermost worklist w_0 and subjected to a complete two-phase slice. The algorithm iteratively applies a 2-phase slice to every node in w_0.

Our algorithm uses three colors for marking a node. We call the colors `phase1`, `phase2` and `undefined` and define a transitive order `undefined` < `phase2` < `phase1` on them. Initially all the nodes in the TSDG are colored `undefined`. In Phase 1, if the algorithm finds a node that is colored `undefined` or `phase2`, then the node is colored `phase1` and added to the slice. A node that is already colored `phase1` need not be sliced again. In Phase 2, if the algorithm finds a node that is colored `undefined`, it is colored `phase2` and added to the slice. Nodes that are already colored `phase1` or `phase2` need not be

Input: the slicing criterion s
Output: the slice $\mathcal{S}$
$\mathbf{w0} = \{s\}$ /* *Initialization* */

```
while w0 ≠ ∅ do
    remove next element x from w0
    w1 = {x}

    /* Phase 1 */
    while w1 ≠ ∅ do
        remove next element x from w1
        for all y | (y, x) ∈ E_id
            if y.color < phase1 then
                y.color = phase1
                w0 = w0 ∪ y
        for all y | (y, x) ∈ E_dd ∪ E_cd ∪ E_s ∪ E_pi ∪ E_c
            if y.color < phase1 then
                y.color = phase1
                w1 = w1 ∪ y
        for all y | (y, x) ∈ E_po
            if y.color < phase2 then
                y.color = phase2
                w2 = w2 ∪ y
    endwhile

    /* Phase 2 */
    while w2 ≠ ∅ do
        remove next element x from w2
        for all y | (y, x) ∈ E_id
            if y.color < phase1 then
                y.color = phase1
                w0 = w0 ∪ y
        for all y | (y, x) ∈ E_dd ∪ E_cd ∪ E_s ∪ E_po
            if y.color < phase2 then
                y.color = phase2
                w2 = w2 ∪ y
    endwhile

endwhile
```

Figure 5.6: The Three-Color Iterated Two Phase Slicing Algorithm

sliced again. The final slice consists of every node that has been colored either `phase1` or `phase2`. The algorithm is given in Figure 5.6.

Example 1: Let us slice the program in Figure 5.2. Initially `print x` is added to the worklist w_0. Then `print x` is extracted and a 2-phase algorithm is applied to it. In Phase 1, `print x` is data dependent on `x=x_out` which is parameter-out dependent on `x_out=x` in procedure `P1`. `x=x_out` is inserted into w_2.

In Phase 2, `x_out=x` is data dependent on `x=a`. `x=a` is data dependent on `a=0`, interference dependent on `a=y` and `a=z` in procedure `P2` in thread θ_2 and interference dependent on `a=a_out` in θ_2. `a=y`, `a=z` and `a=a_out` are inserted into w_0 and `a=0` is inserted into w_2. Next `a=0` is sliced. No further dependencies are found and the first iteration ends.

In the next iteration let us assume `a=a_out` is processed. In Phase 1 `a=a_out` is parameter-out dependent on `a_out=a` which is added to w_2. Then `a=a_out` is control dependent on `call P2` which is control dependent on `START` θ_2. `a=a_out` is summary dependent on `z_in=z` which is data dependent on `z=20`.

In Phase 2, `a_out=a` is data dependent on `a=z` which is already colored `phase1`. No other dependencies are found and this completes the second iteration.

Let `a=y` be extracted from w_0 in the next iteration. In Phase 1, `a=y` is data dependent on `y=y_in` which is parameter-in dependent on `y_in=y` which is data dependent on `y=10`. This completes the iteration and nothing new is found in the iteration that extracts `a=z`. Thus, the algorithm generates a complete and correct slice.

Example 2: Consider the slice of K_{12} in Figure 5.3.

- Iteration 1:
  - Phase 1: $K_{12} \leftarrow K_1 \leftarrow M_4 \leftarrow M_1$. $K_{12} \leftarrow K_{11}$. $K_{11} \leftarrow K_7 \leftarrow M_3$. $K_{11} \leftarrow K_{26}$.
  - Phase 2: $K_{26} \leftarrow K_{25} \leftarrow K_{18}$. $K_{25} \leftarrow N_4$. (N_4 is added to w_0).
- Iteration 2:
  - Phase 1: $N_4 \leftarrow N_1$. $N_4 \leftarrow N_3 \leftarrow N_8$ (N_8 is added to w_2).
  - Phase 2: $N_8 \leftarrow N_7 \leftarrow N_6 \leftarrow N_5$. $N_7 \leftarrow K_{15}$. $N_7 \leftarrow K_6$. $N_7 \leftarrow K_{24}$. $N_7 \leftarrow K_{30}$. $N_7 \leftarrow K_{21}$. All of K_{15}, K_6, K_{24}, K_{30}, $_{21}$ are added to w_0.
- Iteration 3:
  - Phase 1: $K_{21} \leftarrow K_{17}$. $K_{21} \leftarrow K_{19} \leftarrow K_8 \leftarrow K_2$. Since the algorithm is in Phase 1, the parameter-in edge is traversed.

- Iteration 4:
  - Phase 1: $K_{24} \leftarrow K_{22} \leftarrow K_{20} \leftarrow K_9 \leftarrow K_3$. $K_{24} \leftarrow K_{31}$ (K_{31} is added to w_2).
  - Phase 2: $K_{31} \leftarrow K_{30} \leftarrow K_{29}$. Since the algorithm is now in Phase 2, the parameter-in edge to K_4 is not traversed.
- Iteration 5:
  - Phase 1: $K_{30} \leftarrow K_{29}$. $K_{29} \leftarrow K_4 \leftarrow K_2$. $K_{29} \leftarrow K_{22}$. $K_{29} \leftarrow K_{13}$. In Iteration 4, K_{30} was colored `phase2`. In this iteration it is colored `phase1` and sliced again. This time the parameter-in edges are also traversed.
- Iteration 6:
  - Phase 1: $K_6 \leftarrow K_4$.
- Iteration 7:
  - Phase 1: $K_{15} \leftarrow K_{14}$. $K_{15} \leftarrow K_{13} \leftarrow K_3$.

Note that the slice computed is correct but not precise. The call to $f2(b)$ at K_{14} is added to the slice although it is not possible for the value of a at K_{12} to be affected by the call at K_{14}. The other unnecessary nodes added to the slice are K_{15} and K_{13}.

5.3.2 Complexity

Each node in the graph may be colored at most twice. Hence, the maximum number of nodes handled by the algorithm is $2 * N$ and the complexity of the algorithm is $O(N)$.

5.4 A Context-Sensitive Slicing Algorithm

On the ITCFG, G^* of an interprocedural concurrent program we define: an **interprocedural threaded witness** is *an ordered sequence of nodes* $\langle n_1, n_2, \ldots, n_k \rangle$ *belonging to* G^* *s.t. any subsequence of nodes* n_{i_1}, n_{i_2}, ..., n_{i_k} *belonging to the same thread,* θ_i*, form a realizable path in* θ_i.

On the TSDG we define the slice of an interprocedural concurrent program as follows : The **slice** $\mathcal{S}(p)$ of a TSDG at a node p consists of all nodes q on which p transitively depends in such a manner that only interprocedural threaded witnesses are generated in the slice. More formally,

$$\mathcal{S}(p) = \{ q \mid \mathcal{P} = \langle n_1, \ldots, n_k \rangle,$$
$$q = n_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{k-1}} n_k = p, e_i \in E'$$
$$E' = E_{dd} \cup E_{cd} \cup E_{id} \cup E_c \cup E_{pi} \cup E_{po}$$
$\mathcal{P}$ is an interprocedural threaded witness in the corresponding ITCFG $\}$

5.4.1 The Algorithm

The context-sensitive algorithm is an extension of the context-insensitive algorithm with modifications to ensure that only realizable paths are traversed.

The algorithm takes as input the slicing criterion, the ISCR graph of each thread and the TSDG. It starts with the slicing criterion, s, and inserts it into the outermost list w_0. It keeps extracting a node from w_0 and applies a 2-phase algorithm to it. The same coloring scheme as in the context-insensitive algorithm is applied to the elements of the worklists in the context-sensitive algorithm also.

To determine the existence of a realizable path for each thread, a tuple of nodes containing one node per thread is maintained (similar to the tuples used in the intraprocedural algorithm [53]). In the tuple, the node corresponding to a thread is the last node reached (so far) in that thread. Along with the node we also keep the valid set of topological numbers associated with the node and the color of the node. So each element in the state tuple consists of three items, (i) the last node visited by the algorithm in that thread, (ii) the valid set of topological numbers associated with that node and (iii) the color of the node. When a node S_i in thread θ_i is visited, we update the tuple entry for that thread. Later if the algorithm visits a node, S_j, in the same thread θ_i we check if there is a *realizable* path from S_j to S_i using the *RealizablePath* test. If so, S_j is added to the slice (along with the valid set of topological numbers) else it is rejected.

The function $RealizablePath(i, j, mu)$ was described in Section 4.2.3. Here it has the additional functionality that if j is $\perp$, then it implies that the thread has not yet been visited and *RealizablePath* returns all the topological numbers associated with i.

We use the notation $\{X_y, \langle a, b, \cdots, n \rangle\}$ to represent an ISCR/node numbered X_y with the set of topological numbers $\langle a, b, \cdots, n \rangle$. We use the symbol $\perp$ in the state tuple to indicate that the corresponding thread has not yet been visited. The function $\theta(x)$ returns the index of the thread to which x belongs.

Example 1: Let us apply this algorithm to K_{12} in Figure 5.3. In the figure the nodes are numbered on the left for identification and the topological numbers are in circles on the

Input: the slicing criterion s
Output: the slice $\mathcal{S}$
mu = GetTopologicalNumberSet(s)

$$C = (s, [t_0, \ldots, t_n], \text{phase1}), \begin{cases} \textbf{if } \theta(s) == \theta_i \textbf{ then } t_i.node = s,\ t_i.\mu = mu \\ \textbf{else } t_i.node = \perp,\ t_i.\mu = \emptyset \end{cases}$$

$\mathbf{w0} = \{C\}$

while w0 $\neq \emptyset$ **do**
    remove next element from **w0** and insert into **w1**
    while w1 $\neq \emptyset$ **do** /* Phase 1 */
        remove next element $c = (x, T, color)$ from **w1**
        for all $y \mid (y, x) \in E_{id}$
            $t = T[\theta(y)].node$, $mu = T[\theta(y)].\mu$
            Insert (y, $RealizablePath(y, t, mu)$, T, `phase1`, **w0**)
        for all $y \mid (y, x) \in E_{dd} \cup E_{cd} \cup E_s \cup E_{pi} \cup E_c$
            $t = T[\theta(y)].node$, $mu = T[\theta(y)].\mu$
            Insert (y, $RealizablePath(y, t, mu)$, T, `phase1`, **w1**)
        for all $y \mid (y, x) \in E_{po}$
            $mu = T[\theta(x)].\mu$
            Insert (y, $RealizablePath(y, x, mu)$, T, `phase2`, **w2**)
    endwhile

    while w2 $\neq \emptyset$ **do** /* Phase 2 */
        remove next element $c = (x, T, color)$ from **w2**
        for all $y \mid (y, x) \in E_{id}$
            $t = T[\theta(y)].node$, $mu = T[\theta(y)].\mu$
            Insert (y, $RealizablePath(y, t, mu)$, T, `phase1`, **w0**)
        for all $y \mid (y, x) \in E_{dd} \cup E_{cd} \cup E_s \cup E_{po}$
            $mu = T[\theta(x)].\mu$
            Insert (y, $RealizablePath(y, x, mu)$, T, `phase2`, **w2**)
    endwhile
endwhile

procedure Insert (Node: y, TopologicalNumbersSet: mu, Tuple: T, int color, List: $\mathcal{L}$)
    if $mu == \emptyset$ **then return**
    $T[\theta(y)].node = y$
    $T[\theta(y)].\mu = mu$
    $c' = (y, T, color)$
    if c' has not already been calculated **then**
        $\mathcal{S} = \mathcal{S} \cup \{y\}, \mathcal{L} = \mathcal{L} \cup \{c'\}$

Figure 5.7: The Context-Sensitive Interprocedural Slicing Algorithm

right. For simplicity, we use the short notation S_x to represent a node and the expanded form $\{X_y, \langle a, b, \cdots, n\rangle\}$ only at important points.

- Iteration 1:
  - Phase 1: $\{K_{12},\ [(\perp, \emptyset),\ (K_{12}, \langle 33\rangle),\ (\perp, \emptyset)]\} \leftarrow K_1 \leftarrow M_4 \leftarrow M_1$.
    The notation $\{K_{12},\ [(\perp, \emptyset),\ (K_{12}, \langle 33\rangle),\ (\perp, \emptyset)]\}$ indicates that the node sliced is K_{12}. The state of θ_0 is $(\perp, \emptyset)$ indicating that it has not yet been visited. The state of θ_1 is $(K_{12}, \langle 33\rangle)$ indicating that the last node visited in θ_1 was K_{12} and the corresponding valid topological number set is $\langle 33\rangle$. The state of θ_2 is also $(\perp, \emptyset)$ indicating that it has not yet been visited.
    $K_{12} \leftarrow K_{11}$.
    $K_{11} \leftarrow K_7 \leftarrow M_3$.
    $K_{11} \leftarrow K_{26}$.
  - Phase 2: $K_{26} \leftarrow K_{25} \leftarrow K_{18}$.
    $\{K_{25},\ [(\perp, \emptyset),\ (K_{25}, \langle 29\rangle),\ (\perp, \emptyset)]\} \leftarrow \{N_4,\ [(\perp, \emptyset),\ (K_{25}, \langle 29\rangle)\ (N_4, \langle 4\rangle)]\}$. ($N_4$ is added to w_0). The algorithm records the fact that the last node visited in θ_1 was K_{25} with topological number set $\langle 29\rangle$.
- Iteration 2:
  - Phase 1: $N_4 \leftarrow N_1$. $N_4 \leftarrow N_3 \leftarrow N_8$ (N_8 is added to w_2).
  - Phase 2: $N_8 \leftarrow N_7 \leftarrow N_6 \leftarrow N_5$.
    $\{N_7,\ [(\perp, \emptyset),\ (K_{25}, \langle 29\rangle)\ (N_7, \langle 5\rangle)]\} \leftarrow \{K_{15},\ [(\perp, \emptyset),\ (K_{15}, \langle 41\rangle),\ (N_7, \langle 5\rangle)]\}$.
    In this case, since there is no path from the topological number $\langle 41\rangle$ in θ_1 to $\langle 29\rangle$, this node is rejected by the slice giving a precise solution.
    $N_7 \leftarrow K_6$.
    $N_7 \leftarrow K_{24}$.
    $\{N_7,\ [(\perp, \emptyset),\ (K_{25}, \langle 29\rangle)\ (N_7, \langle 5\rangle)]\} \leftarrow \{K_{30},\ [(\perp, \emptyset),\ (K_{30}, \langle 8, 25\rangle),\ (N_7, \langle 5\rangle)]\}$.
    The topological number set of K_{30} is $\langle 8, 25, 38\rangle$ of which there is no path from $\langle 38\rangle$ to $\langle 29\rangle$. Hence $RealizablePath$ returns a subset of the possible topological numbers $\langle 8, 25\rangle$ which are the valid numbers for the node at this point in the slice.
    $N_7 \leftarrow K_{21}$.
    All of K_6, K_{24}, K_{30}, K_{21} are added to w_0.
- Iteration 3:
  - Phase 1: $K_{21} \leftarrow K_{17}$. $K_{21} \leftarrow K_{19} \leftarrow K_8 \leftarrow K_2$. Since the algorithm is in Phase 1, the parameter-in edge is traversed.

- Iteration 4:

  - Phase 1: $K_{24} \leftarrow K_{22} \leftarrow K_{20} \leftarrow K_9 \leftarrow K_3$. $K_{24} \leftarrow K_{31}$ (K_{31} is added to w_2).
  - Phase 2: $K_{31} \leftarrow K_{30} \leftarrow K_{29}$. Since the algorithm is now in Phase 2, the parameter-in edge to K_4 is not traversed.

- Iteration 5:

  - Phase 1: $\{K_{30}, [(\perp, \emptyset), (K_{30}, \langle 8, 25\rangle), (N_7, \langle 5\rangle)]\} \leftarrow \{K_{29}, [(\perp, \emptyset), (K_{29}, \langle 7, 24\rangle), (N_7, \langle 5\rangle)]\}$. Again, a subset of the topological numbers is allowed.
    $K_{29} \leftarrow K_4 \leftarrow K_2$. $K_{29} \leftarrow K_{22}$. $\{K_{29}, [(\perp, \emptyset), (K_{29}, \langle 7, 24\rangle), (N_7, \langle 5\rangle)]\} \leftarrow \{K_{13}, [(\perp, \emptyset), (K_{13}, \langle 34\rangle), (N_7, \langle 5\rangle)]\}$. In this case, K_{13} is rejected as there is no path from $\langle 34\rangle$ to $\langle 7, 24\rangle$. In Iteration 4, K_{30} was colored `phase2`. In this iteration it is colored `phase1` and sliced again.

- Iteration 6:

  - Phase 1: $K_6 \leftarrow K_4$.

The slice computed is both correct and precise.

Example 2: Consider the slice of m_{14} in Figure 5.5 again. In Figure 5.4 we have shown the topological ordering for each thread in circles on the right end of each node. The numbers are *local* to a thread. The ISCRs in $f1$ have two numbers each corresponding to the two call sites. All others have only one topological number each. The *Reach* values are not shown in the figure for lack of space but in this example $i \in j.Reach$ if $i < j$ as there are no branches in the program. The topological numbers are also depicted in the corresponding TSDG shown in Figure 5.5.

The algorithm starts by inserting $\{m_{14}, [(\perp, \emptyset), (m_{14}, \langle 30\rangle), (\perp, \emptyset)\}$ into w_0. Here the state tuple associated with θ_0 and θ_2 is $(\perp, \emptyset)$ as these threads have not yet been visited and the state tuple for θ_1 is $(m_{14}, \langle 30\rangle)$ indicating that the last node visited in θ_1 was m_{14} and its corresponding valid topological numbers set is $\langle 30\rangle$.

The following dependencies are traced in Phase 1: $m_{14} \leftarrow m_{12} \leftarrow m_{21}$. In Phase 2: $m_{21} \leftarrow m_{19} \leftarrow n_3$ in θ_2. The element inserted into w_0 will be $\{n_3, [(\perp, \emptyset), (m_{19}, \langle 9, 23\rangle), (n_3, \langle 3\rangle)]\}$. The state tuple for θ_0 is still $(\perp, \emptyset)$, for θ_1 it is $(m_{19}, \langle 9, 23\rangle)$ indicating that m_{19} was the last node visited in θ_1 and its corresponding valid topological numbers are 9 and 23.

In the second iteration, in Phase 1, $n_3 \leftarrow m_{20}$. The possible topological numbers for m_{20} are 10 and 24 but $RealizablePath(m_{20}, m_{19}, \langle 9, 23\rangle)$ returns $\langle 10\rangle$ since $24 \notin$

9.*Reach* and $24 \notin 23.Reach$ but $10 \in 24.Reach$. The element inserted into w_0 now is $\{m_{20}, [(\perp, \emptyset), (m_{20}, \langle 10\rangle), (n_3, \langle 3\rangle)]\}$ indicating that not all topological numbers associated with m_{20} are now valid.

In the third iteration the algorithm follows the interference dependence from m_{20} to n_2 and inserts $\{n_2, [(\perp, \emptyset), (m_{20}, \langle 10\rangle), (n_2, \langle 2\rangle)]\}$ into w_0.

In the fourth iteration, in Phase 1, the algorithm finds that n_2 is interference dependent on m_8 and m_2. When trying to add m_8, $RealizablePath(m_8, m_{20}, \langle 10\rangle)$ returns $\emptyset$ since $16 \notin 10.Reach$ and hence the algorithm rejects m_8 from the slice but correctly accepts m_2 into the slice as $2 \in 10.Reach$.

Proof of Correctness Let p be the slicing criterion in a given TSDG; let S be the slice computed by the algorithm. Then the correctness of the algorithm can be stated as

Theorem 4 $\mathcal{S}(p) = S$

The proof is given in Section 5.8.1

Complexity In intraprocedural programs each node in the slice may be inserted into the worklist $O(n^{t-1})$ times [68], where n is the number of nodes in a thread and t is the number of threads. This gives a slicing complexity of $O(N^t)$, where N is the number of nodes in the graph. In interprocedural programs, a node may be represented by an exponential number of topological numbers and for each topological number it may be inserted into the slice an exponential number of times. This makes the complexity of the algorithm doubly exponential $O(N^{p^t})$, where p is the calling depth of the call graph. However, in Chapter 8, we show that with certain optimizations, the algorithm is practical.

5.5 Nested Threads

In Figure 5.8 S_4 in θ_1 is interference dependent on S_8 in θ_2. S_8 is interference dependent on S_6 in θ_3. Since, the thread θ_3 has never been visited before, the algorithm would add S_6 to the slice although it is clear that S_4 is not dependent on S_6. In the intraprocedural slicing algorithm [68], when a node S_i in θ_i is visited, we update every element of the tuple that corresponds to a thread that does not execute in parallel with θ_i. A similar technique is used for interprocedural programs: if S_i is dependent on some node $[S_j, T_j, color_j]$, the valid topological number set associated with S_i is μ_i and $color_i$ is determined as explained before, then we create a new tuple $[S_i, T_i, color_i]$ such that

if $||(\theta(S_i), \theta_k)$ is false then

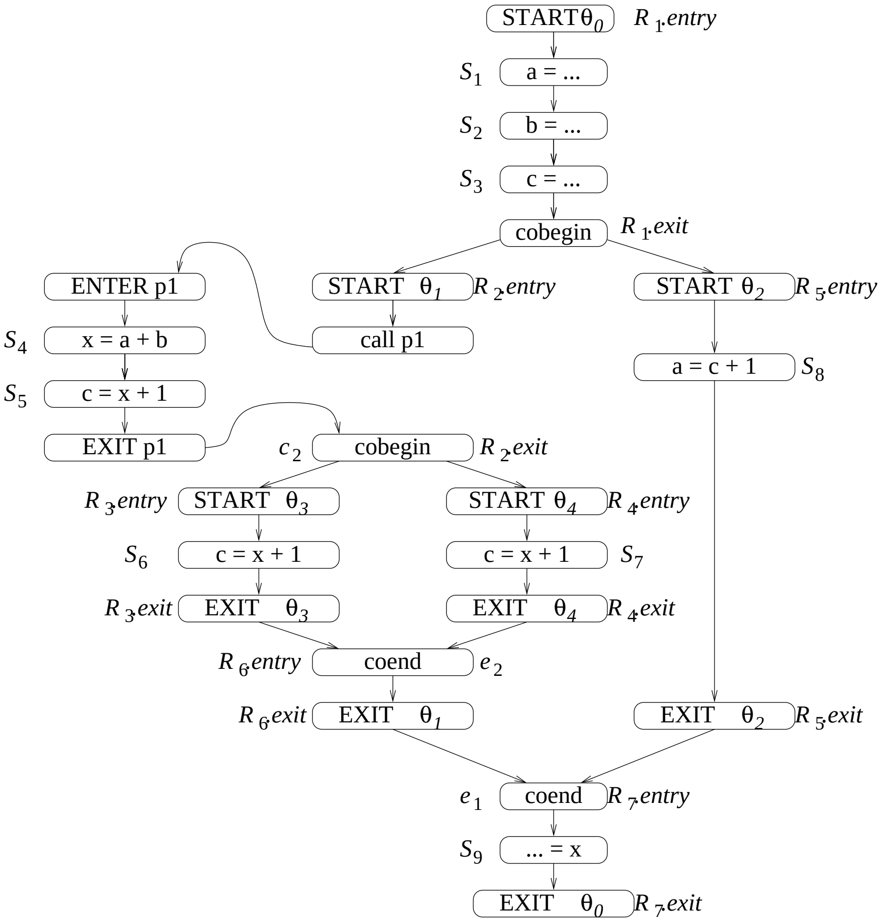


Figure 5.8: Nested threads

$T_i[\theta_k].node = S_i$
$T_i[\theta_k].\mu = \mu_i$
$T_i[\theta_k].color = color_i$

else

$T_i[\theta_k].node = T_j[\theta_k].node$
$T_i[\theta_k].\mu = T_j[\theta_k].\mu$
$T_i[\theta_k].color = T_j[\theta_k].color$

Consider the slice of S_4 in Figure 5.8.

- S_4 is inserted into w_1 as $\{S_4, [(S_4, \mu_1, \text{phase1}), (S_4, \mu_1, \text{phase1}), (\perp, \emptyset, \text{undefined}), (S_4, \mu_1, \text{phase1}), (S_4, \mu_1, \text{phase1})]\}$. Since, θ_1 executes sequentially with θ_0, θ_3 and θ_4, they are marked in the same way as θ_1.

- S_4 is interference dependent on S_8 which is added to the slice as $\{S_8,\ [(S_8,\ \mu_2,$ phase1$), (S_4, \mu_1, \text{phase1}), (S_8, \mu_2, \text{phase1}), (S_4, \mu_1, \text{phase1}), (S_4, \mu_1, \text{phase1})]\}$.

- S_6 is interference dependent on S_8, S_6 belongs to θ_3 and the tuple entry for θ_3 has $(S_4, \mu_1, \text{phase1})$. Since the *RealizablePath* algorithm is applied only to a single thread we need some way to determine whether there is a path from S_6 to S_4.

We need some algorithm to determine whether there is a path from a node in some thread θ_i to another node in a sequentially executing thread θ_j. The threaded control flow graph has a properly nested structure and hence it is easy to divide it into a set of regions that have a proper global ordering.

Each thread is broken into a set of single-entry-single-exit regions, where the entry node is either the $START\theta_i$ node or a `coend` node and the exit node of the region is either a `cobegin` node or the $EXIT\theta_i$ node of the thread. In Figure 5.8, θ_0 is broken into two regions, $\mathcal{R}_1$ and $\mathcal{R}_7$. The region $\mathcal{R}_1$ has its entry node as $START\theta_0$ and the exit node is the `cobegin` node. The region $\mathcal{R}_7$ has the entry node e_1 (`coend`) and the exit node is $EXIT\theta_0$. Similarly, θ_1 is broken into two regions, $\mathcal{R}_2$ and $\mathcal{R}_6$ and the other threads have only one region each.

We calculate reachability for these regions and store the information in bitvectors. A node in a thread may belong to one or more regions. Each topological number of a node maps to a specific region. When we apply $RealizablePath(S_i, S_j, \mu)$, if S_i and S_j belong to different threads then $RealizablePath()$ has the following functionality

- Let $\mathcal{R}_j$ be the regions associated with μ.

- Determine the regions of S_i that have a path to some region in $\mathcal{R}_j$ and denote this as $\mathcal{R}_i$.
- Return the topological numbers of S_i that map to some region in $\mathcal{R}_i$ or return $\emptyset$ if $\mathcal{R}_i$ is empty.

In the example, there is no path from $\mathcal{R}_3$ (the region to which S_6 belongs) to $\mathcal{R}_2$ (the region to which S_4 belongs) and hence, S_6 is correctly rejected from the slice.

5.6 Threads Nested within Loops

In intraprocedural programs, we needed to generate a hierarchy of reachability bit vectors. The RealizablePath algorithm cannot be adapted to a hierarchy of topological numbers. We could not find a way to map a set of topological numbers from one hierarchy to another. In intraprocedural programs, this was not a problem as paths are transitive and hence there were no topological numbers involved. A single node was sufficient as a place holder for a thread. Even if the mapping were possible the memory requirement would be too high.

Instead, we use the conservative estimate, that any node within a loop is reachable from any other node in the loop. If a loop encloses a `cobegin-coend` construct, then we create a separate region for the entire loop and every node in the loop belongs to this region. In Figure 5.9 we show the regions created when a `cobegin-coend` construct is nested within a loop. In the example, θ_1 is divided into three regions as shown. We need not determine topological numbering of realizable path computation for θ_3 and θ_4, as every node in θ_3 is reachable from every other node and similarly for θ_4. We assign all the nodes within the loop to the same ISCR. The rest of the algorithm remains unchanged.

5.7 Conclusion

In this chapter we have given a context-*in*sensitive solution to slicing concurrent interprocedural programs that is efficient and *correct* though not precise. Then we have given a *context-sensitive* slicing algorithm for concurrent interprocedural programs. To the best of our knowledge, no one else has attempted to solve the issue of context-sensitivity. We have shown how to extend the analysis to handle nested threads. For threads nested within loops we give only a conservative solution. In Chapter 8 we will apply this algorithm to the Java concurrency model and give implementation statistics.

