SCALABLE AND ACCURATE APPROACHES FOR PROGRAM DEPENDENCE ANALYSIS, SLICING, AND VERIFICATION OF CONCURRENT OBJECT ORIENTED PROGRAMS

by

VENKATESH PRASAD RANGANATH

B.E., Bangalore University, 1997
M.S., Kansas State University, 2002

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the

requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2006

ABSTRACT

With the advent of multi-core processors and rich language support for concurrency, the paradigm of concurrent programming has arrived; however, the cost of developing and maintaining concurrent programs is still high. Simultaneously, the increase in social ubiquity of computing is reducing the "time-to-market" factor while demanding stronger correctness requirements. These effects are amplified with ever-growing size of software systems. Consequently, there is (will be) a rise in the demand for scalable and accurate techniques to enable faster development and maintenance of correct large scale concurrent software.

This dissertation presents a collection of scalable and accurate approaches to tackle the above situation. Primarily, *the approaches are focused on discovering dependences (relations) between various parts of the software/program and leveraging the dependences to improve maintenance and development tasks via program slicing (comprehension) and verification.*

Briefly, the proposed approaches are embodied in the following specific contributions:

1. New trace-based foundation for control dependences.
2. An equivalence class based analysis to efficiently and accurately calculate escape information and intra- and inter-thread dependences.
3. A new parametric data flow style slicing algorithm with various extensions to uniformly and easily realize and reason about most existing forms of static sequential and concurrent slicing.
4. A new generic notion of property/trace sensitivity to represent and reason about richer forms of context sensitivity.
5. Program dependence based partial order reduction techniques to enable efficient and accurate state space exploration in both static and dynamic mode.

In an attempt to simplify the approaches, they have been based on the basic concepts/ideas of the affected techniques (e.g. program slicing is a rooted transitive closure of dependence relation). As trace-based reasoning is well suited for concurrent systems, an attempt has been made to explore trace-based reasoning wherever possible.

While providing a rigorous theoretical presentation of these techniques, this effort also validates the techniques by implementing them in a robust tool framework called Indus (available from http://indus.projects.cis.ksu.edu) and then providing experimental results that demonstrate the effectiveness of the techniques on various concurrent applications.

Given the current trend towards concurrent programming and social ubiquity of computing, the approaches proposed in this dissertation provide a foundation for collectively attacking scalability, accuracy, and soundness challenges in current and emerging systems.

SCALABLE AND ACCURATE APPROACHES FOR PROGRAM DEPENDENCE ANALYSIS, SLICING, AND VERIFICATION OF CONCURRENT OBJECT ORIENTED PROGRAMS

by

VENKATESH PRASAD RANGANATH

B.E., Bangalore University, 1997
M.S., Kansas State University, 2002

A DISSERTATION

submitted in partial fulfillment of the

requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2006

Approved by:

Major Professor
John Hatcliff

COPYRIGHT

Venkatesh Prasad Ranganath
(venkateshprasad.ranganath@gmail.com)

2006

ABSTRACT

With the advent of multi-core processors and rich language support for concurrency, the paradigm of concurrent programming has arrived; however, the cost of developing and maintaining concurrent programs is still high. Simultaneously, the increase in social ubiquity of computing is reducing the "time-to-market" factor while demanding stronger correctness requirements. These effects are amplified with ever-growing size of software systems. Consequently, there is (will be) a rise in the demand for scalable and accurate techniques to enable faster development and maintenance of correct large scale concurrent software.

This dissertation presents a collection of scalable and accurate approaches to tackle the above situation. Primarily, *the approaches are focused on discovering dependences (relations) between various parts of the software/program and leveraging the dependences to improve maintenance and development tasks via program slicing (comprehension) and verification.*

Briefly, the proposed approaches are embodied in the following specific contributions:

1. New trace-based foundation for control dependences.
2. An equivalence class based analysis to efficiently and accurately calculate escape information and intra- and inter-thread dependences.
3. A new parametric data flow style slicing algorithm with various extensions to uniformly and easily realize and reason about most existing forms of static sequential and concurrent slicing.
4. A new generic notion of property/trace sensitivity to represent and reason about richer forms of context sensitivity.
5. Program dependence based partial order reduction techniques to enable efficient and accurate state space exploration in both static and dynamic mode.

In an attempt to simplify the approaches, they have been based on the basic concepts/ideas of the affected techniques (e.g. program slicing is a rooted transitive closure of dependence relation). As trace-based reasoning is well suited for concurrent systems, an attempt has been made to explore trace-based reasoning wherever possible.

While providing a rigorous theoretical presentation of these techniques, this effort also validates the techniques by implementing them in a robust tool framework called Indus (available from http://indus.projects.cis.ksu.edu) and then providing experimental results that demonstrate the effectiveness of the techniques on various concurrent applications.

Given the current trend towards concurrent programming and social ubiquity of computing, the approaches proposed in this dissertation provide a foundation for collectively attacking scalability, accuracy, and soundness challenges in current and emerging systems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am thankful to Dr. John M. Hatcliff for providing excellent academic support and advice during the doctoral program. I really appreciate and cherish his confidence in allowing me to conduct independent research. Further, his constant coaxing and support to participate in various research activities and gatherings led to a cherishable learning experience.

Although Dr. Daniel Andresen was not directly involved in my research work, he was accommodative to indulge me in a tangential line of research along with his students. I am thankful for this enriching opportunity.

I am thankful to Dr. Torben Amtoft, Dr. Anindya Banerjee, Dr. Matthew B. Dwyer, and Dr. Robby for providing their valuable time to discuss various aspects of my research and for providing me the opportunity to be part of the research community through various professional activities.

I am grateful to the support of the departmental staff in dealing with adminstrativia and the department in providing excellent computing resources.

Georg Jung (aka Zee-German), Matt Hossier, and others in the Santos group made the grey (the walls of our offices are literally painted grey!) and dense workdays/weeks bearable with engaging lunches and useful insights pertaining to work and other obtuse yet interesting topics (such as beer production and tamagotchis). So, thanks guys!

Without the users of Indus, the project would have remained a rather boring academic exercise. I really appreciated their inputs and interest in making Indus an interesting, fun, and useful project.

Although I partially agree with the saying *"Behind every successful man, there is a woman"*, I am grateful to my wife for her company and support during my graduate studies. Without her, my graduate school years in a foreign country would have certainly seemed longer.

Saving the best for last, I would like to thank my mother, father, and sister for advising me in the ways of life and being patient and supportive during my rather long pursuit of higher studies.

"***Home is the first school***" is a popular adage in Kannada, my native language. As I am a staunch believer in foundations, I am forever indebted to my family for providing an excellent first school.

Chapter 1

Introduction

1.1 Motivation

As the social ubiquity of software-based systems increases, the demand for new software and enhancements to existing software increases. In case of the former demand, new techniques to design, develop, and deliver correct new software are required; in case of the latter demand, new techniques to understand, reason, design, and implement correct enhancements to existing software are required. Independent of the demand, social ubiquity requires the software to be safe. Consequently, the correctness requirements of a software increases.

This situation can be addressed by employing program specification and automated program analysis in software development. The success of this approach hinges on the following features of program analyses.

Scalability Automated approaches are more relevant in large scale software development and maintenance as the complexity of software is usually proportional to its size. Hence, to aid software development, automated approaches should efficiently scale to handle large scale software.

Accuracy If the information provided by automated approaches is inaccurate, then the developers will need to spend time in identifying inaccuracies (false positives). This task can be time consuming and result in high usability overhead. To counter this effect, automated approaches should be highly accurate.