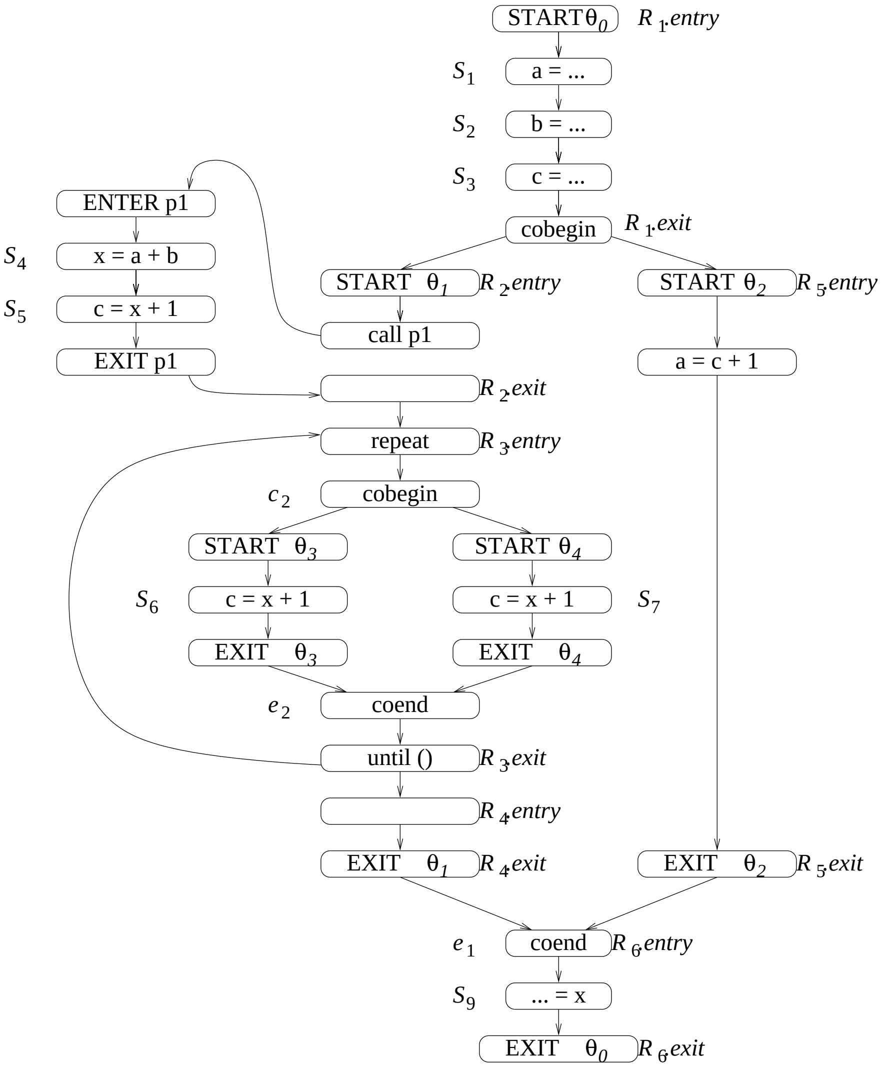


Figure 5.9: Threads nested within loops

5.8 Appendix

5.8.1 Proof of Correctness of the Slicing Algorithm

Definition 8 *There is an* intra-*thread transitive dependence from a node S_i to a node S_j, if there is a sequence of dependencies $S_i \xrightarrow{e_i} \ldots \xrightarrow{e_k} S_j$ such that none of e_i is an interference dependence edge.*

Theorem 5 *If there is an* intra-*thread transitive dependence from an ActualIn node to an ActualOut node at a call site of a procedure, then there is a summary edge from the ActualIn node to the ActualOut node. Conversely, if there is a summary edge from an ActualIn node to an ActualOut node, then there is an* intra-*thread transitive dependence from the corresponding FormalIn node to the corresponding FormalOut node [76, 49].*

Lemma 5 *A tuple $(q_1, T_1, color_1)$ is added to the one of the lists w_0, w_1, or w_2, at some stage if and only if there exists a sequence of tuples $(q_i, T_i, color_i)$, $i = 1, \ldots, n$ that is added to one of these lists where the tuples satisfy the following properties:*

1. *For $i = n$*
   - $q_n = p$
   - $T_n(\theta_j).node = \perp$ `if` $\theta_j \neq \theta(q_n)$
     $T_n(\theta_j).node = q_n$ `if` $\theta_j == \theta(q_n)$
   - $color_n =$ `phase1`.
2. *For $i = 1, \ldots, n-1, n > 1$*
   - $q_i \xrightarrow{e_i} q_{i+1}$, $e_i \in \{cd, dd, c, s, pi, po, id\}$
   - $color_i \in \{phase1,\ phase2\}$
   - *For $j = 1, \ldots, |\theta|$*
     - $T_i[\theta_j].node$ $= T_{i+1}[\theta_j].node$, `if` $\theta_j \neq \theta(q_i)$
       $= q_i$, `if` $\theta_j == \theta(q_i)$
     - $T_i[\theta_j].\mu$ $= RealizablePath(q_i, T_{i+1}[\theta_j].node, T_{i+1}[\theta_j].\mu)$ `if` $\theta_j \neq \theta(q_i)$
       $= T_{i+1}[\theta_j].\mu$ `if` $\theta_j == \theta(q_i)$
     - $T_i[\theta_j].\mu \neq \emptyset$
3. *For $i = 1, \ldots, (n-1)$,* one *of the following holds*
   - $e_i \in \{cd, dd, s\}$
     $color_i = color_{i+1}$

- $e_i = po$
  $color_i \leq color_{i+1}$
- $e_i \in \{pi, c\}$
  $color_i = color_{i+1} =$ `phase1`
- $e_i = id$
  $color_i =$ `phase1`

Proof: There are two parts to this proof:
"If" part: This is obvious.
"Only If" part: Assume that $(q_1, T_1, color_1)$ is added to one of the lists. We need to show that there exists a sequence of tuples $(q_i, T_i, color_i)$, $i = 1, \ldots, n$ having the desired properties that get added to one of the lists. This follows by computational induction from the following two facts:

1. The tuple $(p, T, phase1)$ has such a sequence, where $T[\theta].node = \perp$, for all $\theta \neq \theta(p)$ and $T[\theta(p)].node = p$.

2. If every tuple that is already added to one of the lists has such a sequence, then every new tuple that gets added because of these tuples, has a sequence.

(1) is obvious. (2) follows from the fact that the sequence for the new tuple is got from the sequence corresponding to one of the tuples already in the list.

Lemma 6 *A node S_i is colored* `phase2` *if and only if there exists a sequence of tuples $S_i \rightarrow \ldots \rightarrow S_j \rightarrow S_k$, such that S_j is a $FormalOut$ node in the same procedure as S_i and S_j is also colored* `phase2`*; and S_k is an ActualOut node which may be colored* `phase1` *or* `phase2` *(S_j is parameter-out dependent on S_k).*

Proof This follows from Definition 8 and the fact that all nodes reached by interference dependence edges are colored `phase1`. Hence it is not possible to enter `phase2` except via a parameter-out dependence edge originating at an *ActualOut* node.

Lemma 7 *Given a node S_j and a topological number set μ such that $\mu \neq \emptyset$ and another node S_i such that $S_i \xrightarrow{*} S_j$ and for each edge e_i in the path from S_i to S_j we have $e_i \in \{dd, cd, s, po, c, pi\}$, then $RealizablePath$ $(S_i, S_j, \mu) \neq \emptyset$.*

Proof: By property of data dependence, control dependence, summary dependence, parameter-out dependence, call dependence and parameter-in dependence, there is a realizable path from S_i to S_j.

Theorem 6 *$\mathcal{S}(p) \subseteq S$. If $q \in \mathcal{S}(p)$ then it is added to the slice S.*

Proof: Assume $q \in \mathcal{S}(p)$, then by definition of $\mathcal{S}(p)$, there is a threaded witness from q to p. Let the threaded witness be $\langle q = q_1, q_2, \ldots, q_n = p \rangle$. We shall construct a sequence of tuples $(q_i, T_i, color_i), i = 1, \cdots, n$, that satisfy the property mentioned in Lemma 7, thereby proving the theorem.

Let T_n and $color_n$ be as given in (1) of the property of Lemma 5.

Now, we define inductively T_i given T_{i+1} for $i = 1, \cdots (n-1)$. For each pair of nodes in the witness, q_i and q_{i+1}, let $q_i \xrightarrow{e} q_{i+1}$, where $e \in E'$, i.e., e is of type 'cd', 'dd', 'c', 'pi', 'po', or 'id' (but not 's'). The definition of T_i depends upon the type of e. We shall define T_i for each of these types:

1. e is 'cd' or 'dd' and q_i and q_{i+1} belong to the same thread $\theta(q_i)$:
   In the tuple T_{i+1} associated with q_{i+1}, let $T_{i+1}[\theta(q_i)].\mu$ be μ_{i+1}. Then at q_i we have the tuple T_i such that $T_i[\theta(q_i)].\mu = RealizablePath(q_i, q_{i+1}, \mu_{i+1})$. Since q_i and q_{i+1} belong to a threaded witness and since q_i and q_{i+1} belong to the same thread, there must be a (realizable) path from q_i to q_{i+1} and by Lemma 7, we get $T_i[\theta(q_i)].\mu \neq \emptyset$.

   Also from the algorithm, we get $color_i = color_{i+1}$.

2. e is 'id' (Obviously, q_i and q_{i+1} belong to different threads) or case 'cd', 'dd' such that q_i and q_{i+1} belong to different threads:
   In the tuple T_{i+1} associated with q_{i+1}, let q' be the previous node belonging to $\theta(q_i)$ in the threaded witness. Then $T_{i+1}[\theta(q_i)].node = q'$ and let $T_{i+1}[\theta(q_i)].\mu$ be μ'. Then at q_i we have the tuple T_i such that $T_i[\theta(q_i)].\mu = RealizablePath(q_i, q', \mu')$. If q' is $\perp$ then $RealizablePath(q_i, q', \mu')$ will return $GetTopologicalNumberSet(q_i)$ and hence is not an empty set. Else, since q_i and q' belong to a threaded witness and since q_i and q' belong to the same thread, there must be a path from q_i to q' and by Lemma 7, we get $\mu' \neq \emptyset$.

   Also from the algorithm, we get $color_i = color_{i+1}$ for 'cd' and 'dd' and for 'id' edges $color_i = phase1$.

3. e is 'pi' : if $color_{i+1}$ is `phase1` then all the properties of Lemma 5 are met in the same way as item 1. If $color_{i+1}$ is `phase2`, then by Lemma 6 there is a sequence of nodes $\langle q_i, q_{i+1}, \ldots, q_f, q_a \rangle$, such that all the nodes belong to $\theta(q_i)$. Further q_f is a $FormalOut$ node with color `phase2`, q_a is an $ActualOut$ node with color `phase1` or `phase2`. Also by Lemma 6, q_a is in the sequence. Further, by Theorem 5 q_a is summary dependent on q_i.

   Let the state tuple associated with q_a be T_a such that $T_a[\theta(q_i)].\mu = \mu_a$. Then at q_i we have the tuple T_i such that $T_i[\theta(q_i)].\mu = \mu_i = RealizablePath(q_i, q_a, \mu_a)$. By Theorem 5 there is a path from q_i to q_a and by Lemma 7, we get $\mu_i \neq \emptyset$.

Also from the algorithm, we get $color_i = color_{i+1}$ since there is summary edge from q_i to q_a.

4. case 'c' : if $color_{i+1}$ is `phase1` then all the properties of Lemma 5 are met in the same way as for item 1. If $color_{i+1}$ is `phase2`, then by Lemma 6 there is a sequence of nodes $q_i, \ldots, q_f, q_a$, such that q_f is a *FormalOut* node with color `phase2`, q_a is an *ActualOut* node with color `phase1` or `phase2`, and q_a is call dependent on q_i (by construction).

   Let the state tuple associated with q_a be T_a such that $T_a[\theta(q_i)].\mu = \mu_a$. Then at q_i we have the tuple T_i such that $T_i[\theta(q_i)].\mu = \mu_i = RealizablePath(q_i, q_a, \mu_a)$. By construction, there is a path from q_i to q_a and by Lemma 7, we get $\mu_i \neq \emptyset$.

5. e is 'po' : $color_i \leq color_{i+1}$ and all the properties of Lemma 5 are met in the same way as for item1.

Now construct the tuples $(q_i, T_i, color_i), i = 1, \cdots, n$. It is easy to see that this sequence satisfies the required properties of Lemma 5. Hence the theorem.

Theorem 7 *Proof of $S \subseteq \mathcal{S}(p)$. If a node q is added to the slice S, then it is in $\mathcal{S}(p)$*

Proof: If q is added to the slice then by Lemma 5 there is a sequence of tuples $(q_i, T_i, color_i)$, $i = 1, \cdots, n$ with $q_1 = q$ that obey the properties of tuples given in Lemma 5. We show that these properties imply that there is a threaded witness for q. We have $q_i \xrightarrow{e_i} q_{i+1}$, where $e_i \in \{cd, dd, c, s, pi, po, id\}$. In order to show that this sequence forms a threaded witness, we need to show that for every thread θ_i, the subsequence of nodes $q^i_1, \cdots, q^i_k$ belonging to thread θ_i form a realizable path.

Let q_i and q_{i+1} be any two consecutive nodes added to the slice. We show inductively that if q_{i+1} forms a part of a threaded witness, then q_i also forms a threaded witness. Consider the different dependence edges from q_i to q_{i+1}.

1. case 'pi', 'po' or 'cd', 'dd' such that q_i and q_{i+1} belong to the same thread (*i.e*, $\theta(q_i) == \theta(q_{i+1})$). (In the case of 'pi' and 'po', $\theta(q_i) == \theta(q_{i+1})$ is always true):
   Let μ_{i+1} be the set of topological numbers associated with q_{i+1}. Then in the state tuple for q_{i+1}, we have $T_{i+1}[\theta(q_i)].\mu = \mu_{i+1}$. In the state tuple for q_i, we have $T_i[\theta(q_i)].\mu = \mu_i$ such that $\mu_i = RealizablePath(q_i, q_{i+1}, \mu_{i+1})$. By Lemma 5 $\mu_i \neq \emptyset$. Then by Corollary 3 (Section 4.5.2) there is a realizable path from q_i to q_{i+1} and hence there is a threaded witness for q_i.

2. case 'id' or 'cd', 'dd' such that q_i and q_{i+1} belong to different threads (*i.e*, $\theta(q_i) \neq \theta(q_{i+1})$):

Let the last node visited by the algorithm in $\theta(q_i)$ be some q'. If q' is $\perp$ then q_i is the first node in its thread and it obviously forms a realizable path in its thread. If q' is not $\perp$, then in the state tuple associated with q_{i+1}, we have $T_{i+1}[\theta(q_i)].node = q'$ and $T_{i+1}[\theta(q_i)].\mu = \mu'$. In the state tuple associated with q_i, we have $T_i[\theta(q_i)].\mu = \mu_i$ such that $\mu_i = RealizablePath(q_i, q', \mu')$. By Lemma 5 $\mu_i \neq \emptyset$. Then by Corollary 3 there is a realizable path from q_i to q' and hence there is a threaded witness for q_i.

3. case 's' : In a threaded witness, no pair of adjacent nodes are related by the summary edges. So, in order to show the existence of threaded witness, we first replace every pair q_i, q_{i+1}, that are related by a summary edge by a sequence of nodes that are related by 'cd', 'dd', 'pi' and 'po' edges alone. This is always possible thanks to the property of summary edges (Theorem 5). Also, by Definition 8 and Theorem 5 there is a transitive dependence from q_i to q_{i+1} composed of only 'cd', 'dd', 'pi', 'po' edges. Then we can apply case 1 of this proof to the sequence of nodes from q_i to q_{i+1} that have only 'cd', 'dd', 'pi', 'po' edges. Thus, there is a realizable path from q_i to q_{i+1} and the nodes belong to a threaded witness.

Thus for any q_i added to the slice, there is a threaded witness in $\theta(q_i)$. Hence the theorem.

Chapter 6

DATA DEPENDENCE ANALYSIS OF SEQUENTIAL JAVA PROGRAMS

In the previous chapters, this thesis has explored issues related to slicing of concurrent interprocedural programs that conform to a simple concurrency model. In this chapter we start the analysis of concurrent Java programs. We start by exploring algorithms for data dependence analysis for the object-oriented model of Java. In later chapters we will also analyze other Java specific factors, such as the `fork/join/` thread model, synchronization and monitors.

In this chapter we present an algorithm for data dependence analysis of sequential Java programs. For intra-procedural analysis we give a new context-sensitive and path-sensitive solution to combined escape analysis and pointer analysis which uses a *sparse* representation. The algorithm also builds the Static Single Assignment form of each procedure and generates the SSA def-use edges. For inter-procedural analysis we use escape analysis to summarize the points-to information of a procedure so that it can be reused in each calling context.

6.1 Introduction

A data dependence from a node n_1 to a node n_2 in the control flow graph of a program implies that the program's computation might be changed if the relative order of the components represented by n_1 and n_2 were reversed. There are several kinds of data dependences [55]. In the context of slicing, we consider only data dependences representing flow-dependences. A data dependence exists from a node n_1 to a node n_2 if n_1 defines a variable x, n_2 uses the variable x and there is a path from n_1 to n_2 along which there is no intervening definition of x. For scalar variables, data dependence can be computed using reaching definition analysis [3]. Determining data dependence in the presence of pointers is comparatively complicated as different access paths may refer to the same memory location. Consider the following program consisting of statements S_1 to S_2 in the first column (ignore the second and third columns for the time being).

S_1: $a =$ new cl	$a = O_1$	$a \to O_1$
S_2: $a.next =$ new cl	$O_1.next = O_2$	$O_1.next \to O_2$
S_3: $b = a$	$b = a$	$b \to O_1$
S_4: $b.next =$ new cl	$O_1.next = O_4$	$O_1.next \to O_4$
S_5: $x = a.next$	$x = O_1.next$	$x \to O_4$

For data dependence analysis of local variables (a, b and x) we can still use reaching definition analysis. However, for field variables ($a.next$, $b.next$), a data dependence analysis based on the simple names of access paths would generate an incorrect data dependence from S_2 to S_5 and would be unable to detect the real data dependence from S_4 to S_5. Clearly, we need to apply some form of pointer analysis before or in conjunction with the data dependence analysis.

Pointer analysis has been the subject of much research. Some solutions trade precision for speed by using context-insensitive or path-insensitive algorithms. *Path-sensitive* analysis takes into account the flow of control between different program points, *path-insensitive* analysis ignores control flow and assumes that statements can execute in any order. *Context-sensitive* analysis considers only program paths along which procedure calls and returns are properly matched; *context-insensitive* analysis does not make this distinction.

However, although there are many applications of data dependence analysis, we are interested in data dependence analysis as a basis for a static slicing tool and hence require as precise a solution as possible. The greater the precision of the data dependence, the greater the precision and speed of the slice generated [60]. In addition, experiments have shown that the precision of points-to information can greatly impact the performance and usefulness of subsequent analyses [80].

Java has a simple memory model: conceptually, all objects are allocated from a garbage-collected heap. In addition, Java does not allow nested objects. An interesting result of this model is that any object may be represented as a tree with only two levels [25]. At the root of the tree is the object allocated from the heap and the children of the root are the fields of the object. If we represent each heap allocated object as O_i, then every field variable can be represented as $O_i.f$. In the example above, the objects allocated on the heap by the `new` statements at S_1, S_2 and S_4 are O_1, O_2 and O_4 respectively. Further, we deduce that $a.next$ and $b.next$ both refer to $O_1.next$. Now, if we apply reaching definitions analysis using the names $O_1.next$ instead of $a.next$ and $b.next$, we get the correct data dependencies for the field variables.

The basic idea behind data dependence analysis is to reduce every field variable to the form $O_i.f$ before applying reaching definitions analysis. The question that arises is, how do we convert field variables to this form ? Currently, (to the best of our knowledge) there are two ways in deriving this information in a *path-sensitive and context-sensitive* manner.

Connection Graphs The first is to build connection graphs [25]. This has been described in Section 2.6. By traversing the connection graph at each program point it is possible to convert each field variable to the form $O_i.f$. However, the connection graph does not permit strong updates of field variables and hence gives conservative results. A strong update at an assignment of a variable v deletes the existing points-to edges of v and inserts the new edges, whereas a weak update merges the old points-to edges and the new points-to edges. As a result of weak updates, in the example above, $b.next$ would point to both O_2 and O_4. Although, in this case this does not affect the accuracy of the data dependence, it is easy to construct an example where this would lead to spurious data dependence edges.

Compositional Pointer and Escape Analysis The second method for converting the field variables into the desired form is to use Whaley and Rinard's pointer analysis [89]. This method essentially builds a complete points-to graph at each program point. A points-to graph explicitly represents, for each dereference point in a program, a superset of the set of symbols or memory locations that may be accessed by that dereference. Their algorithm iteratively applies a transfer function similar to that of Choi *et al* [25] to each statement until it reaches a fixed point. However, this transfer function supports strong updates. In this case also, the pointer analysis does not directly determine the data dependence edges, but the data dependence can be derived from the points-to graphs.

Contributions Both of these methods have the disadvantage that they conceptually require points-to graphs at each program point. A practical implementation may keep points-to graphs at the granularity of a basic block. This works fine for sequential programs but may lead to more conservative results during inter-thread analysis. A more complete comparison of related work is given in Section 6.5. In this chapter we give a new combined points-to, escape analysis and data dependence algorithm that

- is path-sensitive and context-sensitive,
- does not require building points-to graphs at each program point. Instead, our solution uses a sparse representation that needs to track only the points-to information that changes at each program point. This makes our solution more efficient both in terms of space and time complexity.
- The solution builds a complete SSA graph with def-use edges for all variables including field variables. This automatically subsumes data dependence analysis.
- Our pointer analysis is even more precise than Whaley and Rinard's solution in that it allows path-sensitive information to be propagated beyond merge points.

6.2 Intraprocedural Analysis

A procedure is represented by the standard control flow graph representation where each node represents a statement or predicate in the procedure and the edges represent the direction of flow. To represent the variables in the procedure we adapt the following notation

- We create one object for each `new` statement in the procedure (using a 1-limited scheme). These objects are represented as O_i. We call O_i an "object variable".
- l_i represents a local primitive or reference variable.
- A field variable is represented as $l_i.f$, where l_i is a local reference variable and f is the field name. Since we analyze Java bytecode, all multiple-level reference

expressions of the form $l_i.f.g.h$ are split into a sequence of two level references of the form $l_i.f$. For example, a statement of the form $l_1.f.g.h = l_2$ would be transformed into the sequence of statements $l_3 = l_1.f$; $l_4 = l_3.g$; $l_4.h = l_2$.

As mentioned earlier, the aim of the analysis is to convert each local field variable, $l_i.f$ to the form $O_i.f$. We call the variables in the form $O_i.f$ "object field variables".

- Static field variables are represented as $C.f$, where C is the class to which the static field f belongs.
- Arrays are treated monolithically as a single object with the special field $\langle el \rangle$, denoting access to an element of the array.
- In Java bytecode, the parameters of a procedure are placed in the first n local variables of the procedure. This is modeled by a set of assignments $l_i = \mathcal{F}_i$, where l_i is a local variable of appropriate type that holds the i^{th} parameter and $\mathcal{F}_i$ is a special variable that models the formal parameter of the procedure. Typically, l_0 is assigned the container object, `this`. During interprocedural analysis the $\mathcal{F}_i$ variables are mapped to the actual parameters being passed at each call site.

We first explain the pointer and data dependence analysis as applied to local variables and later show how to handle parameters and static variables.

6.2.1 Pointer Analysis

With each node, S_i, of the CFG we associate the following sets: $S_i.USE$ is the set of variables used by the node and $S_i.DEF$ is the set of variables defined by the node. For each variable $d_j \in S_i.DEF$ we associate the set $d_j.PTS$ which is the set of object variables that the variable d_j points to. Initially, the PTS set is empty for each definition. The initial values for $S_i.DEF$ and $S_i.USE$ are given in Table 6.1.

The basic idea behind our pointer analysis algorithm is as follows: each local variable l_i which is assigned an object variable (O_i) is said to point to that object variable. If l_i is further assigned to another variable l_j then l_j is also set to point to the same object variables that l_i points to. If the variable l_i points to O_i, then the variable $l_i.f$ can be converted into the form $O_i.f$. If $O_i.f$ is also a reference variable then by tracing the assignments to $O_i.f$ we can determine what $O_i.f$ points to.

```
input: The CFG of a procedure
initialization: at each node n_i of the CFG
    initialize n_i.DEF, n_i.USE sets as given in table 6.1
    initialize d_j.PTS = ∅ for each d_j ∈ n_i.DEF

change = 1
while change == 1 do
    change = 0
    for each node n_i in topological order do
        for each variable v_j ∈ n_i.DEF do
            if v_j is a field variable l_j.f then
                n_i.DEF = n_i.DEF ∪ RdPoint(l_j, n_i).f
            /* Ignore all other variables. */
        endfor

        ψ = {}
        for each variable v_j ∈ n_i.USE do
            if v_j ∈ V_f and v_j is a field variable l_j.f then
                n_i.USE = n_i.USE ∪ RdPoint(l_j, n_i).f
            if v_j ∈ V_o and v_j is an object variable O_x then
                ψ = {O_x}
            if v_j ∈ V_l ∪ V_s ∪ V_of then
                ψ = RdPoint(v_j, n_i)
        endfor

        if n_i is an assignment v_k = v_j, then
            for each d_k ∈ n_i.DEF and d_k ∈ V_l ∪ V_s ∪ V_of do
                d_k.PTS = d_k.PTS ∪ ψ.
        if any DEF, USE or PTS set has changed then
            change = 1
    endfor

endwhile
```

Figure 6.1: Intraprocedural analysis

Table 6.1: Initial Values for USE and DEF

LHS/RHS	Type of variable	variable	DEF	USE
LHS	local/static	a	a	-
	field	$a.f$	$a.f$	a
RHS	local/static	a	-	a
	field	$a.f$	-	a, $a.f$
	new cl	-	-	O_1

The algorithm uses a function $RdPoint(v, n_i)$ for the variable v used at a node n_i as follows: $RdPoint(v, n_i)$ searches back in the CFG from n_i for the nodes with the most recent assignments to v. Let d_j be a definition of v at some node n_j that reaches the use of v at n_i. Then $RdPoint(v, n_i)$ includes all the objects in $d_j.PTS$ for every d_j that is a reaching definition of v. We use the notation $RdPoint(v, n_i).f$ to indicate the set of object field variables $O_i.f$ where $O_i \in RdPoint(v, n_i)$. We use the notation V_o, V_l, V_f, V_{of}, V_s and V_F to represent object variables, local variables, field variables, object field variables, static variables and formal variables respectively. The algorithm is shown in Figure 6.1[1]. Table 6.2 gives an example of pointer analysis.

Table 6.2: Pointer analysis

	D	U	DEF $\rightarrow$ PTS	USE
$S_1 : a =$ new cl	a	O_1	$a \rightarrow O_1$	O_1
$S_2 : a.f =$ new cl	$a.f$	O_2	$a.f, O_1.f \rightarrow O_2$	O_2
$S_3 : b = a$	b	a	$b \rightarrow O_1$	a
$S_4 : b.f =$ new cl	$b.f$	O_4	$b.f, O_1.f \rightarrow O_4$	O_4
$S_5 : x = a.f$	x	$a.f$	$x \rightarrow O_4$	$a.f, O_1.f$
	Before		After analysis	

S_1: O_1 represents the object created by `new cl`, $S_1.USE = \{O_1\}$. $S_1.DEF = \{a\}$ and the algorithm generates the information $a.PTS = \{O_1\}$ which is represented by the shorthand notation $a \rightarrow O_1$.

[1]in computing the topological order of the nodes in a procedure, we ignore all backedges

S_2: O_2 represents the object created by `new cl`, $S_2.DEF = \{a.f\}$. $RdPoint(a, S_2)$ returns O_1, since the definition of a at S_1 reaches S_2 and at S_1, $a.PTS = \{O_1\}$. Therefore $RdPoint(a, S_2).f$ is $O_1.f$ and we get $S_2.DEF = \{a.f, O_1.f\}$. Further, since $S_2.USE = \{O_2\}$ we get $(O_1.f).PTS = \{O_2\}$ (*i.e.*, $O_1.f \rightarrow O_2$).

S_3: $RdPoint(a, S_3)$ returns O_1. Hence $b.PTS = \{O_1\}$.

S_4: $RdPoint(b, S_4)$ returns O_1, since the definition of b at S_3 reaches S_4 and at S_3 $b.PTS = \{O_1\}$. Therefore, we get $S_4.DEF = \{b.f, O_1.f\}$. Further, the algorithm computes that $(O_1.f).PTS = \{O_4\}$ (*i.e.*, $O_1.f \rightarrow O_4$).

S_5: $RdPoint(a, S_5)$ returns O_1 and $RdPoint(a, S_5).f$ computes to $O_1.f$. Therefore, $S_5.USE = \{a.f, O_1.f\}$. Further, $RdPoint(O_1.f, S_5)$ returns O_4, since the definition of $O_1.f$ at S_4 reaches S_5. Hence at S_5 we get $x.PTS = \{O_4\}$ (*i.e.*, $x \rightarrow O_4$).

6.2.2 The Static Single Assignment Form

The simplest way to implement $RdPoint(v, n_i)$ is to traverse the CFG backwards until we find each node with the appropriate definition. This is obviously very inefficient. Instead, we use an adaptation of the method proposed by Chase *et al* [21] and also used by Wilson and Lam [90].

We start by building the DJ graph [83] of the procedure. The DJ graph of a procedure is the dominator tree [3] augmented with "join edges" or J-edges. An edge $S_x \rightarrow S_y$ in the CFG is named a J-edge if S_x does not strictly dominate S_y. Further, S_y is termed a *merge* node. The DJ graph is linear in the size of the CFG and can be constructed in linear time.