Soundness The task of checking if a software correctly satisfies a specification involves considering every possibility affecting the specification. Hence, automated approaches should be sound and consider all necessary possibilities (not miss any necessary possibility).[1]

The above approach is slowly making its way into everyday development cycle in the various forms such as simplification of code refactoring and trivial program comprehension tasks,[2] detection of common and/or subtle programming errors,[3] reporting of useful metrics[4], and generation of

[1] More possibilities may be considered at the cost of inaccuracy.
[2] Most widely available integrated development environments (IDEs) such as Eclipse, NetBeans, and IntelliJ support such features.
[3] Various standalone and IDE-based tools such as Checkstyle, PMD, FindBugs, and EscJava2 provide this feature.
[4] Metrics calculation features are supported by tools such as JDepend, Metrics, and CAP.

glue/framework code and data.[5]

The main reason for the availability and success of these solutions is the nominal time, resource, and developer effort required to use these solutions and the existence of various approaches to prove the soundness and accuracy of these solutions.

These solutions exhibit a common trait — *the cost of the solution is directly dependent on the size of the aspect of the software/program processed by the solutions and the accuracy of the information catered by the solution.* For example, the cost of calculating abstractness index (as done in JDepend) for a program is proportional to the size of the class hierarchy (aspect) of the program. Similarly, the cost of "inline a local variable" refactoring is proportional to the size of the data and control flow web involving the local variable.

In most cases the size of static aspects of programs such as class hierarchy, nesting levels of conditionals, and occurrences of identifiers are linearly related to the size/measure of the (static) representation of the program such as the number of classes, statements, and expressions. Hence, solutions that process static aspects of programs can be readily used with large scale software with good scalability and high level of accuracy and soundness.

This is not true in case of solutions based on dynamic aspects of programs. Typically, the size of dynamic aspects of programs such as aliasing, side effects, and virtual method dispatch are linearly related to the size/measure of the (dynamic) behavior of programs such as the number of inter-procedural control flow paths, reachable objects, and number of distinct assignments to a variable. As the size of the behavior of a program is significantly larger than the size of its representation, solutions based on dynamic aspects of programs can be intractable. Although abstraction techniques can be used to trim down the size of the behavior and address intractability, the information provided by such abstraction based solutions will often be inaccurate despite being sound.

These observations apply equally in the context of sequential and concurrent programs. Further, most often the size of the dynamic behavior of concurrent programs (such as thread interleaving) is directly dependent on the degree of concurrency of the program. Hence, the cost of solutions that depend on dynamic aspects based on such dynamic behavior increases rapidly; the increase is usually either exponentially or combinatorially in terms of the size of the representation of the program. As a result, most solutions fail to be useful (comparably accurate) and scalable.

Two good examples of solutions that fit the above description are

Program Slicing is a solution to statically identify program parts that may dynamically depend (influence) a set of given program points. Over two decades of research illustrates that program slicing can aid in program comprehension, debugging, maintenance, testing, and specialization.

Model checking is a solution to statically verify properties of an input program by verifying the same properties on a representative abstract model of the program. Similar to program slicing, numerous research efforts suggest that model checking can aid in the hard task of verifying properties of concurrent and/or distributed software.

Clearly, both of the above solutions operate on both static and dynamic aspects of the input program; hence, they share the same drawbacks as most other dynamic aspect based solutions.

Given the obvious usefulness of these solutions and the increase in software ubiquity, there is a pressing need to develop techniques/approaches that enable existing realizations of these solutions to be *scalable, accurate,* and *sound.*

[5] Frameworks used in Architecture-based (CORBA, SOAP) application development support such features.

1.2 Contributions

This dissertation will focus on a collection of approaches that address correctness/soundness, accuracy, and scalability issues in the realm of understanding and reasoning about software. Specifically, the dissertation will describe the following contributions in the context of program slicing and model checking of concurrent Java programs.

- *"Execution in an infinite event processing loop until terminated"* is a common trait of reactive systems. As the programs of such systems have not exit points, the existing notions of control dependences cannot be directly applied to such programs. Further, such applications can lead to incorrect results.

  This situation has been addressed via new definitions of control dependence that are applicable to (such) programs with zero or more exit points without modifying the program structure and loss of accuracy. These notions were proven to be correct in the realm of behavior preservation in program slices. Various coincidence properties between the new notions and the existing notions of control dependences were also proven. As for feasibility, polynomial time algorithms to calculate these new control dependences were proposed and realized.

- Efficient calculation of precise slices of concurrent programs is challenging as it is difficult to statically reason about the dependences that arise when multiple threads perform interfering reads/writes on shared data. The use of heap-allocated data in such programs makes the task even more difficult due to complexity stemming from aliasing. Existing presentations of slicing of concurrent object-oriented programs have made worse case aliasing assumptions when generating edges in program dependence graphs for situations where objects are accessed by different threads.

  As a solution, a scalable equivalence class based escape analysis that is focused on improving the accuracy of interference and ready dependences in concurrent Java programs was proposed. At a nominal cost, this analysis was extended to accurately calculate side effect information, write-write and locking-based coupling, and aliasing based data dependence. Further, the analysis was generalized to be applicable in the calculation of other inter-thread data relations.

- Program slicing was initially proposed as a data flow style analysis involving a set of dependence relation with its correctness reasoning based on projections of program behavior. Although various efforts [RAB+05, HDD+03] have leveraged projection based reasoning to prove correctness of slicing in different settings, there has been no effort to realize slicing as a data flow style analysis or justify the drawbacks of data flow style program slicing algorithms.

  In an attempt to understand and demystify data flow style program slicing, a simple non-graph based (data flow style) parametric program slicing algorithm that is applicable to both sequential and concurrent programs was proposed. This algorithm was successfully parametrized to calculate backward, backward control, and forward slices in both calling context sensitive and insensitive modes.

  Along with other extensions, a novel and general notion of property sensitivity to efficiently improve the accuracy of calling context sensitive slicing of concurrent programs and to reason about various forms of context sensitivity was also proposed.

- Despite the existence of advanced program analysis and possible runtime cost benefits, existing partial order reduction techniques for model checking do not leverage information from analysis of the model under scrutiny.

  As an exploratory first step, a simple and efficient approach to improve the scalability of model checking of concurrent Java programs by leveraging statically calculated dependence

information in both offline and on-line (dynamic) mode of operation was proposed. As a by-product, the first stateful dynamic partial order reduction (based on static dependence information) algorithm was also proposed.

In the past decade, program slicing has been extended to concurrent programs. Despite these efforts, program slicing features were not provided by any publicly available commercial or free tools. As part of this dissertation, I have created the first publicly available batteries-included[6] Java program slicing framework as part of the *Indus* project (`http://indus.projects.cis.ksu.edu`). This project provides software artifacts that embody the approaches/techniques described in this dissertation.

1.3 Organization

The dissertation is organized as follows. The details about the new trace-based definitions of control dependences along with the correctness proof in the context of program slicing are described in Chapter 2. This is followed by the exposition about equivalence class based escape analysis and its application to improve interference and ready dependence in Chapter 3. This chapter also contains description about various extensions and applications of the analysis along with the experimental results. In Chapter 4, a constrained version of Java language that is later used in the description of the slicing algorithms is introduced. The parametric slicing algorithm along with various instantiations is presented in Chapter 5. Various optimizations to the slicing algorithm(s), the details of property sensitivity, and an empirical evaluation of the algorithms is also presented in this chapter. Chapter 6 contains the description of stateful algorithms that leverage static program dependence based static and dynamic partial order reduction techniques along with an analysis of preliminary experimental data. The last chapter concludes the dissertation by summarizing the contributions and listing possible future efforts.

[6] Every analysis required to create a program slicer is provided with the framework.