Using the DJ graph, we generate ϕ-nodes [31] for local variables only (not field variables). As explained in Section 2.4, a ϕ-node is a *merge* node where we place ϕ-*functions* in order to merge the values of a variable from distinct incoming control flow paths. A ϕ-node for a variable v has $DEF = \{v\}$ and $USE = \{v, v, \ldots\}$, where there are as many USE elements as there are incoming edges at the merge node. We do not rename the variables but in a single pass we generate the SSA def-use edges for the local variables. The ϕ-nodes for the field variables have to be generated dynamically as and when they are converted into the corresponding "object field variable".

procedure RdPoint (Use u_i, Node n_i)
begin
    if $u_i.ssapred \neq null$ **and** $u_i.ssatype \neq$ **changeable**
        return $u_i.ssapred.PTS$
    if n_i is a *merge* node **then**
        $ssatype =$ **changeable**
    else $ssatype =$ **fixed**
    $n_i = n_i.dom$
    while $(n_i \neq null)$ **do**
        for each $d_j \in n_i.DEF$ **do**
            if $d_j.symbol == u_i.symbol$ **then**
                CreateDefUseEdge($d_j, u_i, ssatype$)
                return $d_j.PTS$
        for each $u_j \in n_i.USE$ **do**
            if $u_j.symbol == u_i.symbol$ **then**
                CreateDefUseEdge($u_j.ssapred, u_i, u_j.ssatype$)
                return $u_j.ssapred.PTS$
        if n_i is a *merge* node **then**
            $ssatype =$ **changeable**
        $n_i = n_i.dom$
    endwhile

    /* no definition found */
    create a new assignment $d_i = 0$ or $d_i = null$
    CreateDefUseEdge($d_i, u_i, ssatype$)
    return $d_i.PTS$
end

Figure 6.2: Algorithm for RdPoint()

During a single postorder traversal of the dominator tree, we cache the dominance frontier of each node. When an assignment to an "object field variable", $O_i.f$, is discovered at a node S_j, we need to insert a ϕ-node for $O_i.f$ in each node belonging to the *iterated* dominance frontier of S_j. Starting with the nodes in the dominance frontier cached at S_j, we recursively determine the other nodes in time proportional to the size of the iterated dominance frontier[2].

To determine $RdPoint(v, S_i)$ for a local variable, we follow backwards one SSA def-use edge. For other variables, to determine $RdPoint(v, S_i)$ we walk back up the dominator tree (the D-edges of the DJ graph) until we come to an assignment of v at some node S_j. If we reach the procedure entry without finding an assignment we generate a warning (as this might imply a null pointer assignment) and create a dummy initialization assignment.

Having found an assignment of v at S_j, we draw a def-use edge from the assignment at S_j to the use at S_i. As the pointer analysis is applied iteratively, in subsequent passes, a look-up can follow the def-use edge backwards without having to traverse the dominator tree.

However, if the traversal up the dominator tree crosses a *merge* node, S_b, then we mark the def-use edge as `changeable`. A subsequent call to $RdPoint(v, S_i)$ will once again traverse the dominator tree in case a new ϕ-node has been inserted into S_b. If a new ϕ-node for v has been inserted into S_b, then the previous def-use edge is deleted and a new edge is drawn from S_b to S_i. This is illustrated in the example shown in Figure 6.3.

Figure 6.3(a) shows the control flow graph of a program section. The corresponding DJ graph and the initial SSA def-use edges for the local variable a are shown in Figure 6.3(b). In the interests of clarity, these SSA def-use edges are not shown in the subsequent figures. In Figure 6.3(c), at S_1, a points to O_1. At S_2, We use the SSA def-use edge for a to determine that a points to O_1 and generate the object field variable $O_1.i$. Similarly, at S_3, S_4 and S_5, we convert all uses or definitions of $a.i$ to corresponding uses or definitions of $O_1.i$. At S_3, there is a use of $O_1.i$ and so the algorithm searches up the dominator tree till it comes to the definition at S_2. Since the traversal includes S_3 which is a *merge* node, we mark the edge as `changeable`. At the end of the first iteration the algorithm discovers a definition of $O_1.i$ at S_4 and creates a ϕ-node at S_3 which is in the iterated dominance frontier of S_4. In the

[2]This is similar to answering the question cd(e) in the Roman Chariots Problem [71, 38]

second iteration all the `changeable` def-use edges from S_2 are replaced by `fixed` def-use edges from the ϕ-node.

One optimization can be applied to speed up the traverse up the dominator tree. When traversing up the dominator tree, we first check if $v \in S_j.DEF$. If yes, then we generate the def-use edge from S_j to S_i. If not, we check if $v \in S_j.USE$. If not, we continue up the dominator tree. If yes, then we copy the def-use edge from v in $S_j.USE$ to S_i. We do not need to traverse further up the dominator tree. If the def-use edge is marked `changeable`, the new edge is also marked `changeable`. For example, in Figure 6.3(c), when traversing up the dominator tree from S_5, we find a *use* of $O_1.i$ at S_3 and hence we need not traverse further up the tree till the definition at S_2. The source of the def-use edge to $O_1.i$ at S_3 is S_2 and the edge is marked `changeable`. Hence we make a def-use edge from the S_2 to S_5 and mark it also `changeable`. A similar optimization is applied in the second iteration also, in Figure 6.3(d). This way, in each iteration the maximum nodes visited for a particular variable v is equal to the number of nodes in the dominator tree.

The algorithm is given in Figure 6.2. The algorithm uses the notation $u_i.symbol$ and $d_i.symbol$ to refer to the variable associated with a particular use u_i or def d_i. $u_i.ssapred$ refers to the static single assignment that reaches the use u_i and $n_i.dom$ is the immediate dominator of the node n_i. CreateDefUseEdge() generates the SSA def-use edge of appropriate type (`changeable` or `fixed`).

Lazy generation of object field variables Note that when we generate an object variable O_i at an object creation site, we do not generate all the corresponding fields of the object. Only when a field of the object is defined or used, do we create the corresponding object field variable.

6.2.3 Path-sensitives Merges

In Figure 6.4, due to the merge, $S_{15}.DEF = \{O_1.g, O_2.g\}$; and both $O_1.g$ and $O_2.g$ point to O_3 and O_4. That is, $O_1.g \rightarrow \{O_3, O_4\}$ and $O_2.g \rightarrow \{O_3, O_4\}$. We have lost the path-sensitive information that *either* $O_1.g$ points to O_3 or $O_2.g$ points to O_4. As a result, at S_{16}, t points to either O_3, O_4 or O_5. At S_{17}, we get a spurious definition of $O_4.f$ resulting in a spurious data dependence from S_{17} to S_{18}.

It is possible to remove this imprecision by matching the points-to object to the corresponding definition as follows: for every definition $a.f = v$, where $a.f$ and v

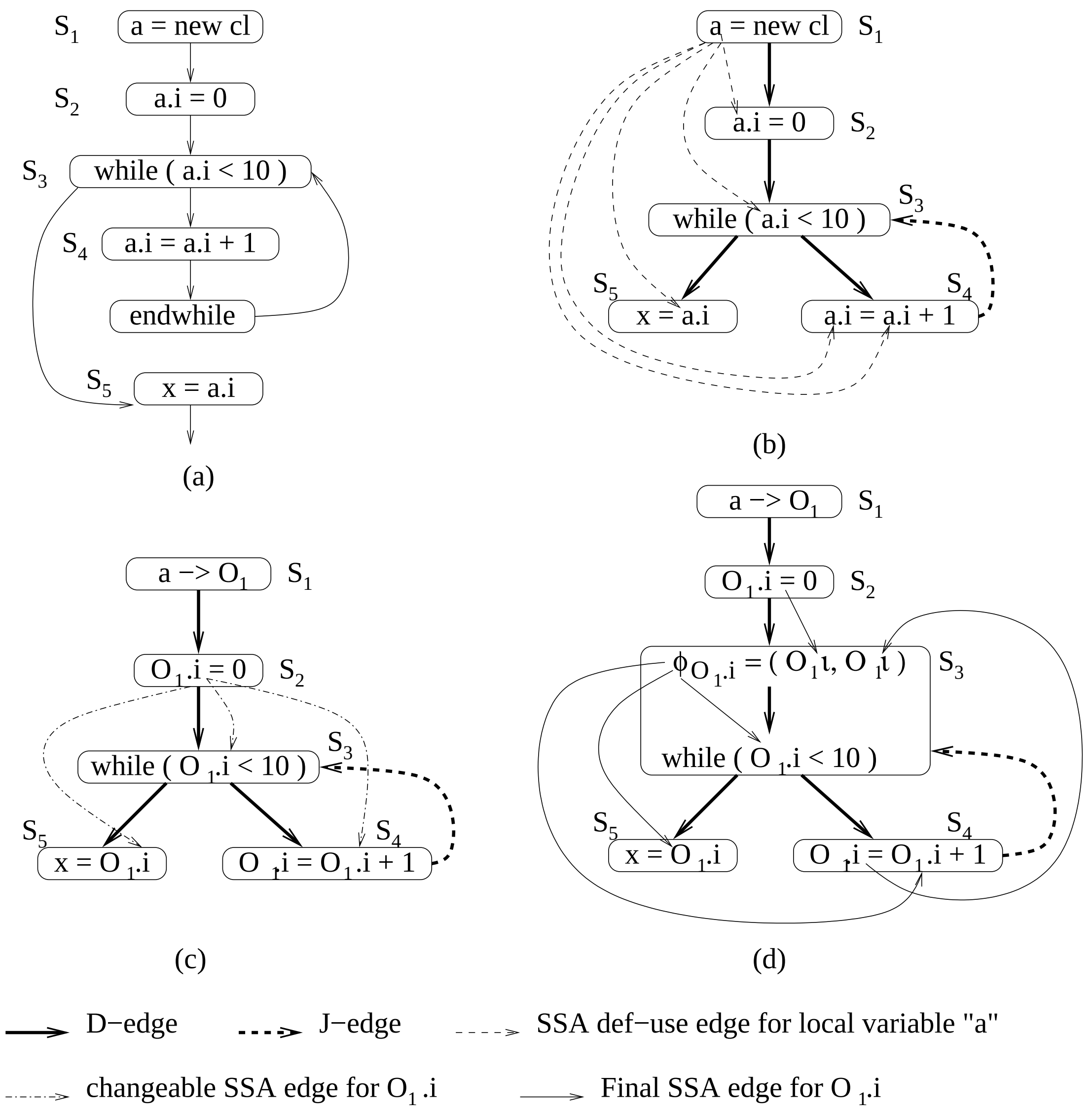


Figure 6.3: (a) The control flow graph, (b) The DJ graph with SSA edges for the local variable a, (c) First iteration with the changeable SSA edges for $O_1.i$, (d) The final SSA form with ϕ-node for $O_1.i$

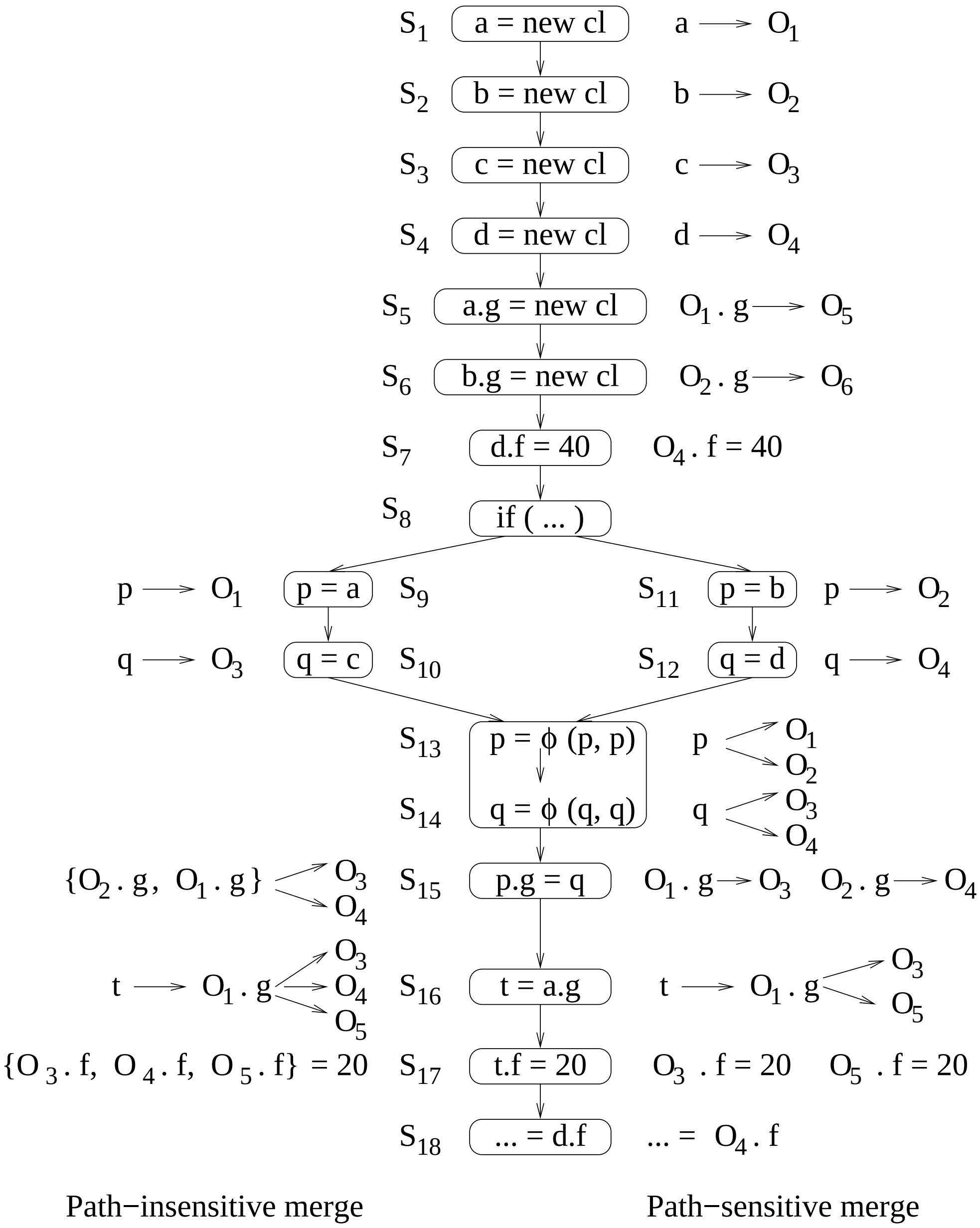


Figure 6.4: Path sensitive merges - The Control Flow Graph

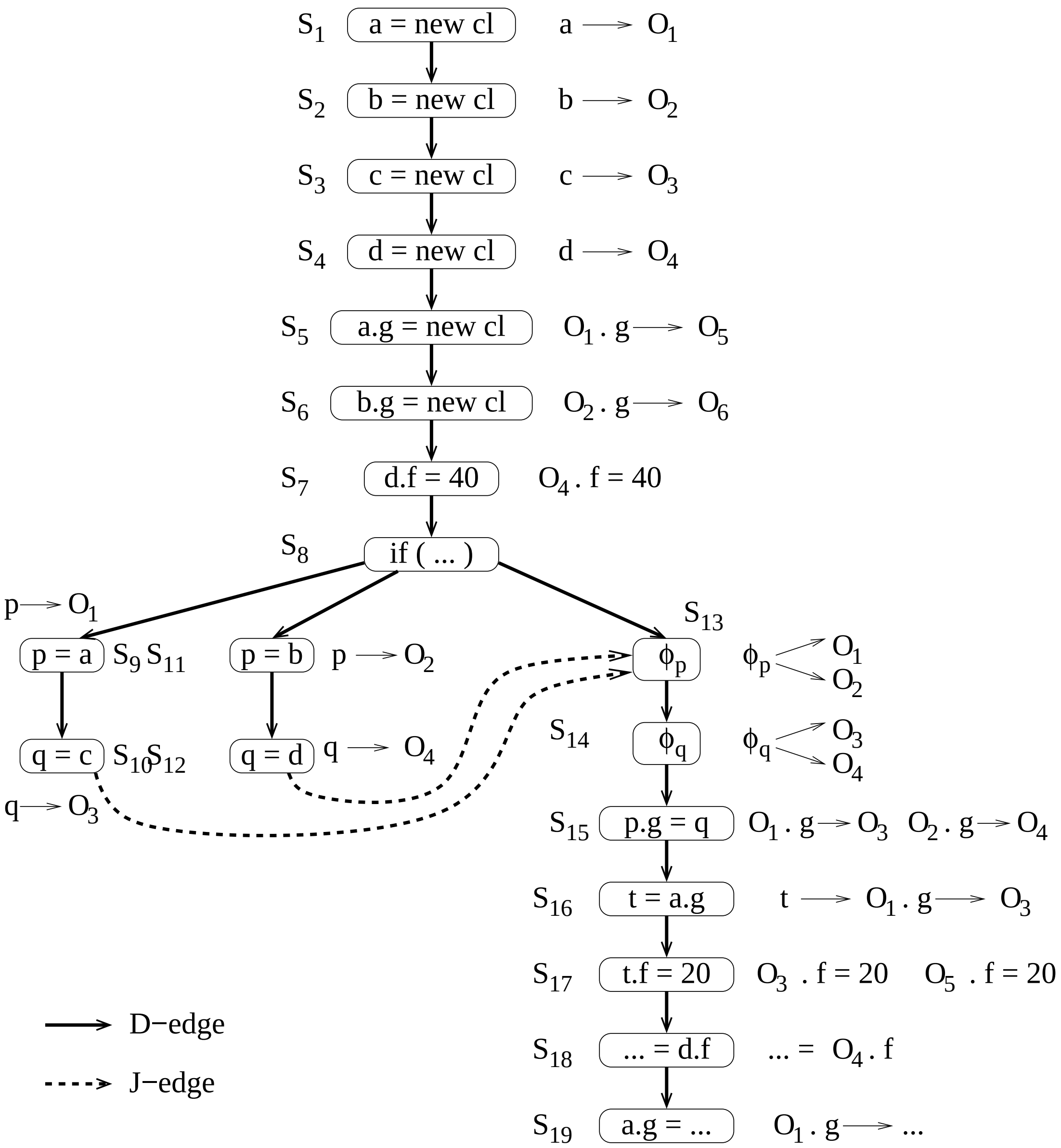


Figure 6.5: Path sensitive merges - The corresponding DJ graph

could be any reference variables, we check if the reaching definition of a and v are both ϕ *nodes belonging to the same merge node*. If they are, then we match the corresponding reaching definitions of $a.f$ with those of v. If not, all the definitions of $a.f$ are mapped to all the definitions of v.

This is illustrated graphically in Figure 6.4 and Figure 6.5. At S_{15} the reaching definition of p is S_{13} which is a ϕ-node for p and the reaching definition of q is S_{14} which is a ϕ-node for q belonging to the same merge point as S_{13}. Hence the algorithm generates the path-sensitive information that $O_1.g$ points to O_3 and $O_2.g$ points to O_4. Subsequently, at S_{15}, t is assigned to $a.g$ which resolves to $O_3.g$ which now points to either O_3 or O_5, but not O_4. Then, at S_{17}, we are able to determine that only $O_3.f$ or $O_5.f$ are assigned and hence there is no data dependence from S_{17} to the use of $O_4.f$ at S_{18}. With path-sensitive information at the merge, the spurious data dependence does not arise.

6.2.4 Strong Updates

A strong update is one in which the definition of a variable v at a program point *kills* all other definitions of v. In the context of our analysis framework, this implies that when v is used at S_i, the traversal up the dominator tree stops at the first definition of v at some node S_j and generates a def-use edge from S_j to S_i. Conversely, a weak update does not kill the other definitions. In the context of our analysis, this implies that when traversing up the dominator tree, even after finding a definition of v, at S_j, we keep traversing up the tree till we locate all the definitions of v at S_j^1, S_j^2 .... The traversal stops when it encounters a node where it can apply a strong update. In general, we can perform strong updates at all nodes except those nodes where more than one variable of the form $O.f$ is defined.

Weak Updates

In Figure 6.4, at S_{16} there is a use of $O_1.g$. At S_{15} there is a definition of $O_1.g$. However, the definition of $O_1.g$ at S_{15} should not kill the definition of $O_1.g$ at S_5 as the definition of $O_1.g$ at S_{15} is generated only along one branch of the `if` statement. The traversal continues beyond S_{15} and stops when it encounters the definition of $O_1.g$ at S_5. We require a def-use edge from both S_5 and S_{15} to S_{16}. This, however, violates the "single assignment" property of the SSA form.

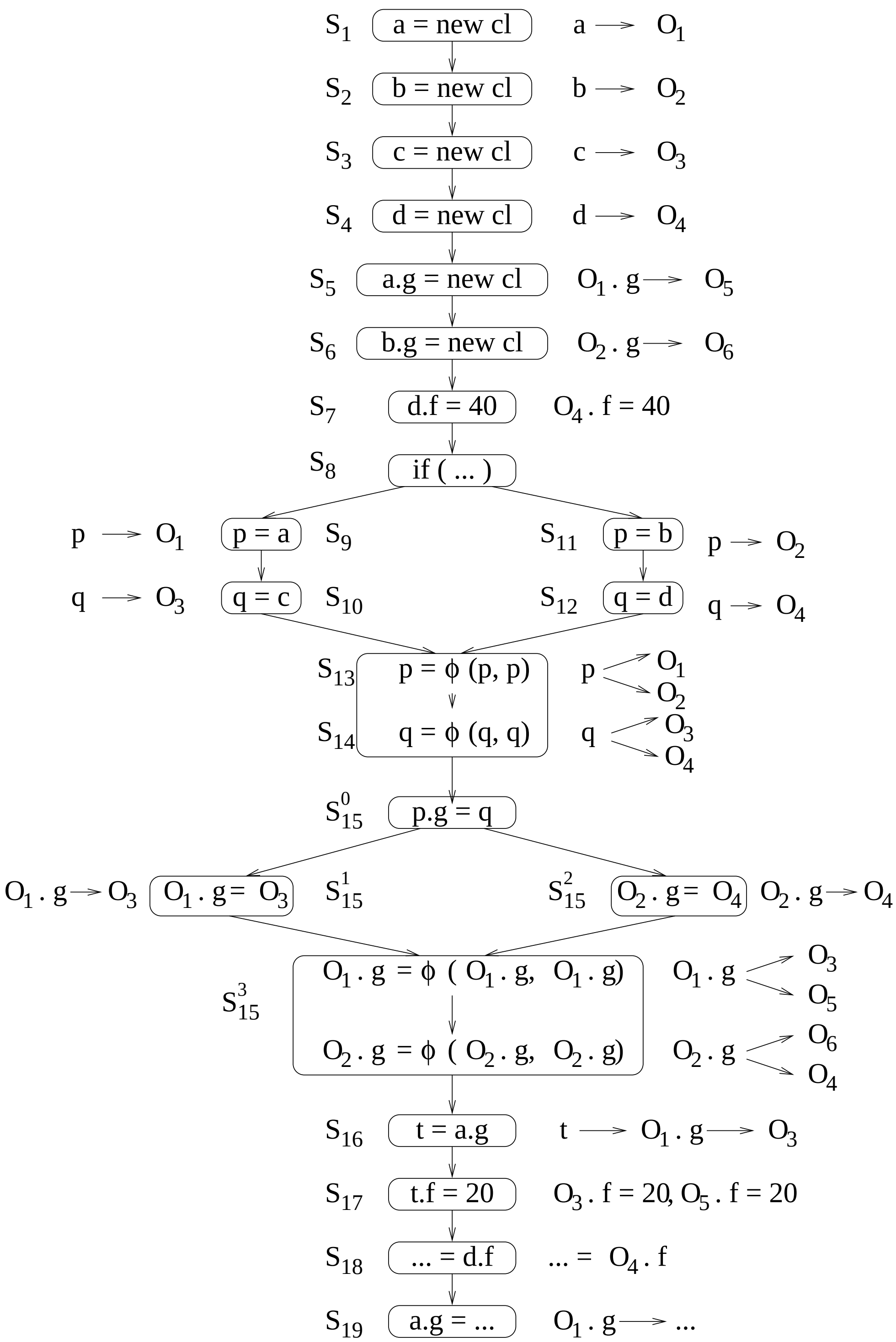


Figure 6.6: Generating special ϕ-nodes - The modified CFG

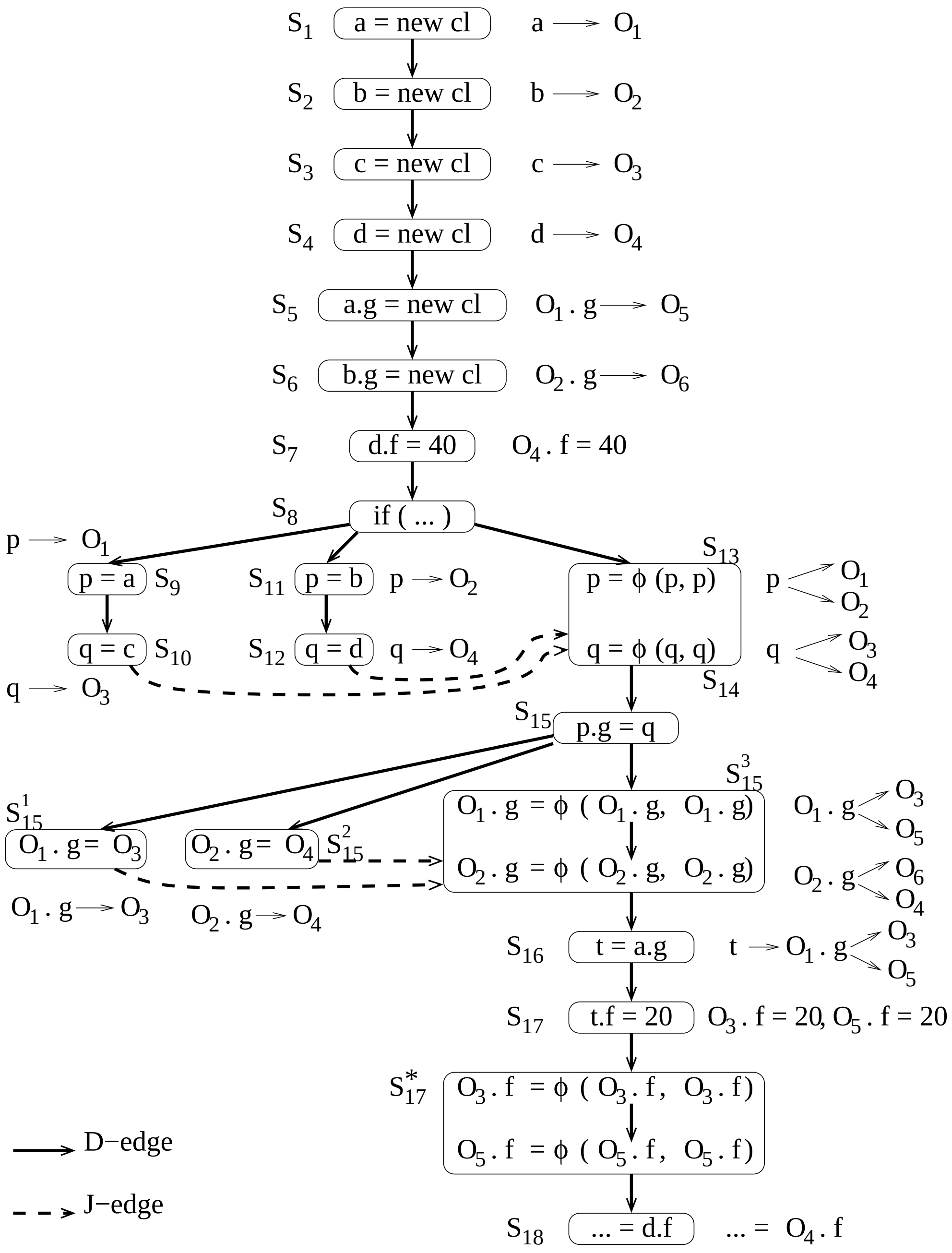


Figure 6.7: Generating special ϕ-nodes - The modified DJ graph

Two different definitions of $O_1.g$ reach S_{16}. Clearly, there is a need for a ϕ-node after S_{15} and before S_{16}. However, no standard ϕ-nodes placement algorithm can determine this since S_{16} is not a merge node at all. So we have the choice of either modifying the ϕ placement algorithm or modifying the graph so that we can use a standard algorithm. We choose the second option. The modifications to the CFG are explained with the help of the example in Figure 6.4. Since two different variables are being defined at S_{15}, we convert the node into a branch node with one definition at each branch as shown in Figure 6.6. We generate nodes S_{15}^1 to S_{15}^n for the n definitions and a *merge* node S_{15}^{n+1} where the ϕ-functions can be placed.

Since we are using the DJ graph method of generating ϕ-nodes, we need to update the DJ graph also. The corresponding changes in the DJ graph are achieved in an incremental manner as follows: S_{15} dominates all the branches (S_{15}^1 and S_{15}^2) and the merge node (S_{15}^3). There is a J-edge from each branch to the merge node. The DJ successors of S_{15} become the the DJ successors of S_{15}^3. This is depicted in Figure 6.7, where two branches S_{15}^1 and S_{15}^2 have been generated at S_{15}. The merge node is S_{15}^3 and has the ϕ-functions. This is, however, only the conceptual model. In practice, we do not actually extend the CFG or the DJ graph. We only create special ϕ-functions and generate the SSA def-use edges according to the conceptual model. At S_{17}^*, for example, we create special ϕ-functions for $O_3.f$ and $O_5.f$. For the first use at $S_{17}*$, *RdPoint* will start the traversal at S_{17} and for the second use, *RdPoint* will start the traversal from the immediate dominator of S_{17}.

Then each use has exactly one reaching definition.

In general, when n ($n > 1$) object fields, $O_1.f$, $O_2.f$, ..., $O_n.f$ are defined at the same node, S_x, we create n special ϕ-functions S_1, S_2, ..., S_n such that at each S_i there are two uses of $O_i.f$. The reaching definition for the first use is the definition of $O_i.f$ at S_x and the reaching definition of the second use is determined by applying $RdPoint(O_i.f, pred(S_x))$, where $pred(S_x)$ is the predecessor of S_x in the CFG.

Arrays

For arrays, we cannot perform strong updates since we are treating them as monolithic objects with the distinguished element $\langle el \rangle$. In order to simplify the handling of arrays, we treat all definitions of an array element $A[i]$ as both a use and a definition of the variable $A.\langle el \rangle$. This is equivalent to inserting special ϕ-nodes for $A.\langle el \rangle$. Then for array elements also we have exactly one reaching definition for each use.

Table 6.3: EscapeIn and EscapeOut nodes

Node	field type	DEF $\rightarrow$ PTS	USE
E_{in}	Reference	$\mathcal{F}_i^{\psi}$.f $\rightarrow \mathcal{F}_{new}^{\psi}$	$\mathcal{F}_{new}$
E_{in}	Primitive	$\mathcal{F}_i^{\psi}.f$	
E_{out}	Reference	$\mathcal{F}_i^{\psi}.f$	$\mathcal{F}_i^{\psi}.f$
E_{out}	Primitive		$\mathcal{F}_i^{\psi}.f$

6.2.5 Parameter Variables

At the procedure entry we generate a $FormalIn$ node for each parameter passed to the procedure. Each $FormalIn$ node is of the form $l_i = \mathcal{F}_i$ (*i.e.*, $DEF = \{l_i\}$ and $USE = \{\mathcal{F}_i\}$), where l_i is the i^{th} local variable and $\mathcal{F}_i$ is the i^{th} formal parameter. Typically, $l_0 = \mathcal{F}_0$ represents the implicit `this` reference parameter.

If $\mathcal{F}_i$ is a reference parameter, it is treated as a special object variable similar to the O_i objects discussed so far. For example, for a $FormalIn$ node $l_i = \mathcal{F}_i$, our pointer analysis algorithm would generate $l_i.PTS = \{\mathcal{F}_i\}$. Field variables $\mathcal{F}_i.f$ are generated and treated in the same way as the "object field variables" $O_i.f$.

`return` statements are treated as an assignment to a special variable $\mathcal{F}_r$. That is, a statement `return x;` would be treated as $\mathcal{F}_r =$ `x`. We create a $FormalOut$ node for $\mathcal{F}_r$ with $USE = \{\mathcal{F}_r\}$. The data dependence analysis automatically generates an SSA def-use edge from each `return` statement (where $\mathcal{F}_r$ is defined) to the $FormalOut$ node (where $\mathcal{F}_r$ is used). This "merges" all the return values.

Escape Analysis

A variable of the form $\mathcal{F}_i.f.g.h$ is "visible" outside the procedure through the parameter variable $\mathcal{F}_i$. In the bytecode, a statement $\mathcal{F}_2.f.g.h = 10$ is represented by $l_1 = \mathcal{F}_2.f$; $l_2 = l_1.g$; $l_2.h = 10$. We need some form of reachability analysis [25] to determine that l_1 and l_2 escape. We say that l_1 and l_2 escape *through* $\mathcal{F}_2$. We call $\mathcal{F}_2$ the *escape gateway*. The gateways are a property of each DEF, USE and PTS element at each node and is represented by ψ. That is, if $d \in n_i.DEF$ and $d.\psi = \{\mathcal{F}_1\}$, them we say that the definition d escapes through $\mathcal{F}_1$.

We need to create an $EscapeOut$ node (similar to the $FormalOut$ nodes) for each escape variable that is defined. We need to create an $EscapeIn$ node (similar to the $FormalIn$ nodes) for each escape variable that is used before it is defined.

A variable that is visible outside its procedure is marked $ArgEscape$ and a variable that is visible outside its thread is marked $GlobalEscape$. The rules for escape analysis are as follows:

- A reference $FormalIn$ variable escapes the method. At every $FormalIn$ node, $l_i = \mathcal{F}_i$, if l_i is a reference variable, we set $l_i \rightarrow \mathcal{F}_i^{\mathcal{F}_i}$ where the superscript indicates that the object variable $\mathcal{F}_i$ escapes through the gateway $\mathcal{F}_i$. We mark $\mathcal{F}_i^{\mathcal{F}_i}$ as $ArgEscape$.

- If an object variable is assigned to an escape variable, then the object variable escapes through the same gateways as the assigning variable. For example, if we have the assignment $\mathcal{F}_1^{\mathcal{F}_1}.f = v$ and if v points to O_2, then O_2 escapes with $\mathcal{F}_1$ and is represented as $O_2^{\mathcal{F}_1}$. It is marked $ArgEscape$.

- If n_i is a `return` statement $\mathcal{F}_r = v$, and at the reaching definition of v, we have $v \rightarrow O_i$ then O_i is marked $ArgEscape$ and is represented as $O_i^{\mathcal{F}_r}$.

- For each definition $\mathcal{F}^{\psi}.f$ or $O_i^{\psi}.f$ $(\psi \neq \emptyset)$ we create an $EscapeOut$ node (similar to the $FormalOut$ node).

- For each use $\mathcal{F}^{\psi}.f$ $(\psi \neq \emptyset)$ the pointer analysis searches up the dominator tree for a corresponding definition. If there is none, we create an $EscapeIn$ node. A variable $O_i^{\psi}.f$ that has no reaching definition within the procedure also escapes. However, since O_i was clearly created within the procedure, the field $O_i^{\psi}.f$ could not have been assigned in the calling procedure but, it could have been assigned in a parallel thread. So we create an $EscapeIn$ node for it and mark $O_i^{\psi}.f$ as $GlobalEscape$.

The details of the DEF, USE and PTS sets of the $EscapeIn$ and $EscapeOut$ nodes for an arbitrary variable $\mathcal{F}_i.f$ are given in Table 6.3. $\mathcal{F}_{new}^{\psi}$ is a new formal variable that escapes through the same set ψ and represents the actual object created outside the procedure.

Examples In the Figure 6.8 we give the code and pseudo bytecode for a simple Producer-Consumer example which will be used as a running example in this chapter and the next chapter. The DJ graph for the example is broken over the three figures Figure 6.10, Figure 6.11 and Figure 6.12[3].

Example for EscapeIn In Figure 6.10 in the method *Consumer.run*, at t_{10} there is a use of $\mathcal{F}_0.buff$. A traversal up the DJ graph determines that there is no reaching definition of $\mathcal{F}_0.buff$ in the procedure. Therefore we create an $EscapeIn$ node at t_3. Since $\mathcal{F}_0.buff$ is a reference variable, we generate a new formal variable and name it $\mathcal{F}_1$. At t_3, we get $\mathcal{F}_0.buff = \mathcal{F}_1$ and the pointer analysis will generate the information that $\mathcal{F}_0.buff$ *points to* $\mathcal{F}_1$ (*i.e.*, $\mathcal{F}_0.buff \rightarrow \mathcal{F}_1$).

At t_{12}, the use of $\mathcal{F}_1.seq$ generates an $EscapeIn$ node at t_4. A new formal variable is not created as $\mathcal{F}_1.seq$ is not a reference variable. (Note: we have not shown the "escape gateways" in the figure).

Example for EscapeOut In Figure 6.12 at x_5 in $Producer.\langle init\rangle$, $\mathcal{F}_0.buff$ is defined. The corresponding $EscapeOut$ node created at x_7 is $\mathcal{F}_0.buff = \mathcal{F}_0.buff$. The DEF element is created only as a place holder for the points to information. The pointer analysis generates the further information $\mathcal{F}_0.buff \rightarrow \mathcal{F}_1$.

The definition of $\mathcal{F}_0.id$ at x_6 generates an $EscapeOut$ node "$= \mathcal{F}_0.id$" at x_8. x_8 has an empty DEF set as $\mathcal{F}_0.id$ is not a reference variable and does not need to store points-to information.

♣ Although we generate $EscapeIn$ nodes only for variables that are used within the procedure, our SSA based algorithm will automatically also generate an $EscapeIn$ node for an escape variable that is *defined* along one path but not another. This ensures that the value passed to the procedure reaches the end of the procedure if it is not killed along all the paths to the procedure exit.

6.2.6 Static Variables

Static variables are treated just like parameter variables. An assignment $cl.f = O_i$ makes O_i escape with the gateway $cl.f$, O_i is marked $GlobalEscape$ and is represented as $O_i^{cl.f}$. $EscapeIn$ nodes are generated for static variables that are used before being defined within the procedure and $EscapeOut$ nodes are generated for every static variable that is defined within the procedure.

[3] In the interests of readability we have omitted the SSA def-use edges in Figures 6.10, 6.11 and 6.12

```
class Producer extends Thread {
      private Buffer buff;
      private int id;

      public Producer ( Buffer b, int id ) {
            buff = b;                                     l0.buff = l1
            this.id = id;                                 l0.id = l2
      }

      public void run() {
            int i = 0;                                    l1 = 0
            while (i < 10) {                              while(l1 < 10)
                  buff.put(i);                            l2 = l0.buff; l2.put(l1)
                  i = i + 1;                              l1 = l1 + 1
            }
      }
}

class Consumer extends Thread {
      private Buffer buff;
      private int id;

      public Consumer ( Buffer b, int id ) {
            buff = b;                                     l0.buff = l1
            this.id = id;                                 l0.id = l2
      }

      public void run() {
            int i = 0;                                    l1 = 0
            int value = 0;                                l2 = 0
            while (i < 10) {                              while(l1 < 10)
                  value = buff.get();                     l3 = l0.buff; l2 = l3.get()
                  i = i + 1;                              l1 = l1 + 1
            }
      }
}
```

Figure 6.8: Producer-Consumer example

```
class Buffer {
    private int seq;
    private boolean avail = false;
    private int dc = 0;

    public synchronized int get() {
        while ( avail == false ) {                    while(l0.avail == false)
            wait();                                   l0.wait()
        }
        avail = false;                                l0.avail = false
        dc = dc - 1;                                  l0.dc = l0.dc - 1
        dc = dc - 1;                                  l0.dc = l0.dc - 1
        notify();                                     l0.notify()
        return seq;                                   returnl0.seq
    }

    public synchronized void put(int value) {
        while ( avail == true ) {                     while(l0.avail == true)
            wait();                                   l0.wait()
        }
        seq = value;                                  l0.seq = l1
        avail = true;                                 l0.avail = true
        dc = dc + 1;                                  l0.dc = l0.dc + 1
        dc = dc + 1;                                  l0.dc = l0.dc + 1
        notify();                                     l0.notify()
    }
}

class ProducerConsumerTest {
    public static void main(string[]args) {
        Buffer b = new Buffer();                      l1 = new Buffer
        Producer p1 = new Producer(b, 1);             l4 = new Producer
                                                      l4.⟨init⟩(l1, 1);
                                                      l2 = l4
        Consumer c1 = new Consumer(b, 1);             l5 = new Consumer
                                                      l5.⟨init⟩(l1, 1);
                                                      l3 = l5
        p1.start();                                   l2.start()
        c1.start();                                   l3.start()
    }
} /* Producer-Consumer example with pseudo bytecode */
```

Figure 6.9: Producer-Consumer example – continued

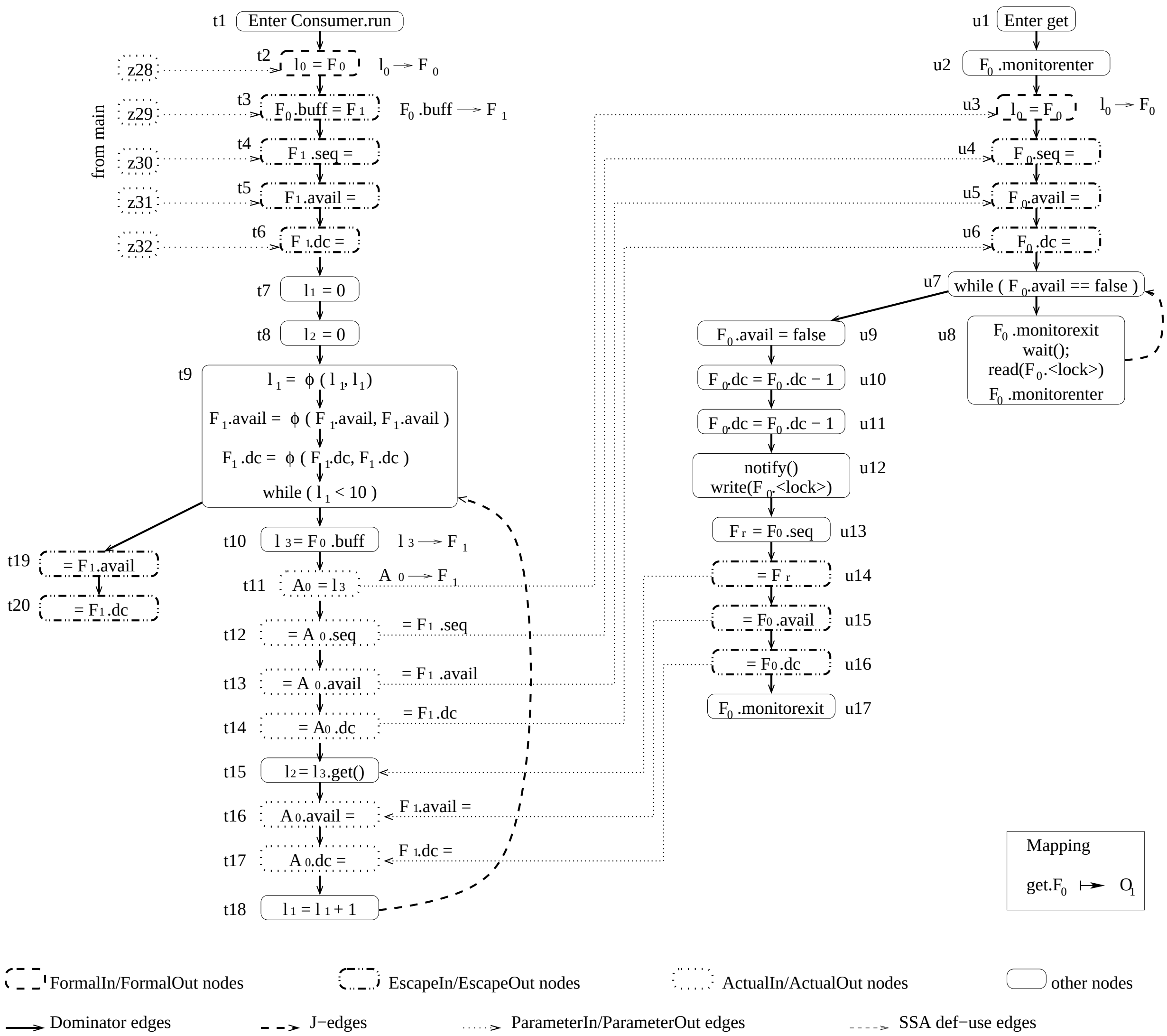


Figure 6.10: The SSA graph for $Consumer.run()$ and $get()$

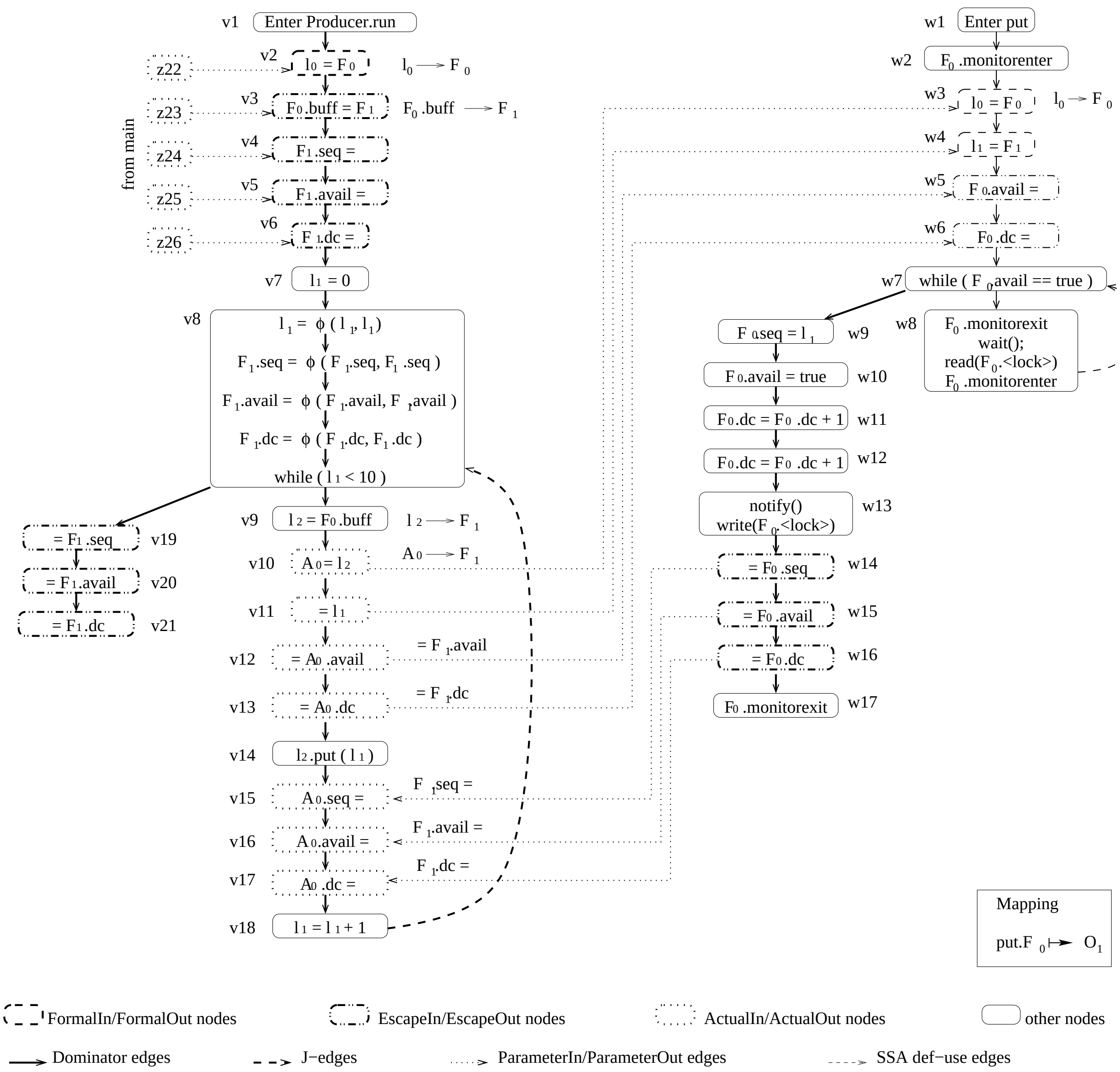


Figure 6.11: The SSA graph for *Producer.run*() and *put*()

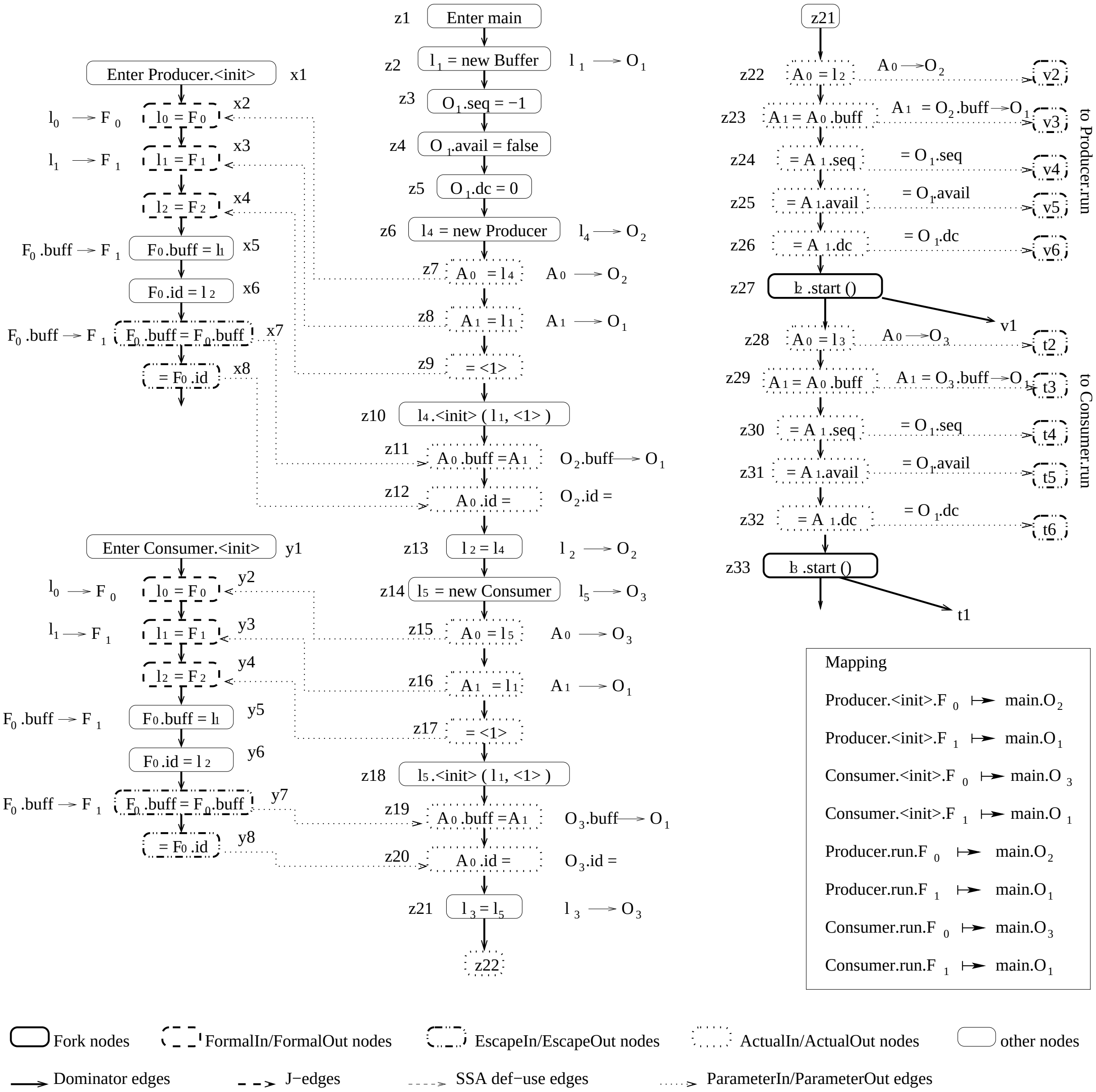


Figure 6.12: The SSA graph for *main*(), *Producer*.⟨*init*⟩ and *Consumer*.⟨*init*⟩

```
    static void printlist ( list head ) {
S1    while ( head != null ) {          while ( l0 ≠ null )
S2      println(head.value);            print l0.value
S3      head = head.next                l0 = l0.next
      }
    }
    Program code                        Pseudo-bytecode
```

Figure 6.13: A Recursive Procedure

6.2.7 Recursive Data Structures

In the absence of "hidden" [90] variables, recursive data structures are not a problem as each reference variable points to a finite set of objects created at a finite number of object creation sites. However, in the case where objects are created outside the current procedure, the algorithm may loop infinitely. Consider the simple procedure shown in Figure 6.13, which receives the head of a link list as a parameter and then prints out each item in the list.

Table 6.4 shows the DEF, USE and PTS sets generated by the algorithm. In the first iteration, at the $FormalIn$ node F^0_{in}, we get $l_0 \to \mathcal{F}^{\mathcal{F}_0}_0$. At S_3, $\mathcal{F}^{\mathcal{F}_0}_0.next$ is used. A corresponding $EscapeIn$ node is created at E^2_{in}. As $\mathcal{F}^{\mathcal{F}_0}_0.next$ is a reference variable a new $FormalIn$ variable, $\mathcal{F}_1$ is created at E^2_{in}. At the end of the iteration, at S_3, $l_0 \to \mathcal{F}^{\mathcal{F}_0}_1$. In the second iteration, there are two reaching definitions for l_0 at S_3 ($S_3.USE = \{\mathcal{F}^{\mathcal{F}_0}_0.next, \mathcal{F}^{\mathcal{F}_0}_1.next\}$). The reaching definition for $\mathcal{F}^{\mathcal{F}_0}_0.next$ is at E^2_{in} but $\mathcal{F}^{\mathcal{F}_0}_1.next$ has no reaching definition. Hence another $EscapeIn$ node, E^3_{in} is created.

In each subsequent iteration, a new $EscapeIn$ node would be created and the algorithm would loop for ever.

To solve this problem, we use a 1-limited scheme for dealing with recursive data structures. When the algorithm detects two uses or two definitions of the same field at the same program point such that both escape through the same gateway parameter, it merges the two variables. For example, at S_3, there is a use of $\mathcal{F}^{\mathcal{F}_0}_0.next$ and $\mathcal{F}^{\mathcal{F}_0}_1.next$. Both variables access the same field `next` and both escape through $\mathcal{F}_0$. At this point we discard $\mathcal{F}^{\mathcal{F}_0}_1.next$. The corresponding $EscapeIn$ nodes would not be generated and the algorithm will terminate.

Table 6.4: Recursive data structures

	DEF $\rightarrow$ PTS	USE
First iteration		
$F_{in}^0 : l_0 = \mathcal{F}_0$	$l_0 \rightarrow \mathcal{F}_0^{\mathcal{F}_0}$	$\mathcal{F}_0$
$E_{in}^1 : \mathcal{F}_0.value =$	$\mathcal{F}_0.value$	
$E_{in}^2 : \mathcal{F}_0.next = \mathcal{F}_1$	$\mathcal{F}_0.next \rightarrow \mathcal{F}_1^{\mathcal{F}_0}$	$\mathcal{F}_1$
S_0 : ϕ-node for l_0	l_0	l_0, l_0
S_1 : while $l_0 \neq$ null		$l_0, null$
S_2 : print $l_0.value$		$\mathcal{F}_0^{\mathcal{F}_0}.value$
$S_3 : l_0 = l_0.next$	$l_0 \rightarrow \mathcal{F}_1^{\mathcal{F}_0}$	$\mathcal{F}_0^{\mathcal{F}_0}.next$
Second iteration		
$F_{in}^0 : l_0 = \mathcal{F}_0$	$l_0 \rightarrow \mathcal{F}_0^{\mathcal{F}_0}$	$\mathcal{F}_0$
$E_{in}^1 : \mathcal{F}_0.value =$	$\mathcal{F}_0.value$	
$E_{in}^2 : \mathcal{F}_0.next = \mathcal{F}_1$	$\mathcal{F}_0.next \rightarrow \mathcal{F}_1^{\mathcal{F}_0}$	$\mathcal{F}_1$
$E_{in}^3 : \mathcal{F}_1.value =$	$\mathcal{F}_1.value$	
$E_{in}^4 : \mathcal{F}_1.next = \mathcal{F}_2$	$\mathcal{F}_1.next \rightarrow \mathcal{F}_2^{\mathcal{F}_0}$	$\mathcal{F}_2$
S_0 : ϕ-node for l_0	l_0	l_0, l_0
S_1 : while $l_0 \neq$ null		$l_0, null$
S_2 : print $l_0.value$		$\mathcal{F}_0^{\mathcal{F}_0}.value$, $\mathcal{F}_1^{\mathcal{F}_0}.value$
$S_3 : l_0 = l_0.next$	$l_0 \rightarrow \{\mathcal{F}_1^{\mathcal{F}_0}, \mathcal{F}_2^{\mathcal{F}_0}\}$	$\mathcal{F}_0^{\mathcal{F}_0}.next$, $\mathcal{F}_1^{\mathcal{F}_0}.next$

6.3 Interprocedural Analysis

The intraprocedural analysis of a procedure is summarized by the set of $FormalIn$, $EscapeIn$, $FormalOut$ and $EscapeOut$ nodes. At a procedure invocation site, we generate one $ActualIn$ node for each $FormalIn$ and $EscapeIn$ node and we generate one $ActualOut$ node for each $FormalOut$ and $EscapeOut$ node. This completely summarizes the procedure invoked. For each formal **reference** variable $\mathcal{F}_i$, we create an $ActualIn$ variable $\mathcal{A}_i$ which is treated during pointer analysis as a local temporary variable. The structure of the $ActualIn$ and $ActualOut$ nodes is shown in Table 6.5 and 6.6 where exp is the actual expression passed as a parameter to the procedure and v_r is the variable to which the return value is assigned. In the tradition of the *System Dependence Graph* [49], we generate a $parameter - in$ edge from the $ActualIn$ node to the corresponding $FormalIn$ or $EscapeIn$ node and a $parameter - out$ edge from each $FormalOut$ or $EscapeOut$ node to the corresponding $ActualOut$ node (the parameter-in and parameter-out edges are special kinds of interprocedural data dependence edges). The $ActualIn$ and $ActualOut$ nodes together completely summarize the called procedure. For programs without recursion, the inter-procedural analysis processes the procedures in reverse topological order on the call graph. For programs with recursion, the analysis iterates over each strongly connected component of the call graph.