Chapter 2

Control Dependence

Control dependence is a notion that relates program points based their mutual effect on the control flow reachability, and it is heavily used in compiler optimization, testing, and other applications. Although this notion has studied for over two decades, I discovered that existing notions of control dependence could not be directly applied to systems with multiple exit points and applied, either directly or indirectly, to systems with zero exit points. Along with other researchers[1], I explored these shortcomings and proposed new remedial definitions of control dependence, specifically in the context of sequential intra-procedural program slicing. In particular, I identified the shortcomings, proposed and honed the remedial definitions for control dependences, participated in setting up and proving the correctness of the definitions in the context of slicing, and designed proof-of-concept algorithms. The results from this effort were published under the title *A New Foundation For Control-Dependence and Slicing for Modern Program Structures* [RAB+05] at *European Symposium On Programming (ESOP)* held as part of ETAPS 2005 conference.

This chapter describes the contributions from the above mentioned effort and their extensions along with proof-of-concept algorithms and correctness proofs as described in the 2004 tech report [RAB+04] titled *A New Foundation For Control-Dependence and Slicing for Modern Program Structures.*

2.1 Basic Definitions

2.1.1 Control Flow Graphs

When dealing with foundational issues of control dependence, researchers often cast their work in terms of a simple imperative language phrased in terms of control flow graphs. We follow that practice here and base our presentation on a definition of control-flow graph adapted from Ball and Horwitz [BH93].

Definition 1 (Control Flow Graphs)
A control-flow graph $G = (N, E, n_0)$ is a labeled directed graph in which

[1] The other researchers were Torben Amtoft, Anindya Banerjee, Matthew Dwyer, and John Hatcliff. As it was a collaborative effort, I use "we" instead of "I" while referring to the authors.

- N is a set of nodes that represent statements in a program,
- N is partitioned into two subsets N^S, N^P, where N^S are *statement nodes* with each $n_s \in N^S$ having at most one successor, where N^P are *predicate nodes* with each $n_p \in N^P$ having two distinct successors, and $N^E \subseteq N^S$ contains all nodes of N^S that have no successors, i.e. N^E contains all end nodes of G,
- E is a set of labeled edges that represent the control flow between graph nodes, and
- the start node n_0 has no incoming edges and all nodes in N are reachable from n_0. □

As stated earlier, existing presentations of slicing require that each CFG G satisfies the *unique end node property*: there is exactly one element in $N^E = \{n_e\}$ and n_e is reachable from all other nodes of G. The definition above *does not* require this property of CFGs, but we will consider CFGs with the unique end node property when comparing our work to previous work.

To relate a CFG with the program that it represents, we use the function *code* to map a CFG node n to the code for the program statement that corresponds to that node. Specifically, for $n_s \in N^S$, $code(n_s)$ yields the code for an assignment statement, and for $n_p \in N^P$, $code(n_p)$ the code for the test of a conditional statement. The function *def* maps each node to the set of variables defined (i.e. assigned to) at that node (always a singleton or empty set), and *ref* maps each node to the set of variables referenced at that node.

A CFG *path* π from n_i to n_k is a sequence of nodes $n_i, n_{i+1}, \ldots, n_k$ such that for every consecutive pair of nodes (n_j, n_{j+1}) in the path there is an edge from n_j to n_{j+1}. A path between nodes n_i and n_k can also be denoted as $[n_i..n_k]$. When the meaning is clear from the context, we will use π to denote the set of nodes contained in π and we write $n \in \pi$ when n occurs in the sequence π. Path π is *non-trivial* if it contains at least two nodes. A path is *maximal* if it is infinite or if it terminates in an end node.

The following definitions describe relationships between graph nodes and the distinguished start and end nodes [Muc97]. Node n *dominates* node m in G (written $dom(n, m)$) if every path from the start node s to m passes through n (note that this makes the dominates relation reflexive). Node n *post-dominates* node m in G (written $post\text{-}dom(n, m)$) if every path from node m to the end node n_e passes through n. (Note that for graphs that do not have the unique end node property, the domination relation is well-defined but the post-domination relation is not well-defined.) Node n *strictly post-dominates* node m in G if $post\text{-}dom(n, m)$ and $n \neq m$. Node n is the *immediate post-dominator* of node m if $n \neq m$ and n is the first post-dominator on every path from m to the end node n_e. Node n *strongly post-dominates* node m in G if n post-dominates m and there is an integer $k \geq 1$ such that every path from node m of length $\geq k$ passes through n [PC90]. The difference between strong post-domination and the simple definition of post-domination above is that even though node n occurs on every path from m to n_e (and thus n post-dominates m), it may be the case that n does not strongly post-dominate m due to a loop in the CFG between m and n that admits an infinite path beginning at m and not containing n. Hence, strong post-domination is sensitive to the possibility of non-termination along paths from m to n.

A CFG G of the form (N, E, n_0) is *reducible* if E can be partitioned into disjoint sets E_f (the *forward* edge set) and E_b (the *back* edge set) such that (N, E_f) forms a DAG in which each node can be reached from the entry node n_0 and for all edges $e \in E_b$, the target of e dominates the source of e. All "well-structured" programs, including Java programs, give rise to reducible control-flow graphs. A CFG that is not reducible is referred to as an *irreducible* CFG. The Java virtual machine bytecode language allows for the construction of programs whose corresponding control flow graphs

are irreducible. Our definitions and correctness results apply to both reducible and irreducible control flow graphs.

2.1.2 Program Execution

The execution semantics of program CFGs is phrased in terms of transitions on program states (n, σ) where n is a CFG node and σ is a store mapping the corresponding program's variables to values. A series of transitions gives an *execution trace* through p's statement-level control flow graph. It is important to note that when execution is in state (n_i, σ_i), the code at node n_i has not yet been executed. Intuitively, the code at n_i is executed on the transition from (n_i, σ_i) to successor state (n_{i+1}, σ_{i+1}). Execution begins at the state node (n_0, σ_o), and the execution of each node possibly updates the store and transfers control to an appropriate successor node. Execution of a node $n_e \in N^E$ produces a final state (halt, σ) where the control point is indicated by a special label halt – this indicates a normal termination of program execution. The presentation of slicing in Section 2.4 involves arbitrary finite and infinite non-empty sequences of states written $\Pi = s_1\,,\, s_2, \ldots$.

2.1.3 Notions of Dependence and Slicing

A *program slice* consists of the parts of a program p that (potentially) affect the variable values that are referenced at some program points of interest [Tip95]. Traditionally, the "program points of interest" are called the *slicing criterion*. A slicing criterion C for a program p is a non-empty set of nodes $\{n_1, \ldots, n_k\}$ where each n_i is a node in p's CFG.

The definitions below are the classic ones of the two basic notions of dependence that appear in slicing of sequential programs: *data dependence* and *control dependence* [Tip95].

Data dependence captures the notion that a variable reference is dependent upon any variable definition that "reaches" the reference.

Definition 2 (data dependence) Node n is *data-dependent* on m (written $m \xrightarrow{dd} n$ – the arrow pointing in the direction of data flow) if there is a variable v such that

1. there exists a non-trivial path π in p's CFG from m to n such that for every node $m' \in \pi - \{m, n\}$, $v \notin \mathit{def}(m')$, and
2. $v \in \mathit{def}(m) \cap \mathit{ref}(n)$. □

Control dependence information identifies the conditionals that may affect execution of a node in the slice. Intuitively, node n is control-dependent on a predicate node m if m directly determines whether n is executed or "bypassed".

Definition 3 (control dependence) Node n is *control-dependent* on m in program p (written $m \xrightarrow{cd} n$) if

1. there exists a non-trivial path π from m to n in p's CFG such that every node $m' \in \pi - \{m, n\}$ is post-dominated by n, and
2. m is not strictly post-dominated by n. □

For a node n to be control-dependent on predicate m, there must be two paths that connect m with the unique end node n_e such that one contains n and the other does not. There are several slightly different notions of control-dependence appearing in the literature, and we will consider several of these variants and relations between them in the rest of the exposition. Here we simply note that the above definition is standard and widely used (e.g., see [Muc97]).

We write $m \xrightarrow{d} n$ when either $m \xrightarrow{dd} n$ or $m \xrightarrow{cd} n$. The algorithm for constructing a program slice proceeds by finding the set of CFG nodes S_C (called the *slice set*) from which the nodes in C are reachable via $\xrightarrow{d}$.

Definition 4 (slice set) Let C be a slicing criterion for program p. Then the slice set S_C of p with respect to C is defined as follows:

$$S_C = \{m \mid \exists n \,.\, n \in C \text{ and } m \xrightarrow{d}{}^{*} n\}.$$ □

The notion of slicing described above is referred to as "backward static slicing" because the algorithm starts at the criterion nodes and looks backward through the program's control-flow graph to find other program statements that influence the execution at the criterion nodes.

In many cases in the slicing literature, the desired correspondence between the source program and the slice is not formalized because the emphasis is often on applications rather than foundations, and this also leads to subtle differences between presentations.

When a notion of "correct slice" is given, it is often stated using the notion of *projection* [Wei84]. Informally, given an arbitrary trace Π of p and an analogous trace Π_s of p_s, p_s is a correct slice of p if projecting out the nodes in criterion C (and the variables referenced at those nodes) for both Π and Π_s yields identical state sequences. We will consider slicing correctness requirements in greater detail in Section 2.4.1.

2.2 Assessment of Existing Definitions

2.2.1 Variations in Existing Control Dependence Definitions

Although the definition of control dependence that we stated in Section 2.1 is widely used, there are a number of (sometimes subtle) variations appearing in the literature. One dimension of variation is whether the particular definition captures only *direct* control dependence or also admits *indirect* control dependences. For example, using the definition of control dependence in Definition 3, for Figure 2.1 (a), we can conclude that $a \xrightarrow{cd} f$ and $f \xrightarrow{cd} g$, but $a \xrightarrow{cd} g$ does not hold because g does not post-dominate f. The fact that a and g are indirectly related (a does play a role in determining if g is executed or bypassed) is not captured in the definition of control dependence itself but in the transitive closure used in the slice set construction (Definition 4). However, as we will illustrate later, some definitions of control dependence [PC90] incorporate this notion of transitivity directly into the definition of control dependence.

Another dimension of variation is whether the particular definition is sensitive to non-termination or not. Consider Figure 2.1 (a) where node c represents a post-test that controls a loop – which may be infinite (one cannot tell by simply looking at the CFG). According to Definition 3, $a \xrightarrow{cd} d$ holds

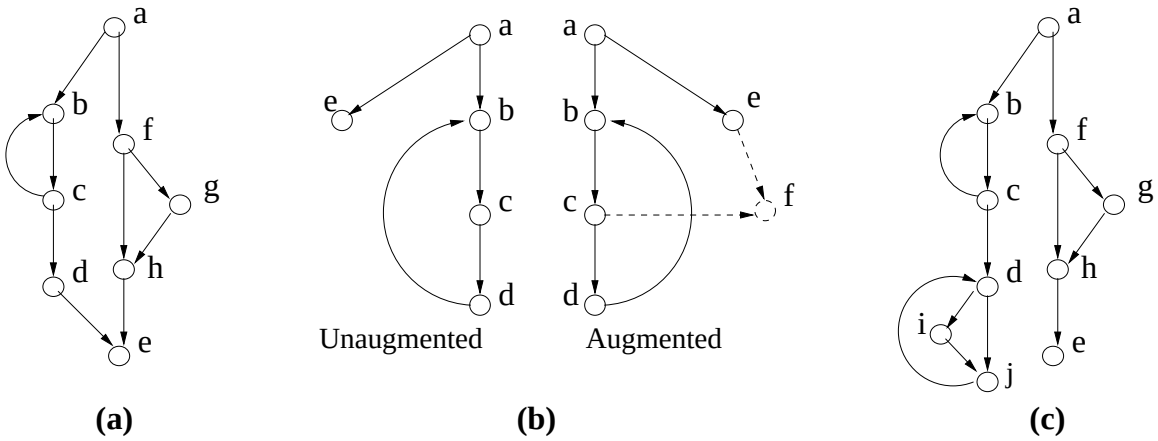

Figure 2.1: (a) is a simple CFG. (b) illustrates how a CFG that does not have a unique exit node reachable from all nodes can be augmented to have unique exit node reachable from all nodes. (c) is a CFG with multiple control sinks of different sorts.

but $c \overset{cd}{\rightarrow} d$ does not hold (because d post-dominates c) even though c may determine whether d executes or never gets to execute due to an infinite loop that postpones d forever. Thus, Definition 3 is *non-termination insensitive*.

We now further illustrate these dimensions by recalling definitions of strong and weak control dependence given by Podgurski and Clarke [PC90] and used in numerous efforts, including the study of control dependence by Bilardi and Pingali [BP96].

Definition 5 (Podgurski-Clarke Strong Control Dependence) n_2 is *strongly control dependent* on n_1 ($n_1 \overset{scd}{\rightarrow} n_2$) if there is a path from n_1 to n_2 that does not contain the immediate post dominator of n_1. □

The notion of strong control dependence is almost identical to control dependence in Definition 3 except that strong control dependence is indirect whereas control dependence in Definition 3 is direct. For example, in Figure 2.1 (a), in contrast to Definition 3, we have $a \overset{scd}{\rightarrow} g$ because there is a path afg which does not contain e, the immediate post-dominator of a. However, given the difference between these variants based on directness, it is not surprising that when used in the context of Definition 4 (which computes the transitive closure of dependences), the two definitions give rise to the same slices.

Definition 6 (Podgurski-Clarke Weak Control Dependence) n_2 is *weakly control dependent* on n_1 ($n_1 \overset{wcd}{\rightarrow} n_2$) if n_2 strongly post dominates n_1', a successor of n_1, but does not strongly post dominate n_1'', another successor of n_1. □

The notion of weak control dependence captures dependences between nodes induced by non-termination, hence, it is non-termination sensitive. Note that for Figure 2.1 (a), $c \overset{wcd}{\rightarrow} d$ because d is a successor of c and strongly post dominates itself, and d does not strongly post-dominate b: the presence of the loop controlled by c guarantees that there does not exist a k such that every path from node b of length $\geq k$ passes through d. Also, in contrast to the notion of strong control

dependence, the notion of weak control dependence is direct. Hence, $n_1 \overset{scd}{\rightarrow} n_2$ does not imply $n_1 \overset{wcd}{\rightarrow} n_2$ but $n_1 \overset{scd}{\rightarrow} n_2$ does imply $n_1 \overset{wcd^*}{\rightarrow} n_2$.

In assessing the above variants of control dependence in the context of program slicing, it is important to note that slicing based on Definition 3 or the strong control dependence above can transform a non-terminating program into a terminating one (i.e., non-termination is not preserved in the slice). In Figure 2.1 (a), assume that the loop controlled by c is an infinite loop. Using the slice criterion $C = \{d\}$, slicing using strong control dependence would generate a slice that includes a but not b and c (we assume no data dependence between d and b or c). Thus, in the sliced program, one would be able to observe an execution of d, but such an observation is not possible in the original program because execution diverges before d is reached. In contrast, the difference between direct and indirect statements of control dependence seems to amount to a largely technical stylistic decision in how the definitions are stated. Table 2.1 shows the control dependences that arise in the CFG of Figure 2.1 (a) for various notions of control dependence that we are considering in this work.