Parameter aliasing If two actual parameters, represented by $\mathcal{F}_i$ and $\mathcal{F}_j$ are aliased at a particular call site, the corresponding $FormalIn$ nodes generated will be $l_i = \mathcal{F}_i$ and $l_j = \mathcal{F}_i$, if $i < j$. We need to check for aliasing contexts not only in the formal parameters but also in the variables that map to the $EscapeIn$ nodes. $EscapeIn$ nodes are discovered only after an initial analysis of the procedure. There are several possible schemes of varying precision to handle parameter aliasing. One possibility is to start with the assumption that none of the $EscapeIn$ parameters are aliased and then re-analyze the procedure if the context differs from this initial assumption. A summary of the procedure for each different aliasing context can be maintained and reused as appropriate [90]. Another, more conservative scheme is to assume that two parameters that are aliased at a call site are aliased at every call site. Then only one summary needs to be maintained for each procedure. A third scheme could be to determine all the relevant alias contexts [22].

Table 6.5: ActualIn nodes

	FormalIn	ActualIn
Reference	$l_i = \mathcal{F}_i$	$\mathcal{A}_i = exp$
Scalar	$l_i = \mathcal{F}_i$	$= exp$
	EscapeIn	ActualIn
Reference	$\mathcal{F}_i.f = \mathcal{F}_j$	$\mathcal{A}_j = \mathcal{A}_i.f$
Primitive	$\mathcal{F}_i.f =$	$= \mathcal{A}_i.f$
Static ref	$cl.f = \mathcal{F}_j$	$\mathcal{A}_j = cl.f$
Static prim	$cl.f =$	$= cl.f$

Table 6.6: ActualOut nodes

	FormalOut	ActualOut
Reference	$\mathcal{F}_r \rightarrow \{\mathcal{F}_j, O_k\} = \mathcal{F}_r$	$v_r = \{\mathcal{F}_j, O_k\}$
Primitive	$\mathcal{F}_r = \mathcal{F}_r$	$v_r =$
	EscapeOut	ActualOut
Reference	$O_i.f \rightarrow \{\mathcal{F}_j, O_k\}$	$O_i.f = \{\mathcal{A}_j, O_k\}$
Reference	$\mathcal{F}_i.f \rightarrow \{\mathcal{F}_j, O_k\}$	$\mathcal{A}_i.f = \{\mathcal{A}_j, O_k\}$
Primitive	$= O_i.f$	$O_i.f =$
Primitive	$= \mathcal{F}_i.f$	$\mathcal{F}_i.f =$
Static ref	$cl.f \rightarrow \{\mathcal{F}_j, O_k\}$	$cl.f = \{\mathcal{A}_j, O_k\}$
Static prim	$= cl.f$	$cl.f =$

Example Consider the call to $Producer.\langle init\rangle$ in `main` in Figure 6.12. For the $FormalIn$ node x_2, we create an $ActualIn$ node z_7 and for the formal variable $\mathcal{F}_0$ we create an $ActualIn$ variable $\mathcal{A}_0$. The corresponding actual parameter is l_4 and so we get $z_7 : \mathcal{A}_0 = l_4$. For the $FormalIn$ node $x_3 : l_1 = \mathcal{F}_1$, we create $z_8 : \mathcal{A}_1 = l_1$. For the $FormalIn$ node $x_4 : l_2 = \mathcal{F}_2$, we do not need to create an $ActualIn$ variable as $\mathcal{F}_2$ is not a reference variable. Hence we get the $ActualIn$ node z_9 : " $= l_1$", with an empty DEF set. There is no $FormalOut$ node in $Producer.\langle init\rangle$, but there are two $EscapeOut$ nodes. At the $EscapeOut$ node x_7, we have $\mathcal{F}_0.buff = \mathcal{F}_0.buff$ and $\mathcal{F}_0.buff \rightarrow \mathcal{F}_1$. The corresponding $ActualOut$ node is $z_{11} : \mathcal{A}_0.buff = \mathcal{A}_1$. For the $EscapeOut$ node "$= \mathcal{F}_0.id''$ we create the $ActualOut$ node "$\mathcal{A}_0.id =''$.

Once, these nodes are created, the pointer analysis continues as before. At z_7, the reaching definition for l_4 is z_6 and this generates the information $\mathcal{A}_0 \rightarrow O_2$. At z_8, we get $\mathcal{A}_1 \rightarrow O_1$. Now, at z_{11}, the reaching definition for $\mathcal{A}_0$ is at z_7 and the reaching definition for $\mathcal{A}_1$ is at z_8. Hence at z_{11} the pointer analysis determines that $O_2.buff \rightarrow O_1$, summarizing the information that $O_2.buff$ was assigned O_1 within the procedure. Similarly, at z_{12}, the pointer analysis determines that $O_2.id$ is defined, summarizing the fact that $O_2.id$ was assigned some value inside the procedure.

6.4 Complexity

We discuss the complexity in terms of the intra-procedural and inter-procedural analysis.

Intra-procedural Analysis

The DJ graph is linear in the size of the CFG and can be constructed in time linear in the size of the CFG [83]. The ϕ-nodes and the SSA def-use edges for local variables are created in time $O(V' * N)$, where V' is the number of local variables and N is the number of nodes in the CFG. On the assumption that the points-to information is sparse the space requirement of the SSA graph is $O(N)$ and in the worst case $O(N^2)$.

The dominance frontiers for each node are cached in linear time and at run time generating the ϕ-nodes for object field variables takes $O(N)$ time per variable. The running time of the algorithm is $O(x * V * N)$, where V is the number of variables and x is the number of iterations. If we analyze the nodes in topological order, the number of iterations tends to be very small.

Inter-procedural Analysis

For non-recursive procedures, each procedure is analyzed exactly once. However, depending on the context, the procedure may have to be re-analyzed for each call site if each call site has a different calling context. In the worst case, this results in complexity exponential on the depth of the call graph. Alternatively, it is possible to use a conservative estimate of aliasing at all the call sites.

6.5 Related Work

We discuss the related work in terms of intra-procedural and inter-procedural algorithms.

Intra-procedural Analysis

One of the key results of this chapter is to analyze heap allocated variables as precisely as possible and in a flow-sensitive manner. We compare our work only with other flow-sensitive analyses. Wilson and Lam [90] assume that locally allocated heap blocks are never unique and hence cannot apply strong updates in the presence of local heap allocated variables. Emami *et al* [36] analyze heap conservatively and Ghiya *et al* [42] use a form of connection analysis that determines if two heap allocated pointers point to the same heap data structure but does not distinguish the field from its object. Algorithms based on alias pairs [24, 48, 56, 21] also exist, but we have yet to investigate how they may be adapted to escape analysis or pointer analysis in the Java memory model.

The work by Choi *et al* [25] generates connection graphs at each program point in order to perform escape analysis. Their algorithm does not apply strong updates to field variables and hence gives conservative results when adapted to pointer analysis or data dependence analysis. Our scheme for naming heap objects is similar to theirs. Others use a *k-limiting* scheme which is useful for shape analysis [79, 21]. Figure 6.14(a) shows the connection graph of Choi *et al* and Figure 6.14(b) shows the points to information carried by our algorithm for the same code (the example was taken from their paper [25]). It is clear that the information maintained by our algorithm is far more compact than theirs. This affects not only the memory

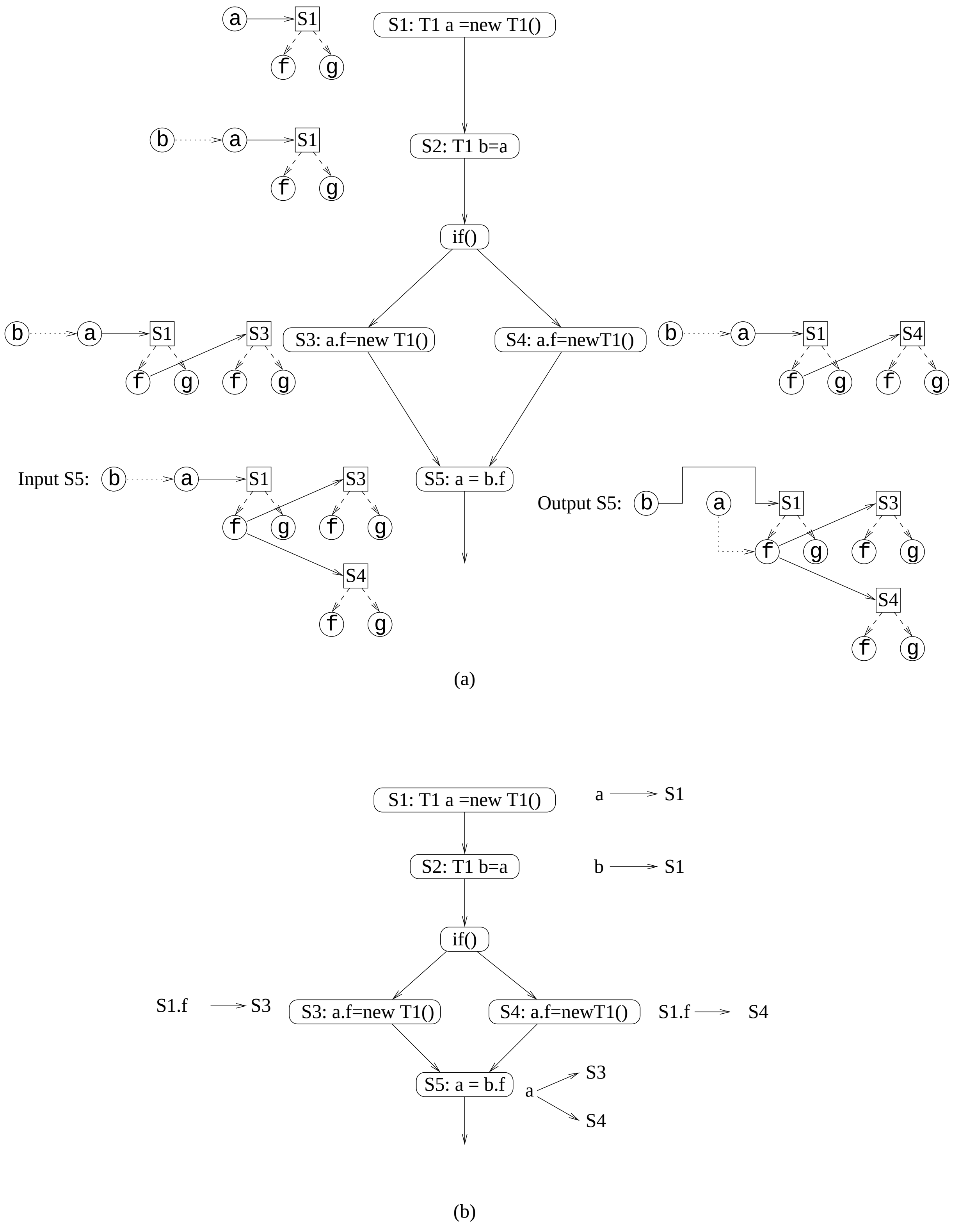


Figure 6.14: Comparison of Connection Graph representation with our sparse representation

requirement but also the overall execution efficiency. Note also the complexity of merging two branches even for such a small example.

The algorithm by Whaley and Rinard [89] is conceptually closest to ours although entirely different in the pragmatics. Our $EscapeIn$ nodes are similar to their *outside* nodes and Choi *et al*'s phantom nodes. Whaley and Rinard use a points-to graph representation that requires keeping the complete points-to information at each program point. They do not discuss the time and space complexity of maintaining such structures and of merging these graphs at merge points. But the space complexity is at least $O(N^2)$ and the complexity of a merge is also probably $O(N^2)$. In terms of precision their algorithm is as precise as ours except that they do not perform path-sensitive merges and hence may produce more conservative points-to information than our algorithm.

In terms of pragmatic implementation strategy, our algorithm is similar to that of Wilson and Lam [90] and that of Chase *et al* [21]. At each node, we maintain only the points-to information relevant to the node and build an SSA graph dynamically to generate the points-to information. This has the useful side effect of generating data dependence information and also converting the program into the useful SSA form. Unlike others, we have used the efficient DJ graph as a basis to build the SSA graph on top of. We have also ensured that the SSA form is maintained in the presence of multiple pointer updates at a program point. We perform path-sensitive merges.

Inter-procedural Analysis

Inter-procedural analyses tend to be either top-down [90] or bottom-up [89, 74, 25, 22]. Any of the standard schemes may be applied. We use an extension of Reps *et al*'s method for summarizing a method [49, 74]. The extension lies in determining all the variables that escape the procedure and adding $EscapeIn$ and $EscapeOut$ nodes for them. This leads to a very simple and elegant summary of the entire procedure.

6.6 Conclusion

In this chapter we have presented a flow-sensitive, context-sensitive, interprocedural data dependence algorithm for Java programs. In the course of generating data dependences, our algorithm also performs precise pointer analysis and escape analysis. Our approach uses a dataflow analysis framework that uses a sparse representation and is efficient in both time and space. It generates the useful SSA form as a by-product.

Our solution combines many different ideas and synthesizes them into an effective algorithm that provides new capabilities and is significantly more powerful than previous solutions. Though many of the basic ideas are not new there are many innovations too

- Intraprocedural analysis:

- Our SSA graph is built on top of the DJ graph and this gives the algorithm some interesting capabilities. When traversing up the dominator tree to find the definition of some variable v, the algorithm notes whether it has crossed any *merge* node. If not, the SSA def-use edge can be used directly in subsequent iterations, avoiding the traversal up the dominator tree.

- The traverse up the dominator tree can be terminated either on finding a definition or a use of the required variable v. This ensures that the number of nodes traversed is at most N for each variable in an iteration, where N is the number of nodes in the CFG.

- Our algorithm performs strong updates even on locally allocated heap blocks. In the example in Table 6.2.1 a weak update on locally allocated heap would imply that the definition of $O_1.f$ at S_4 would not kill the definition of $O_1.f$ at S_2. As a result, at S_5, x would point to both O_2 and O_4. This has a cascading effect as a subsequent definition of some field of x would also have to be treated as a weak update. This is a limitation in both Wilson and Lam's work and Choi *et al*'s work also.

- Our algorithm can build path-sensitive merges at no extra cost.

- Unlike other algorithms, our algorithm maintains the SSA property even in the presence of weak updates by creating special ϕ-nodes.

- Although we treat arrays as monolithic structures we maintain the SSA form even for array variables by simply treating all definitions as both a definition and a use of the array element.

- Unlike previous solutions we perform escape analysis using a sparse representation.

- In interprocedural analysis, our algorithm uses escape analysis innovatively to generate a simple and elegant summary of a procedure.

Although previous authors [25, 89] have not given analysis timing related statistics or complexity analyses, based on the above reasons, we believe our solution is more efficient, more precise and provides greater functionality than previous solutions.

Chapter 7

DATA DEPENDENCE ANALYSIS OF MULTITHREADED JAVA PROGRAMS

In Chapter 6 we have shown how to determine data dependence in the presence of pointers and aliasing. In this chapter we present an algorithm for data dependence analysis of concurrent Java programs. We extend the sequential data dependence analysis to handle programs with threads, monitors and synchronization.

7.1 Introduction

Consider the simple `producer-consumer` program shown in Figure 6.8. The `main` procedure creates a `Buffer` object (`Buffer b = new Buffer()`) and passes it to both the `Producer` and `Consumer` threads. As a result the value of `dc` computed in the procedure `get()` called from the `Consumer` thread depends on the value of `dc` computed in the procedure `put()` called from the `Producer` thread. We say that the two statements are *interference dependent* on each other. In order to determine when a statement may be interference dependent on another, it is necessary to determine which access paths refer to the same memory location. For static variables this is fairly straightforward, as the naming scope of the variables is global, but in the presence of heap allocated pointers this is not so simple. In the previous chapter we have shown how to determine whether two access paths in a procedure may refer to the same memory location. In this chapter, we show how to determine whether two access paths in two different threads may refer to the same memory location.

The slice of a program with respect to a program point p and a variable x consists of all statements and predicates of the program that might affect the value of x at p. Clearly, the slice of `dc = dc - 1` in `get()` must contain the statement `dc = dc + 1` in `put()`. To compute the slice of a concurrent program it is necessary to calculate the interference dependence edges [68].

Further, consider the same program with the following erroneous code in procedure `main`.

$$Producer\ p1 = \text{new}\ Producer(\text{new}\ Buffer(), 1);$$
$$Consumer\ c1 = \text{new}\ Consumer(\text{new}\ Buffer(), 1);$$
$$p1.start();$$
$$c1.start();$$

In this case, the `Buffer` object passed to each thread is different and hence the value of `dc` computed in the `Consumer` thread is *independent* of the value computed in the `Producer` thread. Clearly, we cannot determine interference dependence based on simple names of variables.

To solve this problem, we need to solve the problem of interthread pointer analysis and data dependence. In this chapter we give an inter-thread data dependence algorithm that

- can compute data dependences across threads,
- can handle monitors and synchronization,

7.2 Inter-thread Analysis

Motivation. In the Producer-Consumer example in Figure 6.8 and Figure 6.9, we need to determine that `dc` in `get()` and `dc` in `put()` refer to the same memory location and that `dc = dc - 1` in `get()` is interference dependent on `dc = dc + 1` in `put()`. Intraprocedure analysis generates the symbol $get.l_0.dc$ for `dc` in `get()` and the symbol $put.l_0.dc$ for `dc` in `put()`. Interprocedural analysis reduces $get.l_0.dc = get.l_0.dc - 1$ to $get.\mathcal{F}_0.dc = get.\mathcal{F}_0.dc + 1$ (Figure 6.10) and reduces $put.l_0.dc = put.l_0.dc + 1$ to $put.\mathcal{F}_0.dc = put.\mathcal{F}_0.dc + 1$ (Figure 6.11).

Interthread analysis will now determine that both $put.\mathcal{F}_0$ and $get.\mathcal{F}_0$ *map* to O_1 which is an object created in the `main` procedure (Figure 6.12) and passed to both the threads. Then we reduce $put.\mathcal{F}_0.dc$ and $get.\mathcal{F}_0.dc$ to $O_1.dc$. Now it is possible to use the name $O_1.dc$ to determine that both the statements refer to the same memory location.

7.2.1 The Control Flow Graph

The `run` procedure is treated like any other procedure as far as intraprocedural analysis is concerned. However, at the call site, the control flow graph contains a $Fork$ node. The $Fork$ node has two successors: (i) the entry node of the `run` procedure and (ii) the statement that lexically follows the $Fork$ node. At the $Fork$ node, we insert appropriate *ActualIn* nodes before the call to the `run` procedure as explained in Section 6.2. However, we do not insert *ActualOut* nodes. *ActualOut* nodes are inserted only at a *Join* node, where a *Join* node is a node where two threads join using the Java procedure `join()`. In all other respects the *Join* node is treated as any other merge node. For open-ended threads we never insert *ActualOut* nodes. A procedure is duplicated for every thread from which it is called.

7.3 The Algorithm

Alias analysis can at best be conservative [72] and inter-thread alias analysis is more so [73]. Here we give a conservative algorithm for inter-thread data dependence analysis.

The inter-thread algorithm has two phases: mapping and pointer analysis. The mapping phase collects every node in the threaded CFG that either references, defines or points to a globally escaping variable. The mapping phase starts at the `main` procedure and at each procedure invocation site, for each *ActualIn* node of the form $\mathcal{A}_i \rightarrow \psi$, it maps the corresponding $FormalIn$ node $\mathcal{F}_i$ in the called procedure to ψ. This process repeats recursively down the call graph. The effect of the mapping phase is that parameters of two procedures map to the same name if they may access the same memory location.

After the variables of each procedure have been mapped, the pointer analysis phase attempts to find a definition of a memory location that is referenced in a parallel thread and performs pointer analysis across threads.

7.3.1 Mapping

At each call site in a procedure, we inspect the *ActualIn* nodes for reference variables. For an *ActualIn* node with $\mathcal{A}_i \to \psi$, we generate the mapping $\mathcal{F}_i$ *maps to* ψ, where $\mathcal{F}_i$ is the formal variable in the called procedure corresponding to $\mathcal{A}_i$. For example, in Figure 6.12 z_{27} is the call site for *Producer.run*. z_{22} is an *ActualIn* node with $\mathcal{A}_0 \to O_2$. So we generate the mapping $Producer.run.\mathcal{F}_0 \mapsto O_2$. At z_{23}, $\mathcal{A}_1 \to O_1$ generates the mapping $Producer.run.\mathcal{F}_1 \mapsto O_1$. The other *ActualIn* nodes do not generate any mapping. If the called procedure is the special `run` procedure that starts a new thread, then all the symbols in ψ are marked as *GlobalEscape*. In the example, O_2 and O_1 are marked *GlobalEscape*. The first (and only) parameter passed to `run` is the thread object. However, other objects may enter the thread through the *EscapeIn* nodes.

For each variable, v, in the symbol table let $v.locations$ be the set of nodes where v is defined. Let W be an empty worklist of nodes. The mapping phase locates every node that uses, defines or points to a globally escaping variable and adds it to the worklist. It also replaces each use of a formal variable by the variable that it maps to. If we ignore back edges in the call graph, the call graph reduces to a directed acyclic graph (DAG) with a unique entry node. We visit the nodes of the call graph in an order that ensures that a procedure is visited only if all its predecessors in the DAG have been visited. The algorithm is in Figure 7.1

Recursively Generated Threads In the presence of recursively generated threads (*i.e.,* a thread is generated from a recursive procedure), the mapping algorithm needs to iterate over the strongly connected component of the call graph that generates a thread.

Example In Figure 6.11, in *Producer.run*, we have already explained the mapping that $\mathcal{F}_0 \mapsto O_2$ and $\mathcal{F}_1 \mapsto O_1$. Within, *run*, we replace all occurrences of $\mathcal{F}_0$ by O_2 and $\mathcal{F}_1$ by O_1. Therefore, at v_{10}, $\mathcal{A}_0 \to \mathcal{F}_1$ will be replaced by $\mathcal{A}_0 \to O_1$. Further, since v_{10} is an *ActualOut* node for the call to *put*, we also generate the mapping $put.\mathcal{F}_0 \mapsto O_1$. When the analysis, enters the procedure *put*, all occurrences of $put.\mathcal{F}_0$ will be replaced by O_1. So, at w_{11} and w_{12}, $\mathcal{F}_0.dc = \mathcal{F}_0.dc + 1$ will be replaced by $O_1.dc = O_1.dc + 1$. Similarly, in the procedure *get* in Figure 6.10 also, $get.\mathcal{F}_0 \mapsto O_1$ and at u_{10} and u_{11}, we get $O_1.dc = O_1.dc - 1$.

for each procedure P **do**
for each node $S_i \in P$ **do**
for each symbol $v_i \in S_i.DEF \cup S_i.USE$ **do**
if v_i is an object variable $\mathcal{F}_j$ **then**
if $\mathcal{F}_j \mapsto O_j$ and O_j is marked *GlobalEscape* **then**
replace $\mathcal{F}_j$ by O_j
$W = W \cup S_i$
if v_i is an object variable $O_i^{\mathcal{F}_i}$ **then**
if $\mathcal{F}_j \mapsto O_j$ and O_j is marked *GlobalEscape* **then**
Mark O_i as *GlobalEscape*
$W = W \cup S_i$
if v_i is an object field variable $\mathcal{F}_j.f$ **then**
if $\mathcal{F}_j \mapsto O_j$ and O_j is marked *GlobalEscape* **then**
replace $\mathcal{F}_j.f$ by $O_j.f$
$W = W \cup S_i$
if $v_i \in S_i.DEF$ **then**
$(O_j.f).locations = (O_j.f).locations \cup S_i$
if v_i is an object field variable $O_j.f$ **then**
if either O_j or $O_j.f$ is marked *GlobalEscape* **then**
$W = W \cup S_i$
if $v_i \in S_i.DEF$ **then**
$(O_j.f).locations = (O_j.f).locations \cup S_i$
if v_i is a static variable **then**
$W = W \cup S_i$
if $v_i \in S_i.DEF$ **then**
$v_i.locations = v_i.locations \cup S_i$
endfor
if S_i is an *ActualIn* node s.t. $\mathcal{A}_i \to \psi$ **then**
let P' be the procedure called from the call site corresponding to $\mathcal{A}_i$
let $\mathcal{F}_i'$ be the *FormalIn* node in P' corresponding to $\mathcal{A}_i$
let $\mathcal{F}_i' \mapsto \psi'$, **then**
$\psi' = \psi' \cup \psi$
endfor
endfor

Figure 7.1: Mapping algorithm

7.3.2 Pointer Analysis

We define the modified function $RdPoint'(v, n_i)$ for the variable v at a node n_i as follows: $RdPoint'(v, n_i)$ locates the reaching definition of v at n_j. (Since all the procedures are in SSA form the function traverses exactly one def-use edge backwards). $RdPoint'(v, n_i)$ returns all elements of $n_j.DEF$ that are marked $GlobalEscape$. $RdPoint'(v, n_i).f$ indicates the set of object field variables $O_i.f$ such that $O_i \in RdPoint'(v, n_i)$. The set of *intra*-thread def-use edges can not change but the analysis will now generate the interference dependence edges.

The inter-thread pointer analysis is similar to the intra-procedural analysis except that we process only those variables that escape globally. For the use of static variables and globally escaping object field variables we have to do some extra processing as these are the variables that are visible to more than one thread. If there is a use of a variable v at S_i in thread θ_i and a definition of v at S_j in another thread θ_j such that S_i and S_j may execute in parallel, then we draw an interference dependence edge and also propagate the pointers from the defining site, S_j to the site where the variable is used, S_i.

During the mapping phase, we had collected in the worklist W, all the nodes in the control flow graph that may either use or define a globally visible variable. The pointer analysis iterates over the elements, n_i, in the worklist W. The algorithm is given in Figure 7.2.

Example The globally escaping variable $O_1.dc$ is defined and used at u_{10} and u_{11} in the *consumer* thread (Figure 6.10) and it is also defined and used at w_{11} and w_{12} (Figure 6.11) in the *producer* thread. These locations will be recorded in the symbol table entry for $O_1.dc$. When processing u_{10}, the analysis finds a use of $O_1.dc$. It looks at the location set for $O_1.dc$ and discovers that the definitions at w_{11} and w_{12} execute in parallel with u_{10}. Interference dependence edges are drawn from each of w_{11} and w_{12} to u_{10}. Since, $O_1.dc$ is not a reference variable, no pointer information is transferred.

```
change = 1
while change == 1 do
    change = 0
    for each node n_i ∈ W do
        for each variable v_j ∈ n_i.DEF do
            if v_j is a field variable l_j.f then
                for each v_k ∈ RdPoint'(l_j, n_i).f
                    n_i.DEF = n_i.DEF ∪ v_k
                    v_k.locations = v_k.locations ∪ n_i
                endfor
        endfor

        ψ = {}
        for each variable v_j ∈ n_i.USE do
            if v_j is a field variable l_j.f then
                n_i.USE = n_i.USE ∪ RdPoint'(l_j, n_i).f
            if v_j ∈ V_o and v_j is marked GlobalEscape then
                ψ = {v_j}
            if v_j is a local reference variable then
                ψ = RdPoint'(v_j, n_i)
            if v_j is an object field variable/static variable then
                ψ = RdPoint'(v_j, n_i)
                for each n_j ∈ v_j.locations do
                    if (||(n_i, n_j) == true) then
                        EdgeInterference(n_j, n_i)
                        ψ = ψ ∪ n_j.PTS
                endfor
        endfor
        if n_i is an assignment statement v_k = v_j then
            for each d_k ∈ n_i.DEF and d_k ∈ V_l ∪ V_s ∪ V_of do
                d_k.PTS = d_k.PTS ∪ ψ
        if any DEF, USE, PTS or location set has changed then
            change = 1
    endfor
endwhile
```

Figure 7.2: Interthread pointer analysis algorithm

7.4 Monitors

The inter-thread pointer analysis algorithm generates an interference dependence edge from w_{11} to u_{10}. However, w_{11} is inside a synchronized method and the definition at w_{11} is killed by the definition at w_{12}. Hence this definition cannot reach either u_{10} or u_{11}.

Each monitor is associated with a variable in the symbol table. If at the end of the mapping phase, the variable is not yet marked as $GlobalEscape$, we conservatively assume that the monitor does not exist. With each node we associate two sets $DefEnclose$ and $UseEnclose$. If a definition at S_i is enclosed within a monitor, O_i which is marked $GlobalEscape$, and does not reach the `monitorexit` node then we set $S_i.DefEnclose = S_i.DefEnclose \cup O_i$. Similarly, if a use at S_i is enclosed within a monitor, O_i which is marked $GlobalEscape$, and the reaching definition for the use lies inside the monitor section then we set $S_i.UseEnclose = S_i.UseEnclose \cup O_i$.

In the inter-thread pointer analysis algorithm when checking if the defining node S_i and the use node S_j may execute in parallel we make an additional check: if S_i and S_j are both enclosed within the same monitor, O_i, and either $O_i \in S_i.DefEnclosed$ or $O_i \in S_j.UseEnclosed$ then we do not build interference dependence edges or transfer pointer information.

7.5 `wait/notify` Synchronization

An important issue in `wait/notify` synchronization analysis is to determine which `wait` statements corresponds to which `notify` statement. In our analysis, this reduces to the simple task of detecting interference dependence edges from the $write(O_i.\langle lock\rangle)$ statement at the `notify()` statement to the $read(O_i.\langle lock\rangle)$ statement at the `wait()` statement. `notify()` Once these edges have been calculated, it is possible to apply standard algorithms [58, 37, 54] to determine a partial order induced by the synchronization edges [58, 37, 54]. This may further reduce the number of interference dependences.

7.6 Complexity

The mapping phase visits each node in the control flow graph exactly once. In the pointer analysis phase, the use of a variable at a node S_i requires a test on every node at which the variable is defined. The number of defining sites could in the worst case be $O(N)$ and hence the worst case complexity of the analysis is $O(N^2)$.

7.7 Related Work

Many inter-thread analyses [43, 51, 58, 69] are inappropriate for pointer analysis. Grunwald and Srinivasan [43] present a dataflow framework for reaching definitions for explicitly parallel programs with a weak memory model. Knoop, Steffen and Vollmer [51] present an efficient dataflow analysis framework for bit-vector analysis which applies to the strong memory model. Neither framework applies to pointer analysis.

Tang *et al* [85] and Zhu *et al* [95] analyze multithreaded programs but their pointer analysis is based on the conservative connection analysis that does not distinguish between fields of an object. Rugina and Rinard [78] use a pointer analysis that models heap conservatively and applies weak updates to all heap analysis. They analyze only structured, properly nested threaded structures without monitors and synchronization. Our analysis handles both open-ended threads and structured `fork/join` constructs.

7.8 Conclusion

In this chapter we have presented a flow-sensitive, context-sensitive, inter-thread data dependence algorithm for multithreaded Java programs. Although inter-thread analysis is inherently conservative [73], our inter-thread analysis handles heap analysis with much greater precision than previous solutions. We can handle both structured `fork/join` constructs as well as open-ended threads. We can handle recursively generated threads. We can handle monitors and `wait/notify` synchronization with some precision.

Chapter 8

SLICING CONCURRENT INTERPROCEDURAL JAVA PROGRAMS

In chapter 7 we have described how to determine data dependencies for concurrent Java programs. In this chapter we describe the remaining modifications required to slice concurrent interprocedural Java programs. Then we show how to improve the slicing performance along the lines of the intra-procedural optimizations.

8.1 Modifications

Unlike the structured `cobegin-coend` parallelism construct, Java supports `fork/join` parallelism. At a `fork` node, the parent thread generates a child thread which continues to execute in parallel with the remainder of the parent thread. The child thread may be open-ended or it may `join` the parent thread at some later point. The semantics of a `join` node are similar to the semantics of a `coend` node.

Due to the different structure, the functions $||(i, j)$ and $RealizablePath$ are affected. In this chapter, we review each part of the slicing algorithm in the light of the new control flow structure.

8.1.1 The $||(i, j)$ Function

Let $StartNode(\theta_i)$ be the `START` node of the thread θ_i. In the basic shared-memory concurrency model with `cobegin-coend` parallelism, $||(S_i, S_j)$ is $true$ if $CoAncestor($

$StartNode(\theta(S_i))$, $StartNode(\theta(S_j)))$ is a `cobegin` node (*i.e.*, the nearest common ancestor of the two thread headers is a `cobegin` node). Hence it was sufficient to keep parallel information at the granularity of a thread and keep a pointer from each node to its closest enclosing thread.

Now consider the simple Java program shown in Figure 8.1. S_1, S_3 and S_4 belong to θ_0. However, of these, only S_3 executes in parallel with the nodes in θ_1. Clearly we need to keep some extra information to determine whether two nodes may execute in parallel.

We break each thread into a series of regions $\mathcal{R}_i$ with a designated `START` node returned by the function $StartNode(\mathcal{R}_i)$. Just as $\theta(S_i)$ returns the thread id to which the node S_i belongs, we define a function $ThreadRegion(S_i)$ which returns $\mathcal{R}_i$, if S_i belongs to $\mathcal{R}_i$. We term these regions "ThreadRegions". We start a new ThreadRegion at

- The `START` node of a thread.
- A $Fork$ node.
- A $Join$ node.
- A merge node which has two predecessors belonging to different ThreadRegions.

The ThreadRegion terminates when a new ThreadRegion begins or when the thread ends. We show how to determine the ThreadRegions dependent on the control flow graph and then explain how to use this information to determine the answer to $||(S_i, S_j)$.

- **Straight line code:** In Figure 8.1 the `START` node of the main thread θ_0 is the start node of $\mathcal{R}_0$. The $Fork$ node is designated as the start node of $\mathcal{R}_1$, the `START` node of θ_1 is designated as the start node of $\mathcal{R}_2$ and the $Join$ node is designated as the start node of $\mathcal{R}_3$. $\mathcal{R}_0$, $\mathcal{R}_1$ and $\mathcal{R}_3$ belong to θ_0 and $\mathcal{R}_1$ belongs to θ_1.
- **Threads generated within branches:** Figure 8.2(a) shows the ThreadRegions generated when a thread is created and joins within the same branch of an `if` statement. This is similar to the `cobegin-coend` construct. Figure 8.2(b) shows the ThreadRegions generated when the threads generated

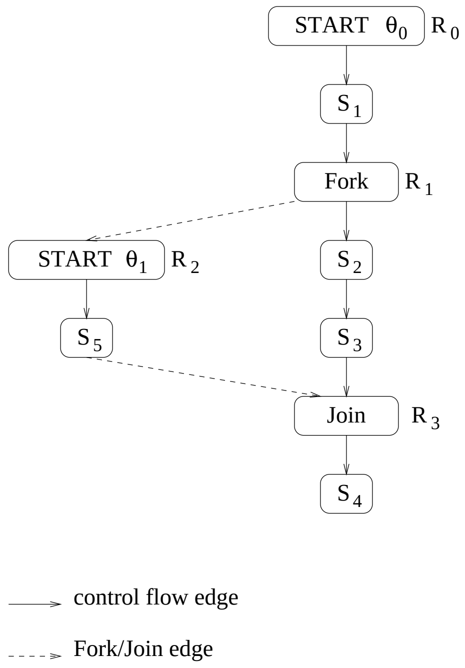


Figure 8.1: Straight line code

are either open-ended or join after the merge corresponding to the `if` statement.

- **Threads generated within loops:** Figure 8.3(a) shows the ThreadRegions generated when a thread is created and joins within the same loop. This is similar to the `cobegin-coend` construct. Figure 8.3(b) shows open-ended threads being generated within a loop. In this case we make a change in the control flow graph as shown in Figure 8.3(c) and mark the ThreadRegions as shown. Basically, we convert the loop entry node into a special $Fork$ node. The edges to the new threads are marked as $FORK\_EDGE$ to differentiate them from the ordinary control flow edges. For every thread generated within the loop we generate two representative threads that fork from the loop header as shown. The "`invokevirtual`" call node is treated as an ordinary call node and the $ActualIn$ nodes are inserted before this call node.

Once these modifications are complete the rule for parallel execution of two nodes S_i and S_j are as follows:

- Let $\theta_i.join$ designate the node at which θ_i joins the parent thread.

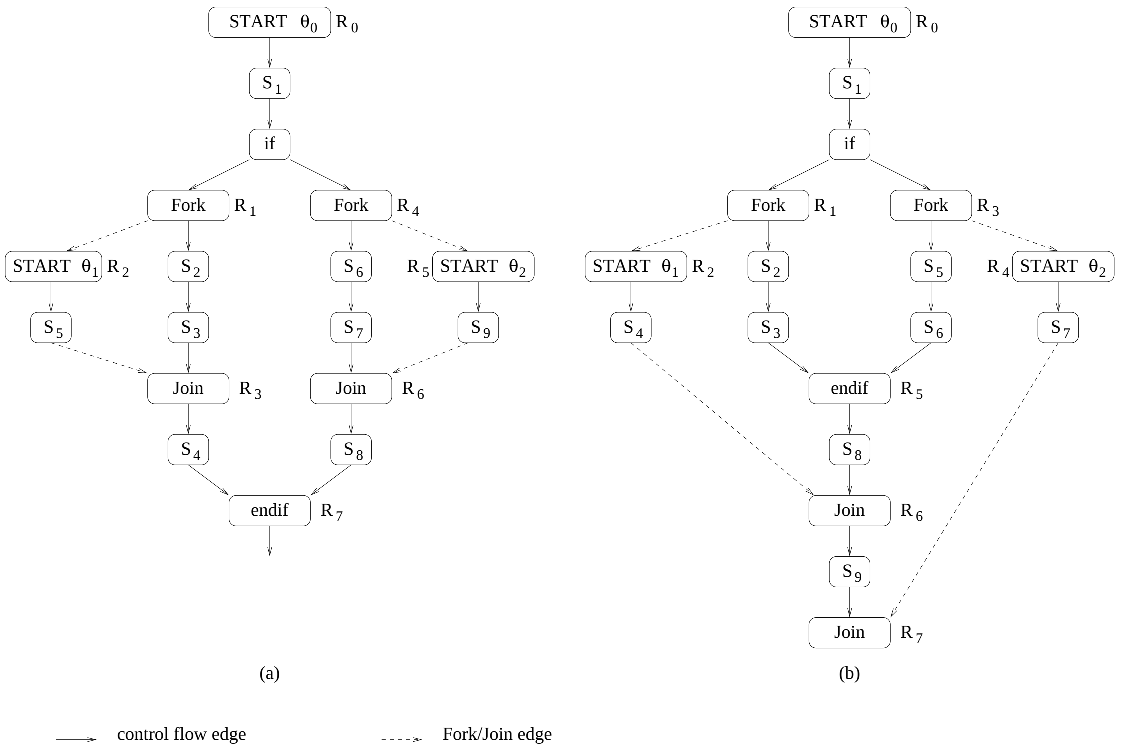


Figure 8.2: Thread generated within a branch

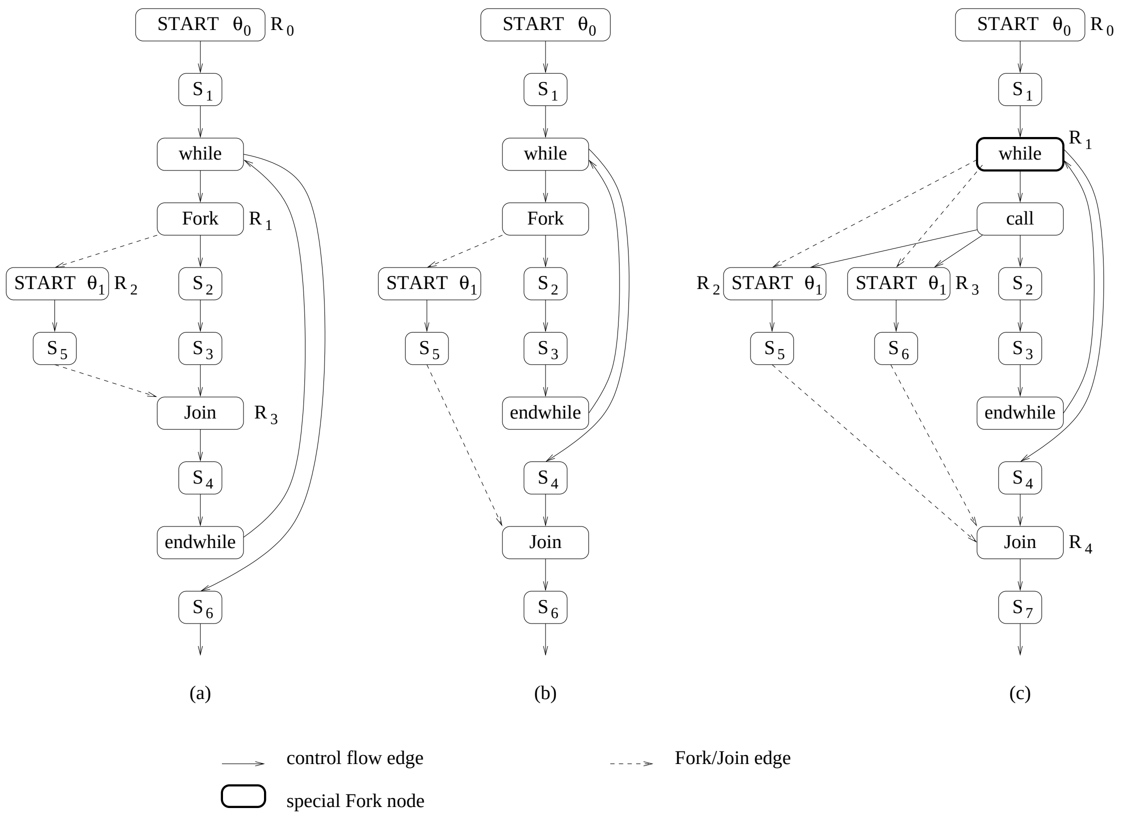


Figure 8.3: Threads generated within a loop

- Let $ThreadRegion(S_i)$ be $\mathcal{R}_i$ and $ThreadRegion(S_j)$ be $\mathcal{R}_j$.
- Let $\theta(S_i)$ be θ_i and $\theta(S_j)$ be θ_j.
- $||(S_i, S_j) == true$ if
  - $\theta_i \neq \theta_j$ and
  - $CoAncestor(\mathcal{R}_i, \mathcal{R}_j)$ is a $Fork$ or special $Fork$ node and
  - if $StartNode(\theta_i)$ is an ancestor of $StartNode(\theta_j)$ and $\theta_j.join$ does not precede $StartNode(\mathcal{R}_j)$ or vice versa.

Given these rules the parallel information can be calculated at the granularity of the ThreadRegions and stored in small bitvectors so that $||(S_i, S_j)$ can be calculated in constant time.

8.1.2 RealizablePath Computation

The $RealizablePath$ computation is determined for each thread as explained in Section 4.2.3. The $RealizablePath$ computation ignores all $Fork$ and $Join$ edges. We define a function $ThreadRegion(S_i)$ which returns the ThreadRegion to which S_i belongs. We also compute $REACH(\mathcal{R}_i, \mathcal{R}_j)$ over the start nodes of all the ThreadRegion and store it in small bitvectors. For example, $\mathcal{R}_j.REACH$ will have one bit for each ThreadRegion and the bit for $\mathcal{R}_i$ will be set if there is a path from $\mathcal{R}_i$ to $\mathcal{R}_j$. Then to find out if there is a path S_i to S_j in the TCFG, we first check if they belong to the same thread. If yes, we apply the $RealizablePath$ algorithm. If not, we need to check if there is a path from the ThreadRegion to which S_i belongs to the ThreadRegion to which S_j belongs (*i.e.*, $ThreadRegion(S_i) \in ThreadRegion(S_j).REACH$).

8.1.3 Control Dependence

Control dependence is calculated using standard techniques [49] for intra-procedural and inter-procedural analysis. At the inter-thread level, we generated control dependence edges from the `START` node of each thread to the corresponding $Fork$ node from which it was generated. In Java, `exit()` statements and statements that throw exceptions can affect the control dependence of interprocedural programs [82, 45]. This is an issue we have not yet addressed.

8.1.4 Data Dependence

The details on how to determine data dependence and interference dependence in Java programs in the presence of threads, monitors and synchronization have already been explained in Chapters 6 and 7.

8.1.5 Synchronization Dependence

The synchronization primitives in Java are `wait` and `notify`. `wait` causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object. The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the `notify` method or the `notifyAll` method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.

`notify()` wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

In our analysis, synchronization primitives `wait` and `notify` are handled as follows: Synchronization edges are drawn from a `notify` node to the corresponding `wait` nodes. Each object has an implicit lock associated with it. `wait` is treated as having an implicit read on the lock variable and `notify` is treated as having an implicit write on the lock variable. Then the data dependence calculation automatically detects the dependence between the relevant `wait` and `notify` nodes. In addition, a `wait` statement implies the release of all locks held by the object and hence this is treated as an edge of a monitor section and all definitions reaching the `wait` statement are downward exposed and visible to other threads.

8.2 Complexity

In intraprocedural programs each node in the slice may be inserted into the worklist $O(n^{t-1})$ times, where n is the number of nodes in a thread and t is the number of threads. This gives a slicing complexity of $O(N^t)$, where N is the number of nodes in the graph. In interprocedural programs, a node may be represented by an exponential number of topological numbers and for each topological number it may be inserted into the slice an exponential number of times. This makes the complexity of the algorithm doubly exponential $O(N^{p^t})$, where p is the calling depth of the call graph. However, similar to the optimizations for intraprocedural programs [68] it is possible to apply some optimizations to the interprocedural analysis also.

8.2.1 Improving the Performance

In intraprocedural slicing optimizations, there is a concept of a less restrictive tuple versus a more restrictive tuple. For example, in Figure 3.8, the tuple $[S_1, S_1, \perp, \perp]$ is less restrictive than the tuple $[S_1, S_1, S_3, \perp]$. The first tuple says that any node in θ_2 and θ_3 may be visited, whereas the second tuple says that any node in θ_3 may be visited but in θ_2 *only* those nodes can be visited *that have a path to* S_3. The essence of the intraprocedural optimizations is to ensure that less restrictive tuples are sliced first and more restrictive tuples are discarded. In order to ensure that the slicing algorithm "finds" less restrictive tuples first, the algorithm gives nodes reached via interference dependence edges a low priority.

We extend the same notion to interprocedural slicing as follows: Inside the procedure *Insert* (Figure 8.4, 8.5), let mu have k topological numbers n_1, n_2, …, n_k. Instead of creating one worklist element, we create k worklist elements. Each worklist element has one topological number from the set mu. The data structures do not change, except that now in a tuple T, $T[i].\mu$ is either the empty set or it is a set with exactly one element. Having made this modification it is possible to order the elements in the worklist. Then we apply the optimization used in intraprocedural slicing as follows:

1. Given that a node S_y has been inserted into the worklist with a tuple $[(s_1, \mu_1)$, (s_2, μ_2), …, $(s_n, \mu_n)]$, if it gets a new tuple $[(s'_1, \mu'_1)$, (s'_2, μ'_2), …, $(s'_n, \mu'_n)]$, where for all i, either s_i is $\perp$ or there is a feasible path from (s'_1, μ'_1) to (s_1, μ_1), then the new tuple is redundant and need not be added to the slice.

2. Nodes reached through interference dependence need to be given a low priority. This is already ensured by the algorithm since nodes reached via interference dependence edges are placed in the outermost queue which is processed only after nodes reached through other edges.

The complexity of the algorithm does not change but the running time of the algorithm was observed to improve on our sample programs.

By breaking each topological number set into its individual components, it might appear that the number of elements inserted into the worklist is going to increase exponentially. However, note that, by the optimization, if there is a path from one topological number n_i to another, n_j in the set mu, then a worklist element will be created only for n_j. Only in the case when a procedure is called from multiple branches of a switch which is not embedded within a loop do we actually end up inserting a large number of elements into the worklist. In practice, the improvement in slicing time to be gained by avoiding unnecessary paths, far outweighs the overhead of breaking the topological set into individual worklist elements.

Another practical optimization that we built into our implementation is as follows: if a thread has no incoming interference dependence edges (*i.e.* there is no node in the thread which is the source of an interference dependence edge) then we need not apply RealizablePath or maintain tuples for any node in that thread. This thread can never be reached via an interference dependence edge and hence may be sliced as a sequential thread. With this optimization, the algorithm reduces to the sequential two-phase algorithm in the absence of threads or interference dependence edges.

The final algorithm with the optimizations built in is given in Figure 8.4.

8.3 Related Work

For concurrent programs, most of the work has been done on intraprocedural slicing. Cheng [23] presents an approach for slicing programs where interprocess communication is channel-based. As Tip [86] notes, he does not state or prove any property of the slices computed. Krinke [53] gives a precise slicing algorithm for programs with shared memory. This however becomes imprecise in the presence of nested threads and threads nested within loops. Nanda *et al* [68] give a more efficient slicing algorithm which remains precise in the presence of all program constructs. None of these handle issues such as synchronization and monitors.

Hatcliff *et al* [46] analyze Java programs with monitors and synchronization. They introduce several new dependencies including a *ready dependence*, where a node, S_i, is ready dependent on another node, S_j, if S_j's failure to complete could imply that S_i never executes. For example, a `wait` statement is ready dependent on the corresponding `notify` statement and all statements that may be reached from a

Input: the slicing criterion s
Output: the slice $\mathcal{S}$
mu = GetTopologicalNumberSet(s)
for each $m \in mu$ **do**

$$C = (s, [t_0, \ldots, t_n], \text{phase1}), \begin{cases} \textbf{if } ||(\theta(s), \theta_i) \text{is } \textbf{false then} & t_i.node = s,\ t_i.\mu = \{m\} \\ \textbf{else } t_i.node = \perp,\ t_i.\mu = \emptyset \end{cases}$$

Insert $(s, \{m\}, [t_0, \ldots, t_n]$, `phase1`, **w0**)
endfor

while w0 $\neq \emptyset$ **do**
remove next element from **w0** and insert into **w1**
while w1 $\neq \emptyset$ **do** /* Phase 1 */
remove next element $c = (x, T, color)$ from **w1**
for all $y \mid (y, x) \in E_{id}$
$t = T[\theta(y)].node$, $mu = T[\theta(y)].\mu$
Insert (y, $RealizablePath'(y, t, mu)$, T, `phase1`, **w0**)
for all $y \mid (y, x) \in E_{dd} \cup E_{cd} \cup E_s \cup E_{pi} \cup E_c$
$t = T[\theta(y)].node$, $mu = T[\theta(y)].\mu$
Insert (y, $RealizablePath'(y, t, mu)$, T, `phase1`, **w1**)
for all $y \mid (y, x) \in E_{po}$
$mu = T[\theta(x)].\mu$
Insert (y, $RealizablePath(y, x, mu)$, T, `phase2`, **w2**)
endwhile

while w2 $\neq \emptyset$ **do** /* Phase 2 */
remove next element $c = (x, T, color)$ from **w2**
for all $y \mid (y, x) \in E_{id}$
$t = T[\theta(y)].node$, $mu = T[\theta(y)].\mu$
Insert (y, $RealizablePath'(y, t, mu)$, T, `phase1`, **w0**)
for all $y \mid (y, x) \in E_{dd} \cup E_{cd} \cup E_s \cup E_{po}$
$mu = T[\theta(x)].\mu$
Insert (y, $RealizablePath(y, x, mu)$, T, `phase2`, **w2**)
endwhile
endwhile

Figure 8.4: The Context-Sensitive Interprocedural Slicing Algorithm With Optimizations

```
procedure Insert ( Node: y, TopologicalNumbersSet: mu, Tuple: T, int color, List: ℒ )
    if mu == ∅ then return
    for each m ∈ mu do
        for all i s.t. ||(θ_i, θ(y) is false do
            T[i].node = y
            T[i].μ = {m}
        endfor
        for each previous tuple, T', of y do
            redundant = true
            for all i do
                if RealizablePath''(T[i], T'[i]) == false then
                    redundant = false
                    break
            endfor
            if redundant == true then return
        endfor
        c' = (y, T, color)
        𝒮 = 𝒮 ∪ {y}
        ℒ = ℒ ∪ c'
    endfor

procedure RealizablePath' ( Node: y, Node: t, TopologicalNumberSet: mu )
    if θ(y) == θ(t) then
        return RealizablePath(y, t, mu)
    if ThreadRegion(y) ∈ ThreadRegion(t).REACH then
        return GetTopologicalNumberSet(y)
    else
        return ∅

procedure RealizablePath'' (
        (Node: x, TopologicalNumberSet: {m_x}),
        (Node: y, TopologicalNumberSet: {m_y}) )
    if y == ⊥ return true
    if Reach(m_x, m_y) return true
    return false
```

Figure 8.5: The Context-Sensitive Interprocedural Slicing Algorithm With Optimizations – continued

`wait` statement are ready dependent on the `wait` statement. However, they ignore the imprecision introduced by the intransitivity of interference dependence. Other than Nanda *et al*, none of the others give implementation details or statistics.

Slicing intraprocedural programs has a limitation in that it is often impractical to slice realistic programs. Procedures need to be inlined causing the control flow graph to grow potentially without bounds. Programs with recursion either cannot be analyzed or are analyzed conservatively.

Millet and Teitelbaum [64] give an algorithm for slicing Promela. They also use an extension of the SDG but it is not clear whether their algorithm is context-sensitive. The only other algorithm for slicing interprocedural concurrent programs that we are aware of is the one by Zhao [92]. This, as we have already shown, is both incorrect and imprecise.

8.4 Conclusion

In this chapter we have shown how to extend the slicing algorithm to real concurrent Java programs. The complexity of the algorithm is exponential in the number of threads but with the suggested optimizations proves to be very practical. We improve on the accuracy of previous solutions without losing out on practicality.

Chapter 9

IMPLEMENTATION

In this Chapter we give details of our implementation of the realizable path algorithm, the data dependence analysis for multithreaded Java programs and context-sensitive and context-insensitive slicing algorithm for concurrent Java programs. The test programs include some student programs as well as several standard Java benchmark programs.

Testing was performed on a uniprocessor 770Mhz Intel Pentium III workstation with 256MB of memory running under Linux.

9.1 Test Program Details

Since the focus of the dissertation is concurrent programs, most of our benchmark programs are multithreaded programs. However, for the sake of completeness, we include results on three standard single-threaded programs also, where relevant. The programs have been described in Table 9.1. `pc` is a simple producer-consumer program that is used for illustrations in many chapters, `multi` is a multithreaded program that downloads a web site and all its links so that it can be browsed offline, `raytracer` is a multithreaded ray tracer, `Slice` is a modified version of a public domain applet obtained from Eric Ruf's web page [77], `hanoi` is an IBM benchmark program, `volano` is the VolanoMark server, and `mandelbrot` is Simon Arthur's applet to explore the Mandelbrot set.

The three singlethreaded programs are `javacup`, a Java constructor of parsers, `espresso`, a javac clone, and `jlex`, a lexer generator.

Table 9.1: Benchmark programs

name	number of classes	size of classes	methods	statements
pc	152	276K	233	35411
multi	158	285K	496	74038
raytracer	251	543K	605	226166
slice	257	563K	745	236169
hanoi	298	587K	705	308964
volano	265	564K	540	116680
mandelbrot	572	673K	786	468171
javacup	193	430K	575	533815
espresso	252	569K	637	662801
jlex	167	367K	430	551044

Table 9.2: The ISCR graph Statistics

name	Build Time	LOC	No. of iSCR	No. of ISCR	Largest Topological No.	Max numbers at an iSCR
pc	0.01	54	9	6	6	1
multi	0.09	55049	2461	1336	13174	348
raytracer	1.93	207249	4212	1636	20425	672
slice	1.68	212155	4965	2474	135166	4031
hanoi	3.65	290059	5170	2629	289155	8256
mandelbrot	22.62	449266	6316	3479	703312	19104

9.2 Realizable Path Analysis

This section gives statistics about the algorithm to build the ISCR graph, the algorithm to generate topological numbering, and the RealizablePath algorithm. In Table 9.2 we give statistics related to the ISCR graph. For each program, we give statistics for the thread with the maximum number of statements and the related ISCR graph characteristics. The table gives the number of statements in the thread, the number of iSCRs, and the number of ISCRs.

The table also gives the largest topological number. This would have governed the size of the bitvector if we had generated bitvectors. The maximum number of

Table 9.3: Data Dependence Analysis of Sequential Java Programs

name	Time in seconds		No of edges		
	whole program	minus libraries	DD	DD(weak)	increase
javacup	439.12	96.14	534304	538859	4555
espresso	579.57	38.8	539604	543264	3660
jlex	230.28	1.56	480846	482557	1711

ISCR nodes visited by *Reach* to determine if there is a path from an ISCR, x to another ISCR y is equal to the maximum number of ISCRs in the graph. As can be seen from the table, the maximum number of ISCRs is of the order of a few thousands and hence *Reach* is very fast. It is clear from the table why the bitvector solution fails. In `mandelbrot`, for example, the largest topological number is 703312. This would require sufficient memory to hold $703312 * 703312$ bits which is roughly 60GigaBytes. In contrast, the solution now needs enough memory to hold 703312 integers which is roughly 2.5MB, which is far more manageable.

The last column gives the maximum number of topological numbers associated with a single iSCR. This governs the number of iterations in the two loops in the $RealizablePath$ algorithm.

The time in seconds to build the ISCR graph and to generate the topological numbers is also given. The time to run the *RealizablePath* algorithm which internally calls *Reach* is not given as it was too small to measure in any test case.

9.3 Data Dependence Analysis of Sequential Java Programs

Analysis time The programs were analyzed with the library files provided with the Java installation and any additional libraries required by the specific program. In general, the library code tends to be much larger than the actual program and it is possible to analyze the libraries and summarize them in appropriate files. We give two sets of analysis time - the first set gives the total analysis time including analysis of all libraries and the second set gives the analysis time of the program when the standard Java library (rt.jar) has been pre-analyzed (the program specific libraries

Table 9.4: Analysis timing in seconds for Data Dependence Analysis of Concurrent Java Programs

name	Build+DD+CD		Mapping	Interthread	Total
	whole program	minus libraries			
pc	1.2	0.02	0.05	0.17	1.42
multi	3.74	0.15	0.13	0.66	4.53
raytracer	49.37	0.14	0.61	5.37	55.35
slice	37.86	0.29	0.65	5.16	43.67
hanoi	99.29	4.07	0.88	8.32	108.49
volano	212.36	1.01	0.73	8.19	221.28
mandelbrot	305.04	0.31	2.02	21.97	329.03

are re-analyzed every time). The time of analysis includes the time taken to build the control flow graph, the control dependence analysis and the data dependence analysis. The time of analysis in seconds is given in Table 9.3.