Very few efforts consider the non-termination sensitive notion of weak control dependence above. We conjecture that there are at least two reasons for this. First, although it bears the qualifier "weak", weak control dependence is actually a larger relation[2] and will thus include more nodes in the slice[3]. Second, many applications of slicing focus on debugging and program visualization and understanding, and in these applications having slices that preserve non-termination is less important than having smaller slices. However, slicing is increasingly used in security applications and as a model-reduction technique for software model checking. In these applications, it is quite important to consider variants of control dependence that preserve non-termination properties, since failure to do so could allow inferences to be made that compromise security policies, for instance invalidate checks of liveness properties [HDZ00]. This motivates our careful consideration of non-terminating program behaviors in the definitions of control dependence and slicing that we provide later in the chapter.

Nodes	$\overset{cd}{\rightarrow}$	$\overset{scd}{\rightarrow}$	$\overset{wcd}{\rightarrow}$	$\overset{ntscd}{\rightarrow}$	$\overset{nticd}{\rightarrow}$
a	b, c, d, f, h	b, c, d, f, g, h	b, c, f, h, e	b, c, f, h, e	b, c, d, f, h
c	b	b	b, d, e	b, d, e	b
f	g	g	g	g	g

Table 2.1: Various control dependences existing in the graph in Figure 2.1 (a).Control dependences denoted by $\overset{ntscd}{\rightarrow}$ and $\overset{nticd}{\rightarrow}$ will be introduced in the following pages.

2.2.2 Unique End node restriction on CFG

All definitions of control dependences that we are aware of require that CFGs satisfy the unique end node requirement – but many software systems fail to satisfy this property. Existing works simply require that CFGs have this property, or they suggest that CFGs can be augmented to achieve this property, e.g., using the following steps: (1) insert a new node e into the CFG, (2) add an edge from each exit node (other than e) to e, (3) pick an arbitrary node n in each non-terminating loop and add an edge from n to e. In our experience, such augmentations complicate the system being analyzed in several ways. Non-destructive augmentation performed by cloning the CFG and augmenting the

[2]In Figure 2.1 (a), the size of $\overset{wcd}{\rightarrow}$ relation is 9 whereas the size of $\overset{scd}{\rightarrow}$ relation is 8.

[3]In Figure 2.1 (a), the transitive closure of strong and weak control dependence starting from d are $\{a\}$ and $\{a, c\}$, respectively.

clone would cost time and space. Destructive augmentation performed by directly augmenting the CFG may clash with the requirements of other clients of the CFG, thus necessitating the reversal of the augmentation before subsequent clients use the CFG. If augmentation is not reversed, the graph algorithms and analyses algorithms should be made intelligent to operate on the actual CFG embedded in the augmented CFG.

Many systems have threads where the main control loop has no exit – the loop is "exited" by simply killing the thread. For example, in the Xt library, most applications create widgets, register callbacks, and call `XtAppMainLoop()` to enter an infinite loop that manages the dispatching of events to the widgets in the application. In PalmOS, applications are designed such that they start upon receiving a start code, execute a loop, and terminate upon receiving a stop code. However, the application may choose to ignore the stop code during execution. Hence, the application may not terminate except when explicitly killed. In such cases, a node in the loop must be picked as the loop exit node for the purpose of augmenting the CFG of the application. But this can disrupt the control dependence calculations. In Figure 2.1 (b), we would intuitively expect e,b,c, and d to be control dependent on a in the unaugmented CFG. However, $a \overset{wcd}{\rightarrow} \{e, b, c\}$ and $c \overset{wcd}{\rightarrow} \{b, c, d, f\}$ n the augmented CFG. It is trivial to prune dependences involving f. However, now there are new dependences $c \overset{wcd}{\rightarrow} \{b, c, d\}$ which did not exist in the unaugmented CFG. Although a suggestion to delete any dependence on c may work for the given CFG, it fails if there exists a node g that is a successor of c and a predecessor of d. Also, $a \overset{wcd}{\rightarrow} d$ exists in the unaugmented CFG but not in the augmented CFG, and it is not obvious how to recover this dependence.

We address these issues head-on by considering alternate definitions of control-dependence that do not impose the unique end-node restriction.

2.3 New Dependence Definitions

In previous definitions, a control dependence relationship where n_j is dependent on n_i is specified by considering paths from n_i and n_j to a unique CFG end node – essentially n_i and the end node delimit the path segments that are considered. Since we aim for definitions that apply when CFGs do not have an end node or have more than one end node, we aim to instead specify that n_j is control dependent on n_i by focusing on paths between n_i and n_j. Specifically, we focus on path segments that are delimited by n_i *at both ends* – intuitively corresponding to the situation in a reactive program where instead of reaching an end node, a program's behavior begins to repeat itself by returning again to n_i. At a high level, the intuition behind control dependence remains the same as in, e.g., Definition 3 – executing one branch of n_i always leads to n_j, whereas executing another branch of n_i can cause n_j to be bypassed. The additional constraints that are added (e.g., n_j always occurs before any occurrence of n_i) limits the region in which n_j is seen or bypassed to segments leading up to the next occurrence of n_i – ensuring that n_i is indeed *controlling* n_j. The definition below considers maximal paths (which includes infinite paths) and thus is sensitive to non-termination.

Definition 7 ($n_i \overset{ntscd}{\rightarrow} n_j$) In a CFG, n_j is **(directly) non-termination sensitive control dependent** on node n_i iff n_i has at least two successors, n_k and n_l, such that

1. for all maximal paths from n_k, n_j always occurs and either $n_i = n_j$ or n_j strictly ($n_j \neq n_i$) precedes any occurrence of n_i;

2. there exists a maximal path from n_l on which either n_j does not occur, or n_i strictly precedes

any occurrence of n_j. □

Remark 1 When we, as above, write "n_i strictly precedes any occurrence of n_j in π" we mean that *(a)* n_i occurs in π; and either *(b1)* n_j does not occur in π, or *(b2)* the first occurrence of n_i in π is earlier than the first occurrence of n_j in π. □

We supplement a traditional presentation of dependence definitions with definitions given as formulae in computation tree logic (CTL) [EMCGP99]. CTL is a logic for describing the structure of sets of paths in a graph, making it a natural language for expressing control dependences. Informally, CTL includes two path quantifiers, E and A, which indicate if a path from a given node with a given structure exists or if all paths from that node have the given structure. The structure of a path is defined using one of five modal operators (we refer to a node satisfying the CTL formula ϕ as a ϕ-node): $\mathsf{X}\phi$ states that the successor node is a ϕ-node, $\mathsf{F}\phi$ states the existence of a ϕ-node in the path, $\mathsf{G}\phi$ states that a path consists entirely of ϕ-nodes, $\phi\mathsf{U}\psi$ states the existence of a ψ-node and that the sub-path leading up to that node consists of ϕ-nodes; finally, the $\phi\mathsf{W}\psi$ operator is a variation on U that relaxes the requirement that a ψ-node exists (if not, all nodes in the path must be ϕ-nodes). In a CTL formula, path quantifiers and modal operators occur in pairs, e.g., $\mathsf{AF}\phi$ says that on all paths from a node, a ϕ node occurs. A formal definition of CTL can be found in [EMCGP99].