Data dependence In Table 9.3, we also show the number of data dependence edges in each program. We investigated the effect of weak updates on heap allocated variables by forcing our implementation to perform only weak updates on heap allocated variables. The corresponding number of data dependence edges and the increase in edges is shown in the table. There was only a marginal increase (less than 1%) in the number of data dependence edges.

Our implementation handles parameter aliasing conservatively. An alias that holds at one call site holds at all call sites.

9.4 Data Dependence Analysis of Concurrent Java Programs

Analysis time The programs were analyzed with the library files provided with the Java installation and any additional libraries required by the specific program. In general, the library code tends to be much larger than the actual program and it is possible to analyze the libraries and summarize them in appropriate files. We give two sets of analysis time - the first set gives the total analysis time including

Table 9.5: Data Dependence Analysis of Concurrent Java Programs

name	No. of edges			
	DD	ID	DD(weak)	ID(weak)
pc	24729	38	25528	38
multi	56874	3556	58378	3580
raytracer	167939	4564	170450	4564
slice	172800	5382	176092	5382
hanoi	223270	5101	229733	5101
volano	236878	0	240325	0
mandelbrot	327739	5826	342479	5826

analysis of all libraries and the second set gives the analysis time of the program when the standard Java library (rt.jar) has been pre-analyzed (the program specific libraries are re-analyzed every time). The time of analysis includes the time taken to build the control flow graph, the control dependence analysis and the sequential data dependence analysis. The time for the mapping algorithm and the interthread pointer analysis is also given. The time of analysis in seconds is given in Table 9.4.

Data dependence The data dependence analysis details are given in Table 9.5. We show the number of data dependence edges and the number of interference dependence edges. We investigated the effect of weak updates on heap allocated variables by forcing our implementation to perform only weak updates on heap allocated variables. Except for `multi` none of the programs showed any change in the number of interference dependence edges, but the number of data dependence edges increased by 1% to 5%.

We also investigated the effect of monitor analysis. Except for the dummy producer-consumer example used as an example in this dissertation, we did not find any improvement in precision to be gained by applying monitor analysis. However, this may have been due to a limitation in our implementation in which we applied monitor analysis only to synchronized methods and not to all synchronized statements (for simplicity of coding - this required less than 20 lines of extra code).

It is interesting to note that in `volano` although a number of threads are generated there are no interference dependence edges at all.

Table 9.6: Escape Analysis

name	% Marked GlobalEscape	% Interfering statements
pc	36.3	0.1
multi	35.9	0.6
raytracer	40.6	0.6
slice	40.9	0.9
hanoi	40.3	0.5
volano	32.2	0
mandelbrot	41.1	0.7

Escape analysis Table 9.6 gives the percentage of statements in the program that are marked *GlobalEscape* and the percentage of statements that may actually try to access a memory location that is also accessed by another parallel executing thread. The statements that are marked *GlobalEscape* are the ones that are determined to globally escape and these are the only statements that are analyzed during the interthread analysis phase. Although the number of nodes that are detected to escape globally is around 40%, the statements that actually contend for access to shared resources turns out to be less than 1% of the statements in our sample programs. We believe that this kind of analysis may improve the precision of tools that use escape analysis to eliminate unnecessary synchronization.

9.5 Slicing Concurrent Interprocedural Java Programs

We tested the context-insensitive and the context-sensitive algorithms on the multithreaded benchmark programs. Of these programs, we do not give statistics for `volano` since this program does not have any interference dependence edges and hence can be sliced as a sequential program.

For the slicing criterion, we chose every node that was the destination of an interference dependence edge as well as some other random nodes. The results are given for averages over all the slicing criterion as well as for the criterion node that generated some maximum or minimum condition.

Table 9.7: Build Time in seconds for Slicer

name	CFG + CD + DD + ID	Summary edges	iSCR + ISCR graph Topological Numbering
pc	1.42	0.54	0.01
multi	4.53	1.63	0.09
raytracer	55.35	11.31	1.93
slice	43.67	18.98	1.68
hanoi	108.49	88.48	3.65
mandelbrot	329.03	192.94	22.61

In Table 9.7 we give the time to build the various components of the threaded system dependence graph. The first column gives the time to parse the program and calculate the control, data and interference dependence edges. The second column gives the time to build the summary edges. The third column gives the time to build the ISCR graph and to generate the topological numbering for all the threads. Clearly, the time to build the ISCR graph and the topological numbering is a small fraction of the total time to build the threaded system dependence graph.

Performance Analysis In Table 9.8 we give the average and maximum time taken to slice a node for the context-insensitive algorithm (column "I"), the context-sensitive algorithm without optimizations (column "V") and the context-sensitive algorithm with optimizations (column "O"). The minimum time, in each case was too small to measure.

The context-sensitive algorithm without optimizations does not scale with program size and except for `pc`, `multi` and `raytracer` it was found to be impractical (*i.e.*, it took more than 20 minutes of CPU time to generate the slice for most nodes).

Another measurement of the performance of the algorithm is the number of nodes inserted into the worklist. In the context-insensitive algorithm some nodes get sliced twice as they get colored `phase2` and later `phase1`. Hence, the number of nodes handled by the worklist is greater than the total number of nodes in the slice. In the context-sensitive algorithm a node may be inserted into the worklist with potentially exponential number of tuples. In Table 9.9 we give the maximum number of times a single node is inserted into the worklist with a different tuple and the total number of nodes handled by the worklists for the context-insensitive

Table 9.8: Time to Slice

name		I	V	O
pc	average	0.0	0.0	0.0
	max	0.0	0.01	0.01
multi	average	0.01	0.33	0.03
	max	0.02	1.72	0.11
raytracer	average	0.03	6.7	0.35
	max	0.05	15.39	1.01
slice	average	0.3	-	0.52
	max	0.08	-	6.09
hanoi	average	0.08	-	3.3
	max	0.13	-	89.8
mandelbrot	average	0.15	-	36.8
	max	0.21	-	1030.9

Table 9.9: Worklist Analysis

name		max insertions per node	Total nodes in worklists	Actual nodes in slice	Ratio
pc	I	2	77	61	1.3
	V	5	218	77	2.8
	O	2	83	77	1.1
multi	I	2	5570	5398	1.03
	V	97	64330	1935	33.3
	O	23	5204	1935	2.7
raytracer	I	2	15269	13457	1.1
	V	189	61696	9542	6.5
	O	17	34843	9542	3.65
slice	I	2	18242	14279	1.3
	O	18	32338	9952	3.25
hanoi	I	2	31273	25305	1.2
	O	58	69290	13366	5.2
mandelbrot	I	2	50788	37539	1.4
	O	55	156558	23684	6.6

Table 9.10: Precision Analysis

name	Zhao (1)	I (2)	O (3)	(2) - (1)	(2) - (3)
pc	68	68	37	0% to 0%	40% to 48%
multi	2010	2010	711	0% to 0%	80% to 88%
raytracer	8055	8079	5574	0% to 5.5%	1.5% to 83.3%
slice	6607	6695	2911	0% to 4%	4.8% to 84.6%
hanoi	21848	22566	8194	0% to 6.5%	5.8% to 90%
mandelbrot	32628	34318	17903	0% to 8.2%	2.8% to 78.4%

algorithm (row "I"), for the context-sensitive algorithm without optimization (row "V") and for the context-sensitive algorithm with optimization (row "O"). The values shown are the worst case values. As a base for comparison, we also show the actual number of nodes in the slice. In the last column we show the ratio of nodes handled by the worklists to the actual number of nodes in the slice.

Although the total number of nodes handled by the worklists is, in general, a small multiple of the number of nodes in the slice, it is clear that some nodes are getting inserted into the worklists an exponential number of times. The improvement due to the optimizations is also clearly visible as a drop in the maximum insertions per node as well as a drop in the total worklist count.

Precision Analysis In Table 9.10 we show the average number of nodes sliced by Zhao's algorithm, the context-insensitive algorithm and the context-sensitive algorithm. Zhao's algorithm misses nodes (cf. Section 5.2 in some slices and in column 4 we give the range of difference between the number of nodes sliced by Zhao and the number of nodes sliced by the context-insensitive algorithm. For example, in the case of the program `hanoi` Zhao's algorithm generated between 0% to 6.5% less nodes than the context-insensitive algorithm, whereas in `pc` and `multi` there was no difference between the two algorithms. In column 5, we give the improvements to be gained by using a context-sensitive algorithm rather than a context-insensitive algorithm. For example, in `hanoi` the context-sensitive algorithm generated between 5.8% to 90% less nodes than the context-insensitive algorithm. Clearly, the gain in precision from context-insensitive to context-sensitive analysis can be worth the extra computing power.

Weak Updates Both the context-insensitive and the context-sensitive algorithm showed an increase of up to 6% in the number of nodes sliced when the data dependence algorithm applied weak updates to the heap variables as opposed to strong updates.

Remarks Zhao's algorithm was found to generate incorrect slices in many test cases. Our context-insensitive algorithm is very fast and requires little overhead in terms of memory (it does not require calculation of the ISCR graph or topological numbering) but is definitely less precise than the context-sensitive algorithm.

The context-sensitive algorithm has exponential complexity, but, in general, few nodes display this exponential behavior and most nodes get inserted into the worklist a small constant number of times. With the optimizations, the algorithm was found to be practical even for large benchmark programs. In our tests, we have analyzed the complete programs including all library methods (except native methods). Often we are not interested in slicing library methods. There are standard techniques [49] for analyzing sequential programs in the absence of library code. These can be easily extended to concurrent programs (but is beyond the scope of this thesis) and would considerably increase the speed of slicing. However, our aim was to see whether the algorithm is practical when applied to large, real world programs and we believe that it is indeed so.

In applications such as model checking [63] and formal verification where it is important to get as small a slice as possible, it may be well worth the additional computation to generate a context-sensitive slice.

9.6 Limitations of the Implementation

Our implementation handles parameter aliasing conservatively. An alias that holds at one call site holds at all call sites. Our algorithm detects the synchronization edges from a `notify` statement to a corresponding `wait` statement but we have not implemented possible improved precision to be gained by determining the partial order induced by the synchronization edges. We analyze monitors conservatively.

Chapter 10

CONCLUSION

Static slicing is a useful tool in analyzing, debugging and understanding concurrent programs. In this dissertation we have given algorithms for analyzing two different models of concurrent programs that use shared memory for communication between threads. The first concurrency model is the basic shared-memory model without any explicit synchronization, where concurrency is expressed by the classical `cobegin-coend` parallelism, and the second is the Java concurrency model.

10.1 Summary of Contributions

The main contribution of this dissertation are slicing algorithms for concurrent intraprocedural and interprocedural programs and a data dependence analysis algorithm for concurrent Java programs.

Slicing Concurrent Intraprocedural Programs: Previous solutions to slicing concurrent programs have been inadequate. For example, Krinke's solution [53] to slicing concurrent intraprocedural programs, though more precise than previous solutions, is imprecise in the presence of nested threads and threads nested within loops. We give a precise intraprocedural slicing algorithm for a basic shared-memory concurrency model, where parallelism is expressed by the classical `cobegin-coend` construct and there is no explicit synchronization. Though the complexity of this algorithm is exponential on the number of threads, a number of optimizations are suggested, using which we are able to get near linear behavior for many practical concurrent programs.

Slicing Concurrent Interprocedural Programs: The previous solution for slicing concurrent Java programs by Zhao [93] is not only incorrect (it can miss nodes that should belong to the slice) but also context-insensitive and hence imprecise. For the basic shared-memory concurrency model of concurrent programs, we give a correct but context-insensitive interprocedural slicing algorithm. The complexity of the algorithm is linear in the size of the graph.

In order to generate a context-sensitive slicing algorithm, we first solve the following path problem: Given an interprocedural control flow graph, G, and a set of nodes $n_1, n_2, \cdots, n_k$ belonging to G we give an incremental algorithm to determine whether there is a realizable path $n_1 \xrightarrow{+} n_2 \xrightarrow{+} \cdots \xrightarrow{+} n_k$. Our solution uses the interval-based approach and is an incremental algorithm.

Using this solution, we generate a context-sensitive slicing algorithm. The context-sensitive analysis handles nested threads with high precision but threads nested within loops are handled conservatively. The complexity of the context-sensitive algorithm is exponential in the number of threads.

Data Dependence Analysis of Concurrent Java Programs: Data dependence analysis is an essential component of slicing and previous solutions to data dependence analysis of Java programs are less efficient than the solution provided in this thesis. Unlike our solution which uses a sparse representation, existing solutions build points-to graphs at each program point [89, 25]. Also, previous solutions that perform inter-thread analysis handle pointer and alias analysis much more conservatively than our solution.

We give a new context-sensitive and path-sensitive solution to combined escape analysis, pointer analysis and data dependence analysis of concurrent Java programs. The algorithm builds the Static Single Assignment form of each procedure and generates SSA def-use edges. The analysis handles programs with threads, monitors and synchronization.

Slicing Concurrent Java Programs: The Java `fork/join` thread structure differs from the basic shared-memory concurrency model. We identify issues that pertain to the Java concurrency model, including the `wait/notify` synchronization and monitors. We discuss these issues and give solutions to some. We show how to slice concurrent Java programs subject to certain restrictions. Although the complexity of the algorithm is exponential in the number of threads, we give some optimizations which make the algorithm practical. We have tested the algorithm on standard Java benchmark programs.

Implementation: We have implemented and tested the algorithms on several student programs as well as standard Java benchmark programs and give performance statistics. As intraprocedural slicing requires inlining of procedures, the intraprocedural slicing algorithm was tested on comparatively small programs without recursion. The algorithm has been implemented for Java source code. The interprocedural slicing algorithm has been implemented for Java bytecode and has been tested on some very large programs.

10.2 Related Research

The past few years has seen a spate of activity in the analysis of Java programs. Bik *et al* [10, 11] have worked on a restructuring compiler that converts implicit parallelism in Java into explicit parallelism. This has the advantage of maintaining portability and derives a speedup in the execution time if the Java interpreter supports true parallel execution of threads. Clausen [28] has developed a bytecode optimizer that, however, supports only sequential Java programs.

Much work has been done in removing unnecessary synchronization in Java with the help of escape analysis [25, 4, 13, 77, 15]. Field analysis [41] is a technique that exploits the declared access restrictions placed on fields to determine properties that are useful in optimizations such as removal of run-time tests, compile time resolution of method calls, object inlining, removal of unnecessary synchronization and stack allocation.

Java requires array bounds checking. This inhibits many useful code optimizations such as partial redundancy elimination or instruction scheduling of memory operations since array bounds checks may raise exceptions and hence block code motion of instructions with side effects. Since it is not expressible at the bytecode level, array bounds checks can only be performed at run time. Bodik *et al* [14] have built an algorithm for eliminating array bounds checks. Flanagan *et al* [39] have built a tool for static race detection analysis of multithreaded Java programs based on a formal type system.

10.3 Limitations and Future Work

Limitations

Since Java is a well-structured language without unconditional `goto`s, we have assumed that the bytecode is also well-structured. However, post-optimization bytecode may not have a properly nested structure. In sequential slicing there are standard techniques to handle unconditional jumps [6, 1]. It remains to be seen whether these techniques can be directly applied to concurrent programs.

Our analysis does not handle many Java constructs such as timed `wait()` and timed `join()`. Also we have not yet built in the functionality to handle exceptions. This is an important aspect of Java analysis as many operations can result in exceptions. Choi *et al* [26] give an efficient modification of the control flow graph to support exceptional control flow. Also Sinha *et al* [82, 45] show how to modify the System Dependence Graph in order to slice interprocedural programs when called programs do not necessarily return to the call site, for example, in the case of a system exit. Extending these to concurrent programs are issues that we leave for future investigation.

Neither the algorithm nor the implementation deals with dynamic loading. It is assumed that all code is statically available. A call to a method in native code is treated as a single statement that both uses and modifies the parameters that are passed to that mathod call. This is conservative.

Future Work

Bytecode optimization In Java, optimizations are normally done in two places: (i) in the compiler; and (ii) in the runtime system. The compiler from Java source to bytecode can perform some large-scale optimizations, but access to the source is a necessity for doing this. The runtime system, whether it does JIT (Just In Time) compilation or interpreting, is limited by having little time to spare for optimizing, and thus can only do some local optimizations. A third possibility is to optimize on the Java bytecode. This has several advantages

- Access to source code is not required. In fact, it will be possible to optimize programs written in other languages if they can be compiled to bytecode.

- Machine independence is maintained.

- Inter-procedural analysis is possible for the user, so library variations are recognized. The user will be able to optimize with respect to his own libraries and still get valid results.

Under such circumstances, a tool that understands and analyzes Java bytecode proves to be very useful. Our static slicer works on Java bytecode. It parses the bytecode, builds the threaded control flow graph, determines the strongly connected components, the control dependences, the data dependences, the interference dependences and the synchronization dependences. With these analyses already in place, it should be possible to build a bytecode optimizer with optimizations for constant propagation, common subexpression elimination, dead code elimination, code motion etc. These are standard optimizations in sequential but not in concurrent programs.

Escape Analysis Our escape analysis algorithm can be used to remove unnecessary synchronizations. Acquiring and releasing locks is not required if the object does not escape its thread. Also objects that do not escape a method can be allocated on the stack instead of the heap. This reduces the garbage-collection overheads of the runtime system.

Shape Analysis Our pointer analysis algorithm creates one object per object creation site. Shape analysis techniques generate several objects per object creation site and cluster objects created at the same run-time location into summary nodes [79]. It should be interesting to see if our sparse pointer representation can be extended to solve shape analysis problems.

Pointer analysis has also been used to extract compact concurrency models of Java that can be used for finite state verification [30]. The combination of pointer analysis and control dependence has many possible applications.

Bibliography

[1] H. Agrawal. On slicing programs with jump statements. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 303–312, 1994.

[2] H. Agrawal, R. Demillo, and E. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static analysis for eliminating unnecessary synchronization from Java programs. In *Proceedings of the Conference on International Static Analysis Symposium*, 1999.

[5] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.

[6] T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging*, Linkoping, Sweden, May 1993.

[7] J. P. Banning. An efficient way to find the side effects of procedure calls and aliases of variables. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1979.

[8] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21:724–736, 1978.

[9] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *Proceedings of the 15th International Conference on Software Engineering*, pages 509–518. IEEE Computer Society Press, 1993.

[10] Aart J. C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency : Practice and Experience*, 9(6):579–619, June 1997.

[11] Aart J. C. Bik, Juan E. Villacis, and Dennis B. Gannon. javar: A prototype Java restructuring compiler. *Concurrency : Practice and Experience*, 9(11):1181–1191, November 1997.

[12] D. Binkley and K. Gallagher. Program slicing. *Advances in Computers*, 43, 1996.

[13] Bruno Blanchet. Escape analysis for object-oriented languages. Application to Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 20–34, 1999.

[14] Rastislav Bodik, Rajiv Gupta, and Vivek Sakar. ABCD: Eliminating array-bounds checks on demand. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 321–333, 2000.

[15] Jeff Bogda and Urs Holzle. Removing unnecessary synchronization in Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 35–46, 1999.

[16] Marc M. Brandis and Hanspeter Mossenbock. Single pass geneaiton of static single assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, 1994.

[17] Benjamin M. Brosgol. A comparison of the object-oriented features of Ada 95 and Java. In *Proceedings of the conference on TRI-Ada '97*, pages 213–229, November 1997.

[18] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[19] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1988.

[20] A. T. Chamillard. Improving static analysis accuracy on concurrent Ada programs: Complexity results and empirical findings. Technical Report 95-049, Dept. of Computer Science, University of Massachusetts at Amherst, 1995.

[21] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1990.

[22] R. Chatterjee, B.G.Ryder, and W.A.Landi. Relevant context inference. In *Proceedings of the Symposium on Principles of Programming Languages*, 1999.

[23] J. Cheng. Slicing concurrent programs. In *Automated and Algorithmic Debugging, 1st Intl. Workshop*, 1993. LNCS 749, 1993.

[24] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side efects. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1993.

[25] J-D. Choi, M. Gupta, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1999.

[26] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Program analysis for software tools and engineering (PASTE 99)*. ACMSOFT, 1999.

[27] Jyh-Herng Chow and Williams Ludwell III Harrison. Compile time analysis of parallel programs that share memory. In *Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*, pages 130–141. ACMSIGPLAN, January 1992.

[28] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency : Practice and Experience*, 9(11):1031–1045, November 1997.

[29] Cyrille Comar, Gary Dismukes, and Franco Gaspereroni. Targeting GNAT to the Java virtual machine. In *Proceedings of the conference on TRI-Ada '97*, pages 149–161, November 1997.

[30] James C. Corbett. Constructing compact models of concurrent Java programs. In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–10, March 1998.

[31] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing the static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[32] Stephan Diehl. A formal introduction to the compilation of Java. *Software - Practice and Experience*, 28(3):297–327, March 1998.

[33] E. Duesterwald, R. Gupta, and M. L. Soffa. Distributed slicing and partial re-execution for distributed programs. In *5th Workshop on Languages and Compilers for Parallel Computing*, 1992. LNCS 757, 1992.

[34] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACMSIGSOFT, 1994.

[35] Matthew B. Dywer. Data flow analysis frameworks for concurrent programs. Technical Report 95-062, Dept. of Computer Science, University of Massachusetts at Amherst, 1995.

[36] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1994.

[37] P. A. Emrath, S. Ghosh, and D. A. Padua. Detecting nondeterminacy in parallel programs. *IEEE Software*, pages 69–77, January 1992.

[38] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[39] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[40] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transaction on Software Engineering*, 17(8):751–761, 1991.

[41] Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Field analysis: getting useful and low-cost interprocedural information. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 334–344, 1999.

[42] R. Ghiya and J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, 1996.

[43] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–168. ACMSIGPLAN, May 1993.

[44] M. Harman and S. Danicic. Using program slicing to simplify testing. *Software Testing Verification and Reliability*, pages 143–162, September 1995.

[45] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis*, pages 11–20, March 1998.

[46] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *Proceedings of the Conference on International Static Analysis Symposium*, 1999.

[47] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, July 1997.

[48] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1997.

[49] S. Horwitz, T.Reps, and D.Binkley. Interprocedural slicing using dependence graphs. *Transactions on Programming Languages and Systems*, 12:26–60, Jan 1990.

[50] Richard Johnson and Keshav Pingali. Dependence based program analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 78–89, 1993.

[51] J. Knoop and B. Steffen. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.

[52] Jens Knoop and Bernhard Steffen. Code motion for explicitly parallel programs. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 13–24. ACMSIGPLAN, May 1999.

[53] Jens Krinke. Static slicing of threaded programs. In *Program analysis for software tools and engineering (PASTE 98)*, pages 35–42. ACMSOFT, 1998.

[54] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel programs with explicit synchronization. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–204, 1995.

[55] D. Kuck, R. Kuhn, D. Padua, D. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1981.

[56] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1992.

[57] Loren D. Larsen and Mary Jean Harrold. Slicing object-oriented software. Technical Report 95-103, Department of Computer Science, Clemson University, March 1995.

[58] J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. ACMSIGPLAN, May 1999.

[59] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[60] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole program analysis. In *Proceedings of the 7th European Engineering Conference*, 1999.

[61] Douglas Long and Lori A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the ACM SIGSOFT Symposium on Testing Analysis and Verification (TAV4)*, pages 21–34. ACMSIGSOFT, October 1991.

[62] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[63] L. Millet and T. Teitelbaum. Slicing Promela and its application to model checking, simulation, and protocol understanding. In *Proceedings of the* 4^{th} *Workshop on Automata Theoretic Verification with the SPIN Model Checker*, November 1998.

[64] L. Millet and T. Teitelbaum. A case for channel dependence analysis:Slicing Promela. In *Proceedings of the 1999 International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'99)*, May 1999.

[65] M. G. Nanda, P. Bhaduri, S. Oberoi, and A. Sanyal. An application of compiler technology to the year 2000 problem. *Software - Practice and Experience*, April 1999.

[66] M. G. Nanda and S. Ramesh. Data dependence analysis of multithreaded Java programs. Submitted for publication.

[67] M. G. Nanda and S. Ramesh. Slicing concurrent interprocedural programs. Submitted for publication.

[68] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proc. of the International Symposium on Software Testing and Analysis*, 2000.

[69] D. Novillo, R. Unrau, and J. Schaeffer. Concurrent ssa form in the presence of mutual exclusion. In *Proceedings of the International Conference on Parallel Processing*, 1998.

[70] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACMSIGSOFT, 1984.

[71] Keshav Pingali and Gianfranco Bilardi. Optimal control dependence computation and the Roman Chariots Problem. *ACM Transactions on Programming Languages and Systems*, 19(3):462–491, May 1997.

[72] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

[73] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.

[74] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1999.

[75] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering*, pages 11–20, New Orleans, LA, Dec 1994.

[76] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report TR 94-14, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark, 1994.

[77] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

[78] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1999.

[79] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.

[80] M. Shapiro and S.Horwitz. The effects of the precision of pointer analysis. In *LNCS 1302, 4th International Symposium on Static Analysis*, 1997.

[81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S.Muchnick and N.D.Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[82] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the International Conference on Software Engineering*, pages 432–441, 1999.

[83] V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1995.

[84] Vugranam C. Sreedhar. *Efficient Program Analysis Using DJ Graphs.* PhD thesis, School of Computer Science, McGill University, Canada, 1995.

[85] X. Tang, R. Ghiya, L. J. Hendren, and G. R. Gao. Heap analysis and optimizations for threaded programs. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, 1997.

[86] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–181, 1995.

[87] J. Vollmer. Data flow analysis of parallel programs. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques (PACT'95)*, pages 168–177, 1995.

[88] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.

[89] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1999.

[90] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1995.

[91] Michal Young, Richard N. Taylor, David L. Levine, Kari A. Nies, and Debra Brodbeck. A concurrency analysis tool suite for Ada programs: rationale, design, and preliminary experience. *ACM Transactions on Software Engineering and Methodology*, 4(1):65–106, January 1995.

[92] J. Zhao. Multithreaded dependence graphs for concurrent Java programs. In *Proceedings of the 1999 International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'99)*, May 1999.

[93] J. Zhao. Slicing concurrent Java programs. In *Proceedings of the IEEE International Workshop on Program Comprehension*, May 1999.

[94] J. Zhao, J. Cheng, and K. Ushijima. A dependence-based program representation for concurrent object-oriented software maintenance. In *Proceedings of 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, pages 60–66, Florence, Italy, March 1998. IEEE Computer Society Press.

[95] Y. Zhu and L. J. Hendren. Communication optimization for parallel C programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 199–211, 1998.

SUMMARY

Slicing is a program transformation that has successfully been used in debugging, testing, program maintenance, reverse engineering, Y2K compliance transformations and applications that involve understanding program behavior. The *slice* of a program with respect to a program point p and a variable x, consists of all statements and predicates of the program that might affect the value of x at point p [88].

For sequential programs, the definition of a slice is given in terms of the standard program dependence graph [38] representation of programs where the nodes, representing assignments and conditions in the program, are connected by control and data dependence edges; the PDG of a program can be obtained from its control flow graph by removing control flow edges and adding the data and control dependence edges. The slice can be defined with respect to a given node in the PDG as the set of all the nodes on which the given node is directly or transitively dependent. Thus, given the PDG, the slice can be computed by a simple reachability algorithm [38].

In contrast, mere reachability produces imprecise slices in interprocedural programs [49]. This is because not all paths in an interprocedural program are *realizable* paths. A *realizable* path is one that corresponds to a legal call/return sequence wherein a procedure returns to the site from where it was invoked.

To slice concurrent programs, we need to introduce a new kind of dependence called *interference dependence* [53]. Interference dependence is a form of data dependence across parallel executing threads where a node S_2 defines a variable v which is used by a node S_1, and S_1 and S_2 execute in parallel. In concurrent intraprocedural programs also a simple reachability algorithm produces imprecise slices since interference dependence is not transitive.

Slicing concurrent inter-procedural programs raises the problems of both realizable paths and intransitivity of interference dependence.

Overview of the thesis

The aim of this thesis was to develop a correct and precise slicing algorithm for concurrent Java programs. Towards this end, we have identified many of the issues and challenges and found solutions to many of them. We have also implemented the algorithms on Java bytecode.

Slicing Concurrent Intraprocedural Programs: Previous solutions to slicing concurrent programs have been inadequate. We give a precise intraprocedural slicing algorithm for a basic shared-memory concurrency model, where parallelism is expressed by the classical `cobegin-coend` construct and there is no explicit synchronization. Though the complexity of this algorithm is exponential on the number of threads, a number of optimizations are suggested, using which we are able to get near linear behavior for many practical concurrent programs.

Slicing Concurrent Interprocedural Programs: The previous solution for slicing concurrent Java programs by Zhao [93] is not only incorrect (it can miss nodes that should belong to the slice) but also context-insensitive and hence imprecise. For the basic shared-memory concurrency model of concurrent programs, we give a correct but context-insensitive interprocedural slicing algorithm. The complexity of the algorithm is linear in the size of the graph.

In order to generate a context-sensitive slicing algorithm, we first solve the following path problem: Given an interprocedural control flow graph, G, and a set of nodes n_1, n_2, $\cdots$, n_k belonging to G we give an incremental algorithm to determine whether there is a realizable path $n_1 \xrightarrow{+} n_2 \xrightarrow{+} \cdots \xrightarrow{+} n_k$.