The following CTL formula captures the definition of control dependence above.

$$n_i \overset{ntscd}{\rightarrow} n_j = (G, n_i) \models \mathsf{EX}(\mathsf{A}[\neg n_i \mathsf{U} n_j]) \wedge \mathsf{EX}(\mathsf{E}[\neg n_j \mathsf{W}(\neg n_j \wedge n_i)]).$$

Here, $(G, n_i) \models$ expresses the fact that the CTL formula is checked against the graph G at node n_i. The two conjuncts are essentially a direct transliteration of the natural language above.

We have formulated the definition above to apply to *execution traces* instead of CFG paths. In this setting, one needs to bound relevant segments by n_i, as discussed above. However, when working on CFG paths, the conditions in Definition 7 can actually be simplified to read as follows: (1) *for all maximal paths from* n_k, n_j *always occurs*, and (2) *there exists a maximal path from* n_l *on which* n_j *does not occur*. The corresponding CTL formula would be

$$n_i \overset{ntscd}{\rightarrow} n_j = (G, n_i) \models \mathsf{EX}(\mathsf{AF}(n_j) \wedge \mathsf{EX}(\mathsf{EG}(\neg n_j)).$$

See Section 2.3.2 for the proof that Definition 7 and its simplification are equivalent on CFGs.

To see that Definition 7 is non-termination sensitive, note that $c \overset{ntscd}{\rightarrow} d$ in Figure 2.1 (a) since there exists a maximal path (an infinite loop between b and c) where d never occurs. Moreover, the definition corresponds to our intuition in Section 2.2.2 in that, in Figure 2.1 (b unaugmented) $a \overset{ntscd}{\rightarrow} e$ because there is an infinite loop through b, c, d and $a \overset{ntscd}{\rightarrow} \{b, c, d\}$ because there is maximal path ending in e that does not contain b, c, or d. In Figure 2.1 (c), note that $d \overset{ntscd}{\rightarrow} i$ because there is an infinite path from j (cycle on jdj) on which i does not occur.

We now turn to constructing a non-termination insensitive version of control dependence. The above non-termination sensitive definition considered all paths leading out of a conditional. Now, we need to limit the reasoning to finite paths that reach a terminal region of the graph. To handle this in the context of CFGs that do not have the unique end-node property, we generalize the concept of *end node* to *control sink* – a set of nodes such that each node in the set is reachable from every

other node in the set and there is no path leading out of the set. More precisely:

Definition 8 (Control sink) A *control sink* κ is a set of CFG nodes that form a strongly connected component such that for each $n \in \kappa$ each successor of n is also in κ. □

Observe that each end node forms a control sink and each loop without any exit edges in the graph forms a control sink. For example, $\{e\}$ and $\{b, c, d\}$ are control sinks in Figure 2.1 (b unaugmented), and $\{e\}$ and $\{d, i, j\}$ are control sinks in Figure 2.1 (c).

Definition 9 (Sink-bounded path) The set of *sink-bounded paths from* n_k (denoted $\mathit{SinkPaths}(n_k)$) contains all *maximal* paths π from n_k with the property that there exists a control sink κ such that

- π contains a node n_s from κ (hence, all nodes following n_s in π will also belong to κ);
- if π is infinite, then all nodes in κ will occur in π infinitely often. □

The latter requirement expresses "fairness". Note that if π_1 is a suffix of π_2, then π_1 is sink-bounded iff π_2 is sink-bounded. Also observe that in a CFG with a unique end node n_e, a path is sink-bounded iff it ends in n_e.

Given a control flow graph, the minor formed by contracting the strongly connected components of the control flow graph will be a DAG with the control sinks being contracted into leaf nodes. This shows:

Lemma 1 *All finite paths can be extended into sink-bounded paths.* □

Existing definitions [BH93, PC90, BP96] of non-termination insensitive control dependence rely on reasoning about paths from the conditional to the end node. We generalize this to reason about paths from a conditional to control sinks.

Definition 10 ($n_i \overset{nticd}{\rightarrow} n_j$) In a CFG, n_j is **(directly) non-termination insensitive control dependent** on n_i iff n_i has at least two successors, n_k and n_l, such that

1. for all paths $\pi \in \mathit{SinkPaths}(n_k)$, $n_j \in \pi$;
2. there exists a path $\pi \in \mathit{SinkPaths}(n_l)$ such that $n_j \notin \pi$. □

This definition is expressed in CTL as

$$n_i \overset{nticd}{\rightarrow} n_j \;=\; (G, n_i) \models \mathsf{EX}(\hat{\mathsf{A}}\mathsf{F}(n_j)) \wedge \mathsf{EX}(\hat{\mathsf{E}}\mathsf{G}(\neg n_j))$$

where $\hat{\mathsf{A}}$ and $\hat{\mathsf{E}}$ represent quantification over sink-bounded paths only; note the similarity to the simplified formula for $\overset{ntscd}{\rightarrow}$ mentioned earlier.

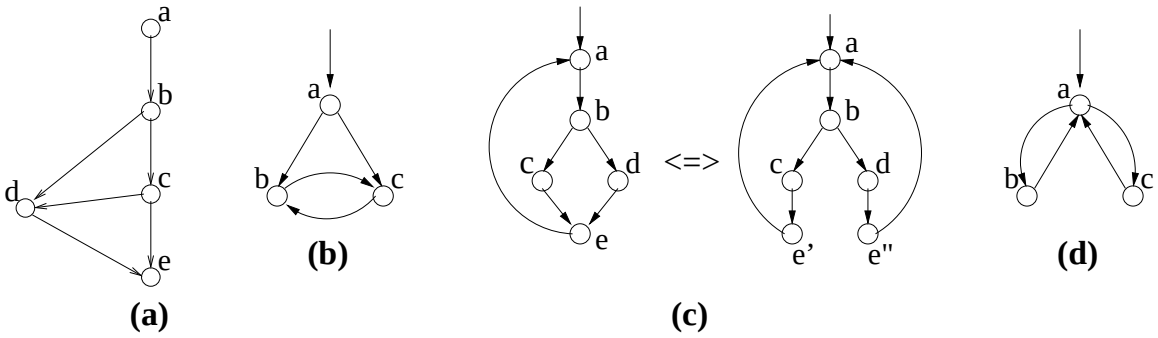

Figure 2.2: More control flow graphs.

To see that this definition is non-termination insensitive, note that $c \overset{nticd}{\not\rightarrow} d$ in Figure 2.1 (a) since there does not exist a path from b to a control sink ($\{e\}$ is the only control sink) that does not contain d. Again, in Figure 2.1 (b unaugmented) $a \overset{nticd}{\rightarrow} e$ because there is a path from b to the control sink $\{b, c, d\}$ and neither the path nor the sink contain e, and $a \overset{nticd}{\rightarrow} \{b, c, d\}$ because there is a path ending in control sink $\{e\}$ that does not contain b, c, or d. It is interesting to note that in Figure 2.1 (c), our definition concludes that $d \overset{nticd}{\not\rightarrow} i$ because although $\{d, i, j\}$ is a control sink and there is a maximal path from d that avoids i (by choosing j over i each time), this path is not sink-bounded thanks to the "fairness" requirement. The consequence of this property is that even though there may be control structures inside of a control sink, they will not give rise to any control dependences. In applications where one desires to detect such dependences, one may apply the definition to control sinks in isolation with back edges removed or use order dependence (described below in Definition 14).

In languages like Java, exception-based control flow paths give rise to control flow graphs with shapes similar to that in Figure 2.2 (a). In this CFG, $b \overset{cd}{\rightarrow} c, b \overset{cd}{\rightarrow} d$, and $c \overset{cd}{\rightarrow} d$. In case of $b \overset{cd}{\rightarrow} d$, it is possible for the control to reach d even if the control flows along $b \rightarrow c$. Hence, b does not *decisively* decide if control can bypass d. However, in case of $c \overset{cd}{\rightarrow} d$, c does *decisively* decide if control can bypass d. The decisiveness stems from the fact that the choice at the control point (c) that prevents the control from reaching the given program point (d) is *final*. Hence, the decisive control dependence relation can be defined as follows.