Using this solution, we generate a context-sensitive slicing algorithm. The context-sensitive analysis handles nested threads with high precision but threads nested within loops are handled conservatively. The complexity of the context-sensitive algorithm is exponential in the number of threads.

Data Dependence Analysis of Concurrent Java Programs: Data dependence analysis is an essential component of slicing and previous solutions to data dependence analysis of Java programs are less efficient than the solution provided in this thesis. Unlike our solution which uses a sparse representation, existing solutions build points-to graphs at each program point [89, 25]. Also, previous solutions that perform inter-thread analysis handle pointer and alias analysis much more conservatively than our solution.

We give a new context-sensitive and path-sensitive solution to combined escape analysis, pointer analysis and data dependence analysis of concurrent Java programs. The algorithm builds the Static Single Assignment form of each procedure and generates SSA def-use edges. The analysis handles programs with threads, monitors and synchronization.

Slicing Concurrent Java Programs: The Java `fork/join` thread structure differs from the basic shared-memory concurrency model. We identify issues that pertain to the Java concurrency model, including the `wait/notify` synchronization and monitors. We discuss these issues and give solutions to some. We show how to slice concurrent Java programs subject to certain restrictions. Although the complexity of the algorithm is exponential in the number of threads, we give some optimizations which make the algorithm practical. We have tested the algorithm on standard Java benchmark programs.

Implementation: We have implemented our algorithms on Java bytecode and tested it on standard Java benchmark programs, as well as several student programs.

In summary, the contributions of the thesis are

- A precise intraprocedural slicing algorithm.
- Given an interprocedural control flow graph, G, and a set of nodes n_1, n_2, $\cdots$, n_k belonging to G we give an algorithm to determine whether there is a realizable path $n_1 \xrightarrow{+} n_2 \xrightarrow{+} \cdots \xrightarrow{+} n_k$.
- A correct but context-insensitive interprocedural slicing algorithm and a context-sensitive algorithm for the basic shared-memory concurrency model of concurrent programs.
- A new context-sensitive and path-sensitive algorithm for combined escape analysis, pointer analysis and data dependence analysis of concurrent Java programs.

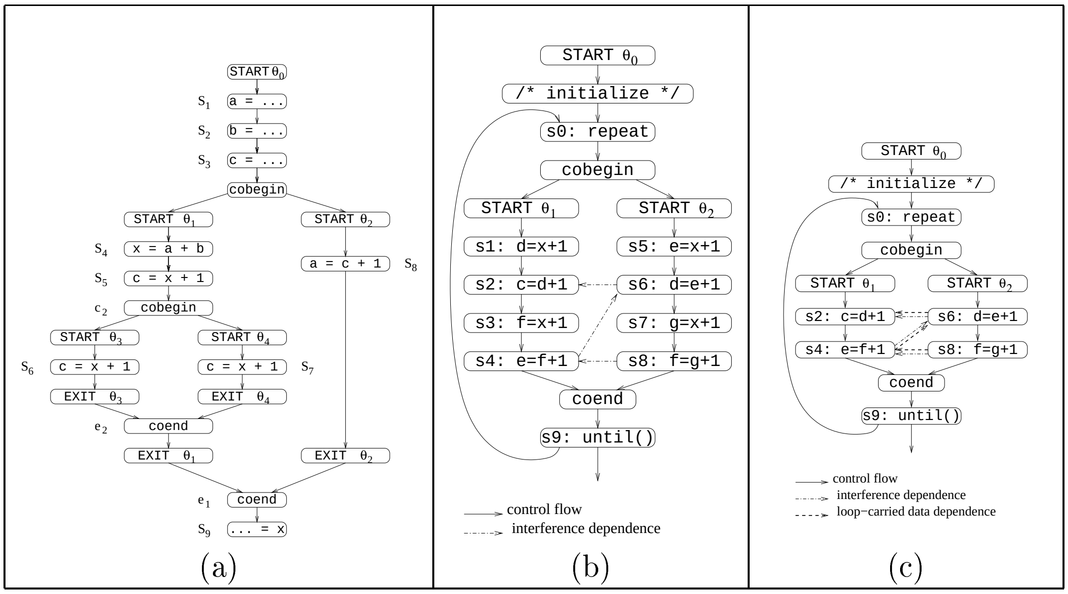


Figure 10.1: Threaded control flow graphs with (a) nested threads and (b), (c) threads nested within loops

- There are many issues in slicing Java programs over and above the basic requirements of determining data and control dependences. We discuss these issues and give solutions to some.
- We have implemented our algorithms on Java bytecode.

Slicing Concurrent Intraprocedural Programs

Consider the simplest model of concurrent programs that consist of processes or threads which interact via shared variables with atomic memory reads and writes. Threads share the same address space and execute concurrently with each other with complete interleaving semantics. There is no explicit synchronization between the threads. At the language level the classical `cobegin−coend` construct is used to express the parallelism. An abstract representation of the concurrent programs is the threaded control flow graph (TCFG) as illustrated in Figure 10.1.

Threaded PDG (TPDG) is an extension of PDGs to concurrent programs [53]. TPDGs capture, besides data and control dependencies between nodes in a single thread, additional dependencies arising out of *interference* between concurrent execution of threads. Interference dependence is given by

Definition 9 *A node S_1 is interference dependent on a node S_2 if S_2 defines a variable v, S_1 uses the variable v and S_1 and S_2 execute in parallel.*

Unlike data dependence and control dependence, interference dependence is not transitive. For example, in Figure 10.1(a), S_4 is interference dependent on S_8 and S_8 is interference dependent on S_5. Yet it is not possible for S_4 to be dependent on S_5 in any execution of the program. Krinke [53] made some preliminary attempts towards slicing concurrent programs. In order to compute a precise slice in the presence of interference dependence, he tries to capture the concept of a proper execution path. He introduced the notion of a *threaded witness* which he defines as follows: a *threaded witness* in a TCFG, G, is an ordered sequence of nodes $\langle n_1, n_2, \ldots, n_k \rangle$ in G such that for any two nodes n_i and $n_j, i < j$ in the sequence, if n_i and n_j belong to the same thread then there is a path in G from n_i to n_j.

Using this definition, he gives a precise slicing algorithm for concurrent programs. However, his solution fails to give precise slices in the presence of nested threads and threads nested within loops.

In Figure 10.1(a), according to Krinke's definition $\langle S_6, S_5, S_3 \rangle$ is a valid threaded witness. As a result his slicing algorithm would (correctly) exclude S_5 but would include both S_6 and S_7 in the slice of S_4 (which is imprecise).

Further, consider the program in Figure 10.1(b). S_2 is interference dependent on S_6 which is interference dependent on S_4. But it is not possible for S_2 to be transitively dependent on S_4 as the definition at S_4 is killed by the definition at S_5 and the definition at S_6 is killed by the definition at S_1. However, by Krinke's definition $\langle S_4, S_6, S_2 \rangle$ is a valid threaded witness as there is a path from S_4 to S_2 induced by the loop and hence his algorithm would (*imprecisely*) include both S_4 and S_8 in the slice of S_2. Figure 10.1(c) is obtained by dropping four nodes from Figure 10.1(b), yet in Figure 10.1(c), it would appear to be correct to accept the transitive dependence from S_2 to S_6 to S_4.

Our Solution

Clearly, Krinke's definition is not strong enough to capture all possible program constructs. So we define

Definition 10 *A trace in the TCFG, G, of a concurrent program, is an ordered set of nodes* $\langle n_1, \cdots, n_k \rangle$ *that represents a valid execution path in* G.

Definition 11 *The trace set of* G*, denoted by* $Tr(G)$ *is the set of all traces in* G.

Given the set of traces, $Tr(G)$ we define

Definition 12 *A* **trace witness** *in* G *is an ordered sequence of nodes* $\langle n_1, n_2, \ldots, n_k \rangle$ *in* G *such that it forms a subsequence of some trace in the set* $Tr(G)$.

In concurrent programs it is necessary to differentiate between loop-independent and loop-carried data dependence when the loop encloses a complete `cobegin-coend` construct. In Figure 10.1(c), it is clear that there is an interference dependence from S_6 to S_2. However, if we unfold the loop we see that there is *also* a loop-carried data dependence from S_6 to S_2. In contrast, in Figure 10.1(b), there is no loop-carried data dependence from S_6 to S_2. Although *lexically* two nodes S_i in thread θ_i and S_j in thread θ_j may execute in parallel, a loop-carried dependence from S_i to S_j refers to *sequentially* executing instances of the two nodes. Given the TCFG, G of a concurrent program, the corresponding TPDG has the following set of edges: E_{dd}, E_{cd} and E_{id} represent the set of data dependence, control dependence and interference dependence edges respectively and E^l_{dd} represents the set of loop-carried data dependence edges induced by the loop l.

Given the TPDG, the slicing criterion is any node in the TPDG.

Definition 13 *The* **slice** *of a TPDG w.r.t. the criterion* p*, denoted by,* $\mathcal{S}(p)$*, consists of all nodes* q *on which* p *transitively depends and there exists a trace witness in* G *for each such* q*. More formally,*

$$\begin{aligned}\mathcal{S}(p) = \{q \quad | \quad & \mathcal{P} =< n_1, \ldots, n_k >, \\ & q = n_1 \xrightarrow{d_1} \ldots \xrightarrow{d_{k-1}} n_k = p, \\ & \mathit{for}\ 1 \leq i < k, \\ & d_i \in \{cd, dd, id\} \cup \{dd_l | l \ \mathit{a\ loop\ in}\ G\} \\ & \mathcal{P} \text{ is a trace witness in } \ G\}\end{aligned}$$

Using these concepts we develop a precise slicing algorithm for concurrent programs [68]. The algorithm is a restricted form of reachability analysis where the analysis carries some additional information on the state of each thread thus ensuring that only trace witnesses are traversed. Our solution consists of the following components:

- An algorithm to build a threaded program dependence graph that differentiates between loop-independent and loop-carried data dependence edges.

- A precise definition of slicing that is applicable to concurrent programs with arbitrary nesting of parallelism and loops.

- A slicing algorithm and its proof of correctness.

- Although the algorithm has a complexity that is exponential in the number of threads, we give a number of optimizations on the basic slicing algorithm that result in an efficient implementation. In practice, we get linear to quadratic complexity.

Slicing Concurrent Interprocedural Programs

An interprocedural concurrent program is represented by an *interprocedural threaded control flow graph (ITCFG)*. A *threaded system dependence graph* (TSDG) is similar to the system dependence graph (SDG) of sequential programs. In addition to the standard edges found in an SDG, the TSDG has interference dependence edges. The complete set of edges, E in the TSDG are the data dependence, control dependence, interference dependence, call, summary, parameter-in and parameter-out edges which are denoted by E_{dd}, E_{cd}, E_{id}, E_c, E_s, E_{pi} and E_{po} respectively. The slicing criterion is defined as a node in the TSDG.

There are two main issues here

1. **Correctness:** A slice is correct if it includes, at least, all the nodes that affect the criterion. A slice is *incorrect* if it excludes some nodes that should be in the slice.

2. **Precision:** Given two correct slices, one slice is more precise than the other if it contains less nodes.

Ideally, one would like to compute the correct slice that is the most precise. But this is, in general, not computable. Hence, we try to compute a slice that is correct and as precise as possible. A simple two-phase algorithm with extensions for interference dependence edges as the one proposed by Zhao [93] is both incorrect and imprecise.

```
cobegin {
  /* START θ1 */
  call P1()
  print x
}{
  /* START θ2 */
  y=10
  z=20
  call P2()
}

procedure P1
  a=0
  x=a

procedure P2
  a=y
  a=z
```

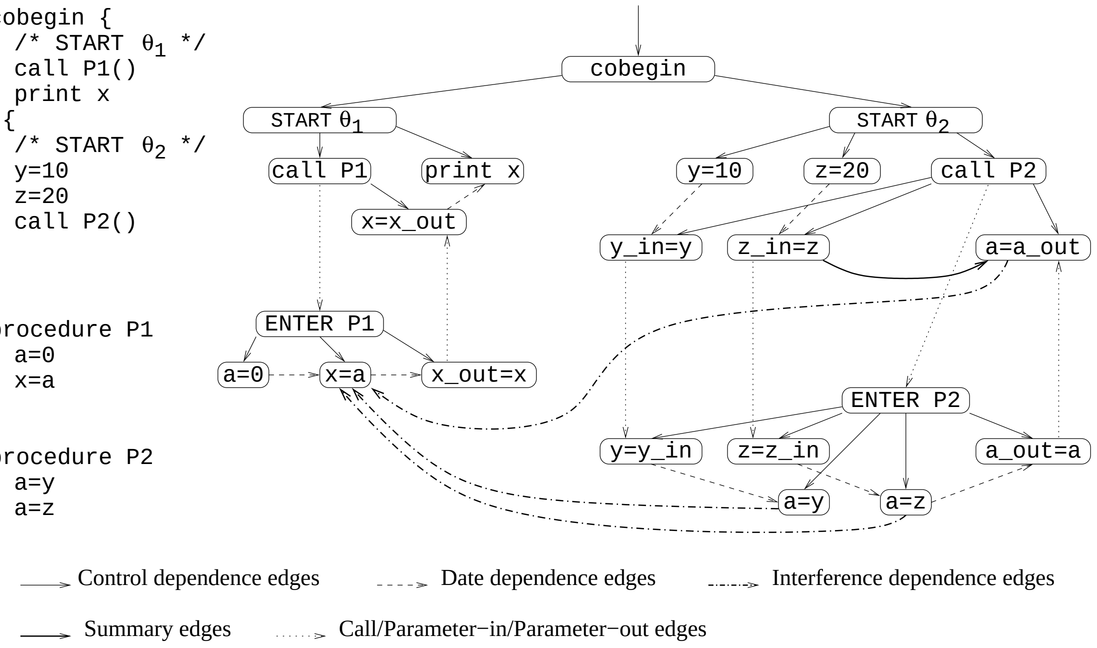


Figure 10.2: Example showing incorrectness of the two-phase algorithm

Correctness Issues

In the example in Figure 10.2 let us apply the two-phase algorithm to the node `print x`. In Phase 1, `print x` is data dependent on `x=x_out` which is parameter-out dependent on `x_out=x` in procedure `P1`.

In Phase 2, `x_out=x` is data dependent on `x=a`. `x=a` is data dependent on `a=0`, interference dependent on `a=y` and `a=z` in procedure `P2` in thread θ_2 and interference dependent on `a=a_out` in θ_2. Of these, `a=y` is data dependent on `y=y_in` which is parameter-in dependent on `y_in=y`. Since the algorithm is in Phase 2, `y_in=y` will not be added to the slice. As a result `y_in=y` and nodes that are transitively dependent on it (`y=10`) will never get added to the slice.

Hence the two-phase algorithm gives an incorrect slice.

Inter-thread Summary Edges Another problem with applying the two-phase algorithm directly is that the summary edges do not adequately represent transitive dependencies in the presence of threads since there may be transitive dependencies that cross thread boundaries.

Consider the slice of K_{12} in Figure 10.3. In Phase 1, we find that K_{12} is data dependent on K_{11} which is summary dependent on K_7 and parameter-out dependent on K_{26}. In Phase 2, we trace the following dependencies backwards: $K_{26} \leftarrow K_{25} \leftarrow N_4 \leftarrow N_3 \leftarrow N_8 \leftarrow N_7 \leftarrow K_{21} \leftarrow K_{19}$. K_{19} is parameter-in dependent on K_8 but since the algorithm is in Phase 2, K_8 will not get added to the slice and nor will any of the nodes that it depends on. Similarly, K_9 and its dependencies will also not get added to the slice.

Here the problem is that some transitive dependencies are missing. There is a transitive dependence from the formal-in node $c = c_{in}$ to the formal-out node $c_{out} = c$ and from $d_{in} = b$ to $a = c_{out}$. These transitive dependencies cannot be detected by the standard intra-thread summary edge algorithm [49].

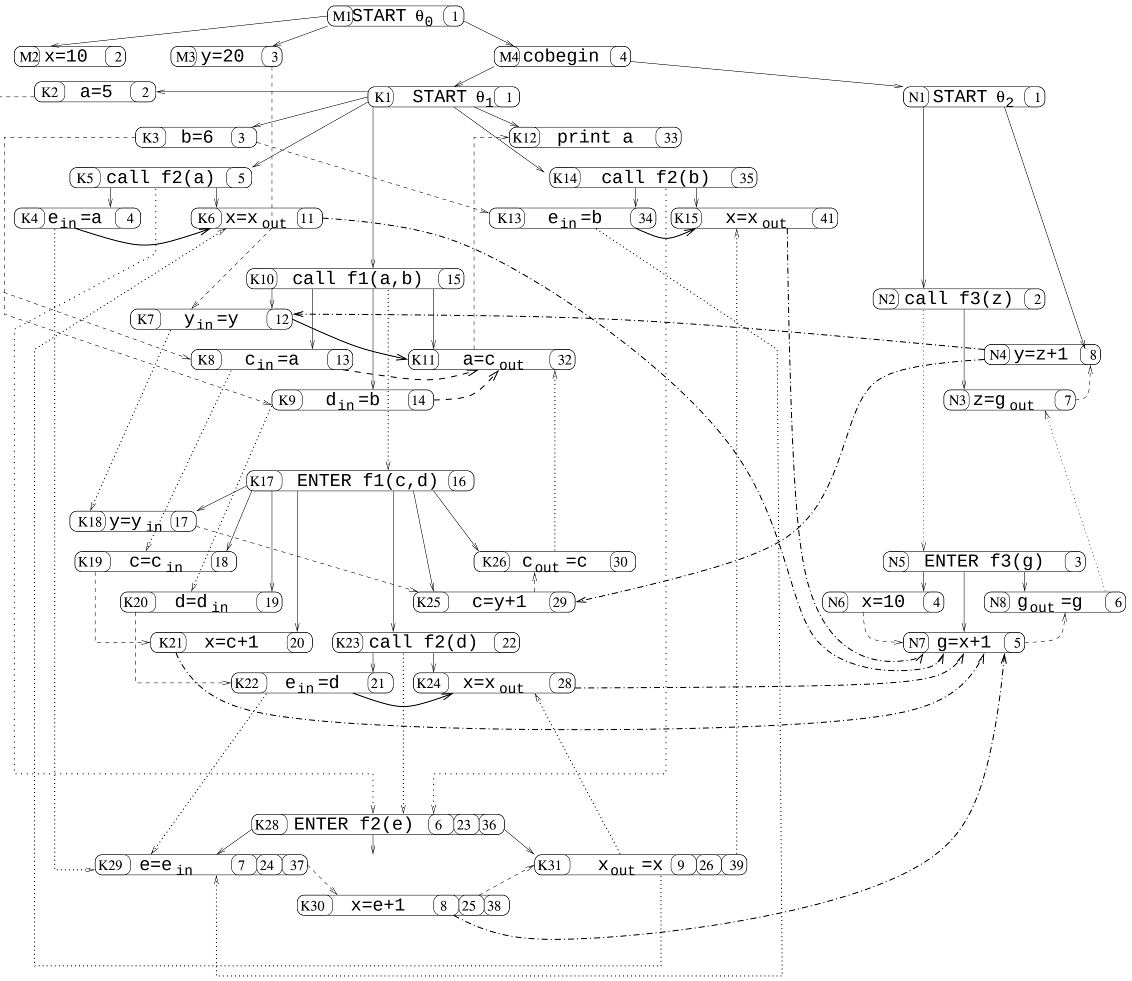
Precision Issues

Even with a more precise computation of summary edges, Zhao's algorithm remains imprecise and context-insensitive. In Figure 10.4 we show the TSDG of a concurrent interprocedural program. In Figure 10.4, m_6 is parameter-out dependent on m_{21} in procedure `f1`. Only the call at m_5 should be added to the slice, but a simple two-phase algorithm picks up the call to `f1` at m_{11} also ($m_{21} \leftarrow m_{19} \leftarrow n_3 \leftarrow m_{13} \leftarrow m_{11}$).

It might appear that a naive marriage of Krinke's algorithm and a corrected version of Zhao's algorithm would generate precise context-sensitive slices. However, this is not the case because unlike intraprocedural programs, paths in interprocedural programs are not transitive.

Intransitivity of Interprocedural Paths Determining whether there is a realizable path between any *two* nodes in the interprocedural graph of a *sequential* program is possible using standard techniques of data flow analysis. Let the relation $Reach(i, j)$ be **true** if there is a realizable path from i to j and **false** otherwise. For example, in Figure 10.4, $Reach(m_8, m_{20})$ and $Reach(m_{20}, m_{19})$ are true but $Reach(m_{11}, m_8)$ is false. In intraprocedural programs, the composition of *Reach* is a realizable path. In interprocedural programs the composition of *Reach* may not be a realizable path. For example, $Reach(m_8, m_{20}) \circ Reach(m_{20}, m_{19})$ does not imply that there is a realizable path through $\langle m_8, m_{20}, m_{19} \rangle$.

As a consequence, a simple extension of Krinke's algorithm to concurrent inter-




```
x=10                proc f1(c,d)
y=20                   x=c+1
cobegin {              call f2(d)
    a=5                c=y+1
    b=6
    call f2(a)
    call f1(a,b)     proc f2(e)
    print a            x=e+1
    call f2(b)
}{
    call f3(z)       proc f3(g)
    y=z+1              x=10
}                      g=x+1
```

Mι : ISCR number in θ_0

Kι : ISCR number in θ_1

Nι : ISCR number in θ_2

(ι) : topological number

- control dependence
- data dependence
- summary
- interference dependence
- call/parameter−in/ parameter−out

- inter−thread summary edges

Figure 10.3: Illustrating inter-thread summary edges

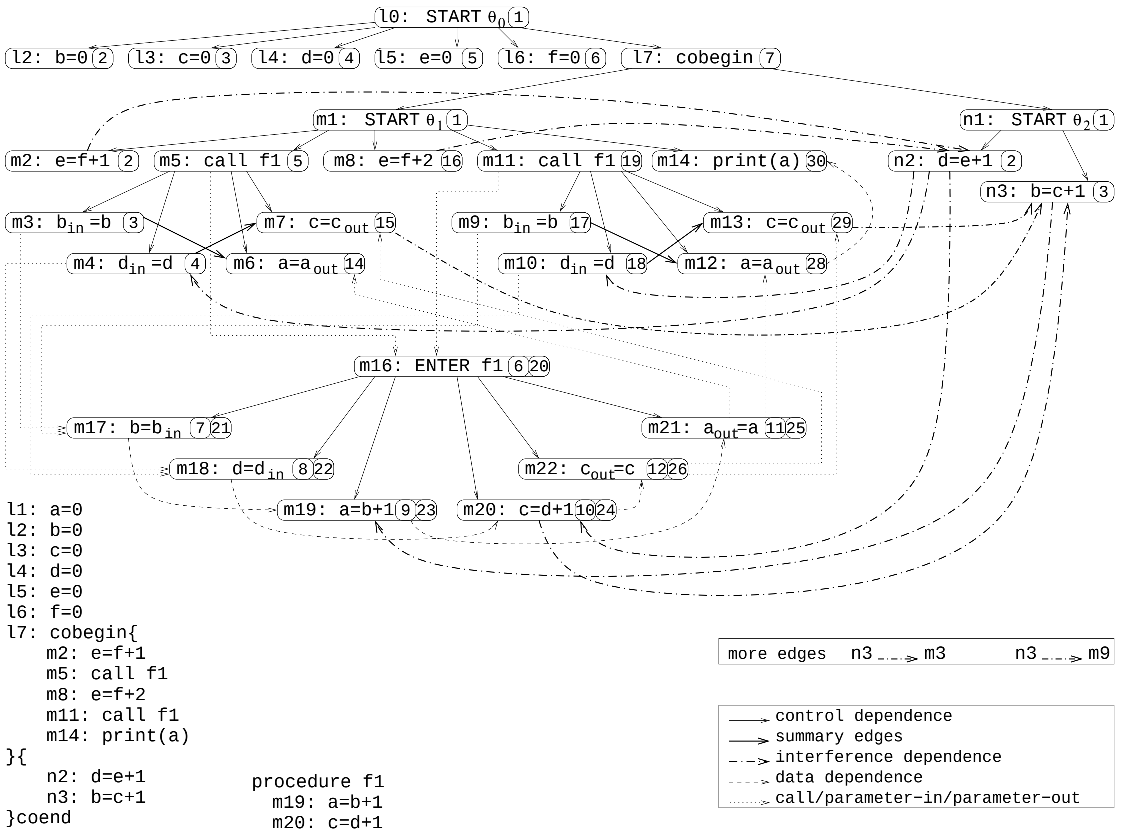


Figure 10.4: An example depicting intransitivity of paths

procedural programs generates context-insensitive slices.

Our Solution

We first generate a slicing algorithm for concurrent interprocedural programs which is correct though not context-sensitive. This algorithm is very efficient and has a complexity that is linear in the size of the TSDG [67].

In order to generate context-sensitive slices, we need to solve a path problem that is defined as follows: *Given an interprocedural control flow graph, G, and a set of nodes* n_1, n_2, $\cdots$, n_k *belonging to G we wish to find out if there is a realizable path* $n_1 \xrightarrow{+} n_2 \xrightarrow{+} \cdots \xrightarrow{+} n_k$. We term this problem the "*Realizable Path*" problem.

This realizable path problem differs from other realizable paths problems in literature. In *context-sensitive* interprocedural data flow analysis realizable paths are identified using some technique such as the matched parenthesis technique. Then data flow analysis is performed along only valid paths. Our realizable path problem is the converse of this situation. Here we are given a set of nodes and we wish to identify whether they form a part of a realizable path. A naive solution would generate all possible realizable paths in the graph using some technique such as the matched-parenthesis technique and then check if the given set of nodes forms a subsequence of any one of the realizable paths.

We solve the Realizable Path problem and use it to generate a slicing algorithm that is both correct and context-sensitive. This algorithm gives precise slices even in the presence of nested threads but does not scale to programs with threads nested within loops. In the case of threads nested within loops, the algorithm generates conservative slices. The complexity of the algorithm is exponential in the number of threads. The contributions can be summarized as follows:

- We give a *correct* slicing algorithm for concurrent interprocedural programs. The complexity of the algorithm is linear in the size of the TSDG.
- Using the standard interval-based approach to data flow analysis we generate an incremental solution to the Realizable Path problem.
- We give a correct and precise slicing algorithm for concurrent interprocedural programs. This algorithm is exponential in the number of threads in the program.

Slicing Concurrent Java Programs

The solutions proposed in the previous section form the first step towards slicing of Java programs. These solutions cannot be directly applied as the `fork/join` thread structure is different from the `cobegin-coend` construct. In addition, Java supports `wait/notify` synchronization that adds to the complexity of the analysis.

Data dependence: In Java all objects are allocated from a garbage-collected heap. Data dependence analysis for scalar variables can be computed using a reaching definition analysis. Determining data dependence in the presence of pointers is comparatively complicated as different access paths may refer to the same memory location as shown below.

S_1: $a =$ new cl

S_2: $a.next =$ new cl

S_3: $b = a$

S_4: $b.next =$ new cl

S_5: $x = a.next$

For data dependence analysis of local variables (a, b and x in the example) we can still use reaching definition analysis. However, for field variables, ($a.next$, $b.next$ in the example) a data dependence analysis based on the simple names of access paths would generate an incorrect data dependence from S_2 to S_5 and would be unable to detect the real data dependence from S_4 to S_5. Clearly, we need to apply some form of pointer analysis before or in conjunction with the data dependence analysis.

Interference dependence: There are two main issues in determining interference dependences

1. We need to determine whether two nodes may execute in parallel.

2. We need to do pointer analysis to determine whether two nodes that execute in parallel refer to the same location.

Determining whether two nodes may execute in parallel is simple in the basic shared-memory concurrency model but not so straightforward in the Java concurrency model. This is further complicated by the presence of `wait/notify` synchronization. Pointer analysis needs to be applied across threads and must take into account the mutual exclusion imposed by monitor sections.

Control dependence: There are standard techniques for determining control dependences in both intraprocedural programs [38] and in interprocedural programs [49]. These techniques have been extended to programs with unconditional jumps, procedures with `exit` statements and `wait/notify` synchronization. Control dependence needs to be analyzed differently in the presence of exceptions too.

Our Solution

Our solution has the following features:

- An analysis method to determine whether two nodes may execute in parallel, in the Java concurrency model.

- An algorithm for data dependence analysis of Java programs that uses a sparse representation [66], unlike previous solutions [89, 25] that build points-to graphs at each program point. Our solution is both more precise and more efficient than the previous solutions and combines escape analysis, pointer analysis and data dependence analysis while generating the useful Static Single Assignment form. Our solution also performs inter-thread analysis and handles monitors and synchronization.

- Adaptation of the interprocedural slicing algorithm to the Java concurrency model.

Implementation

One of the major contributions of this thesis is an implementation of a slicer for Java bytecode. We have implemented both the intraprocedural and interprocedural algorithms. The intraprocedural algorithm inlines each procedure call and hence cannot handle recursive programs. For interprocedural analysis, we implemented both the context-insensitive and the context-sensitive algorithms. For the context-sensitive solution we have implemented optimizations similar to the intraprocedural optimizations that greatly improve the performance of the algorithm. We tested our algorithm on a number of Java benchmark programs as well as several programs written by students and found them to be effective. We give performance statistics. Limitations of our implementation include: We handle exceptions conservatively and we do not handle timed `wait()` and `join()`.

Index