Definition 11 ($n_i \overset{dcd}{\rightarrow} n_j$) In a CFG, n_j is **(directly) decisively control dependent** on node n_i iff n_i has at least two successors, n_k and n_l, such that

1. for all maximal paths from n_k, n_j always occurs and either $n_j = n_i$ or n_j strictly precedes n_i;
2. for all maximal paths from n_l, n_j does not occur, or n_j is strictly preceded by n_i. □

Although the above definition and Definition 7 are almost identical, they differ in the quantification in the second clause. Hence, the above definition implies Definition 7.

Decisive control dependence is useful to answer the question - *"Which is the control point beyond which the control cannot reach the given program point?"* This information is useful when trying to understand procedures with multiple exit points that are embedded in nested control structure and when trying to find the program point to begin backtracking to discover possible control flow

divergence points.

Programs written in unstructured languages such as JVM bytecodes can give rise to irreducible CFGs for which previous definitions prove to be insufficient to capture dependences. For example, in Figure 2.2 (b), b and c cannot be related to a by any of the above dependences as, given the shape of the CFG, the control will reach b and c once it enters the control sink $\{b, c\}$. However, a does influence if b or c will be executed first when the control does enter the control sink $\{b, c\}$. In other words, the order in which b and c are executed within the control sink is determined by a. To capture ordering relationships between nodes such as a, b, and c in irreducible regions of a CFG, we propose a new notion of dependence called *order dependence*.

Definition 12 ($n_i \overset{dod}{\rightarrow} n_j \leftrightharpoons n_k$) Let n_1, n_2, n_3 be distinct nodes. n_2 and n_3 are decisively order-dependent on n_1, written $n_1 \overset{dod}{\rightarrow} n_2 \leftrightharpoons n_3$, if

1. all maximal paths from n_1 contain both n_2 and n_3,
2. n_1 has a successor from which all maximal paths[4] contain n_2 before any occurrence of n_3, and
3. n_1 has a successor from which all maximal paths contain n_3 before any occurrence of n_2. □

We shall use decisive order dependence in our exposition about slicing and associated correctness proofs.

Observe that the above definition is decisive as it requires that n_1 be the final control point to decide the execution order between n_2 and n_3. By relaxing this requirement, we can arrive at a relatively weaker relation. We refer to this relation as *strong order dependence*. As given in the following definition, the universal quantification on the maximal paths is required for one of n_2 and n_3, successor nodes of n_1.

Definition 13 ($n_i \overset{sod}{\rightarrow} n_j \leftrightharpoons n_k$) Let n_1, n_2, n_3 be distinct nodes. n_2 and n_3 are strongly order-dependent on n_1, written $n_1 \overset{sod}{\rightarrow} n_2 \leftrightharpoons n_3$, if

1. all maximal paths from n_1 contain both n_2 and n_3,
2. there exists a maximal path from n_1 where n_2 occurs before any occurrence of n_3
3. there exists a maximal path from n_1 where n_3 occurs before any occurrence of n_2,
4. n_1 has a successor n_4, such that either
   (a) all maximal paths from n_4 contain n_2 before any occurrence of n_3, or
   (b) all maximal paths from n_4 contain n_3 before any occurrence of n_2. □

Strong order dependence definition can be further generalized to capture control dependence, hence, be applicable to reducible regions of the CFG. The generalization is achieved by removing clause (1) from Definition 13 as done in the following definition.

[4] which will contain both n_2 and n_3, thanks to clause (1).

Definition 14 ($n_i \overset{wod}{\rightarrow} n_j \leftrightharpoons n_k$) In a CFG, nodes n_j and n_k ($n_j \neq n_k$) are weakly order dependent on n_i iff

- there exists a maximal path from n_i where n_j strictly precedes any occurrence of n_k,
- there exists a maximal path from n_i where n_k strictly precedes any occurrence of n_j, and
- n_i has a successor n_l such that either
  - on all maximal paths from n_l, n_j strictly precedes any occurrence of n_k, or
  - on all maximal paths from n_l, n_k strictly precedes any occurrence of n_j. □

Given the definition of various forms of order dependences and the property[5] of reducible CFG – every cycle in a reducible CFG has one node that dominates other nodes of the cycle – it is possible to naively conclude that there can be no order dependences of any form between n_i, n_j, and n_k provided they are distinct and occur in a reducible CFG. This is true in case of decisive (as proved in Lemma 3) and strong variants of order dependence. However, this is not true in case of weak order dependence. As an example, observe that b and c are weakly order dependent on a ($a \overset{wod}{\rightarrow} b \leftrightharpoons c$) in the reducible graph in Figure 2.2 (d) while b and c are neither strongly nor decisively order dependent on a. Further, one can observe and prove (although not done in this effort) that, in a reducible CFG, $a \overset{wod}{\rightarrow} b \leftrightharpoons c \implies a \overset{ntscd}{\rightarrow} b \vee a \overset{ntscd}{\rightarrow} c$.

Although order dependence captures the ordering on nodes imposed by control flow, it is overly conservative in cases where such an ordering is required only to preserve the data values observed during execution. In other words, if there is no variable that is used(defined) in b and defined(used) in c, then the data values observed during execution of b and c are independent of the order in which b and c are executed. In such cases, the execution order imposed by a on b and c is uninteresting if the order is observed only by the changes to variables used in a and b and not by the order of program points encountered during execution. This data-sensitive order relation is captured by *data-sensitive order dependence*, a stronger form of *order dependence*.

Definition 15 ($n_i \overset{dsod}{\rightarrow} n_j \leftrightharpoons n_k$) In a CFG, nodes n_j and n_k ($n_j \neq n_k$) are **data-sensitive order dependent** on n_i iff

1. $n_i \overset{sod}{\rightarrow} n_j \leftrightharpoons n_k$;
2. either $n_j \overset{dd}{\rightarrow} n_k$ or $n_k \overset{dd}{\rightarrow} n_j$. □

2.3.1 Examples

Consider Figure 2.1 (c). According to Definition 7, $a \overset{ntscd}{\rightarrow} b$ as the first execution of b depends on the choice made at a. Likewise, $a \overset{ntscd}{\rightarrow} c$ and $a \overset{ntscd}{\rightarrow} f$. Similarly, $f \overset{ntscd}{\rightarrow} g$. Independent of the choice made at f, the control will always reach h. Hence, $f \not\overset{ntscd}{\rightarrow} h$ but $a \overset{ntscd}{\rightarrow} h$. Similarly, $a \overset{ntscd}{\rightarrow} e$, and $c \overset{ntscd}{\rightarrow} b$. If $b \rightarrow c \rightarrow b$ is an infinite loop, control will never reach d; the length of the loop is

[5] Definition (f) in the abstract of [HU74]

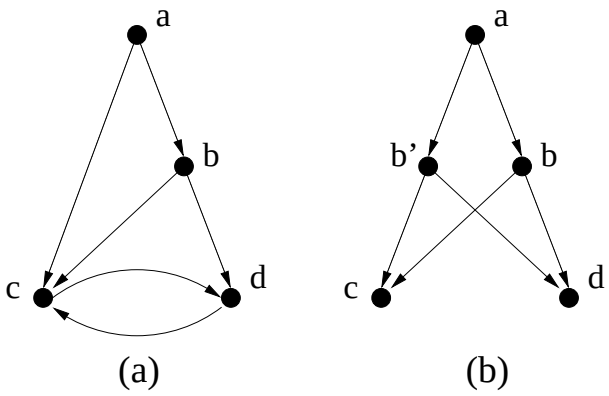

Figure 2.3: Control flow graphs specific to order dependence.

dependent on the choice made at c. Hence, $c \overset{ntscd}{\rightarrow} d$. In the loop starting at d, it is possible that the control will bypass i in an iteration while it reaches i in a subsequent iteration depending on the choice made at d.[6] Hence, $d \overset{ntscd}{\rightarrow} i$.

In a non-termination insensitive setting, loops are assumed to be terminating (provided the loop has an exit node). Hence, in Figure 2.1 (c), the loop $b \rightarrow c \rightarrow b$ is assumed terminating as it has an exit edge $c \rightarrow d$. This implies that the loop cannot indefinitely delay the control from reaching d. Hence, $c \overset{nticd}{\not\rightarrow} d$. As for other non-termination sensitive control dependences for the same graph, most of them hold also in the non-termination *in*sensitive case, except we have $d \overset{nticd}{\not\rightarrow} i$, and also $c \overset{nticd}{\not\rightarrow} j$ as j belongs to the control sink that terminates all sink-bounded paths from c. As for decisive control dependence, in Figure 2.2 (a), $c \overset{dcd}{\rightarrow} d$ and $b \overset{dcd}{\not\rightarrow} d$.

Nodes	$\overset{ntscd}{\rightarrow}$	$\overset{nticd}{\rightarrow}$
a	b, c, f, h, e	b, c, d, i, j, f, h, e
c	b, d, j	b
d	i	–
f	g	g

Table 2.2: Various control dependences (based on new definitions) existing in the graph in Figure 2.1 (c).

In Figure 2.2 (b), b and c are decisively order dependent on a ($a \overset{dod}{\rightarrow} b \leftrightharpoons c$), hence, $b \overset{dd}{\rightarrow} c$ or $c \overset{dd}{\rightarrow} b$ implies $a \overset{dsod}{\rightarrow} b \leftrightharpoons c$. This result also holds for *strong* and *weak* order dependence. On the other hand, b and c are weakly order dependent on a in Figure 2.2 (d) but they are neither related by decisive nor strong order dependence. Also, i and j are weakly order dependent on d ($d \overset{sod}{\rightarrow} i \leftrightharpoons j$) in Figure 2.1 (c).

Focusing only on order dependences, in Figure 2.3 (a), c and d are strongly and weakly order dependent on both b and a. However, c and d are decisively order dependence only on b.

Unlike in Figure 2.3 (a), c and d are neither strongly nor decisively order dependent on b, b', and a in Figure 2.3 (b). The reason for this is the absence of edges $c \rightarrow d$ and $d \rightarrow c$. However, c and d are weakly order dependent on b and b' in Figure 2.3 (b).

[6] Observe that the loop starting at d can be split into two loops as done in Figure 2.2 (c).

2.3.2 Properties of the Dependence Relations

Conservative extension. We begin by showing that the new definitions of control dependence conservatively extend classic definitions: when we consider our definitions in the original setting with CFGs with unique end nodes, the definitions coincide with the classic definitions (as already suggested by the examples in Table 2.1).

Theorem 1 (Coincidence Properties, I) *For all CFGs with the unique end node property, and for all nodes* $n_i, n_j \in N$, $n_i \xrightarrow{cd} n_j$ *if and only if* $n_i \xrightarrow{nticd} n_j$. □

PROOF First notice that for any n and m, m post-dominates n if and only if every sink-bounded path from n contains m.

We shall first prove "only if": so let $n_i \xrightarrow{cd} n_j$. There thus exists a non-trivial path π from n_i to n_j such that every node in π except n_i is post-dominated by n_j. Let π take the form $n_i, n_k, \ldots, n_j$; we can assume that if $n_j \neq n_i$ then $n_k \neq n_i$. Here n_k may equal n_j, but in any case it will hold that n_k is post-dominated by n_j.

Also, we know from $n_i \xrightarrow{cd} n_j$ that n_i is not strictly post-dominated by n_j. Therefore either $n_i = n_j$, or n_i is not post-dominated by n_j. In either case, since the end node is reachable from all nodes, we infer that n_i has a successor n_l which is not post-dominated by n_j.

We have thus found n_k and n_l, such that all sink-bounded paths from n_k contain n_j, and such that there exists a sink-bounded path from n_l not containing n_j. This shows $n_i \xrightarrow{nticd} n_j$.

Next we prove "if": so let $n_i \xrightarrow{nticd} n_j$. Thus n_i has (at least) two successors, n_k and n_l, such that (i) all sink-bounded paths from n_k contain n_j; and (ii) there exists a sink-bounded path from n_l not containing n_j. From (ii) we infer that either $n_i = n_j$ or n_i is not post-dominated by n_j; in either case, n_i is not strictly post-dominated by n_j.

Since the end node n_e is reachable from all nodes, we know from (i) that there exists a path from n_k to n_j; let π be a shortest such path. In order to show that $n_i \xrightarrow{cd} n_j$, it suffices to show that all nodes in π are post-dominated by n_j. But this clearly follows from (i). ■

Before we prove coincidence property between weak control dependence and the non-termination sensitive control dependence, we prove the equivalence between the original and the simplified definition of non-termination sensitive control dependence. For readability, we restate the simplified definition of non-termination sensitive control dependence.

Definition 16 In a CFG, n_j is non-termination sensitive control dependent on n_i iff

(a) n_i has two successors n_k and n_l;

(b) on all maximal paths from n_k, n_j occurs;

(c) there exists a maximal path from n_l on which n_j does not occur.

Lemma 2 *For a CFG, Definition 16 is equivalent to the original definition of non-termination*

sensitive control dependence (Definition 7). □

PROOF First, we restate the definition of directly non-termination sensitive control dependence (Definition 7); we have $n_i \overset{ntscd}{\rightarrow} n_j$ iff:

ntscd(i) n_i has at least two successors, n_k and n_l.

ntscd(ii) For all maximal paths from n_k, n_j always occurs and either it equals n_i or it occurs before any occurrence of n_i.

ntscd(iii) There exists a maximal path from n_l on which either n_j does not occur, or n_j is strictly preceded by n_i.

Since **ntscd(ii)** implies **(b)**, and **(c)** implies **ntscd(iii)**, we are left with showing two implications:

- First, we show that **(b)** implies **ntscd(ii)**: Let π be a maximal path from n_k. By **(b)**, n_j occurs there. Now assume, towards a contradiction, that in π, n_i occurs strictly before any occurrence of n_j. Since there is an edge from n_i to n_k, this means that the graph has a cycle containing n_k but not containing n_j. But then we can find a maximal path from n_k where n_j does not occur, contradicting **(b)**.
- Next, we show **ntscd(iii)** implies **(c)**: Let π be a maximal path from n_l on which n_i occurs strictly before the first (if any) occurrence of n_j. If π does not contain n_j, we are done. So assume that π does contain n_j, but that n_i occurs strictly before. But since there is an edge from n_i to n_l, this means that the graph has a cycle containing n_l but not containing n_j. Then we can find a maximal path from n_l where n_j does not occur, as desired.

This concludes the proof of Lemma 2. Note that we have not assumed the "unique end node" property. ■

As could be expected, we have a similar result relating the corresponding CTL formulae:

Theorem 2 (Simplified NTSCD CTL Equivalence) *The expression of NTSCD as a CTL formula over CFG paths (ϕ_{CFG}):*

$$n_i \overset{ntscd}{\rightarrow} n_j = (G, n_i) \models \mathit{EX}(\mathit{AF}(n_j) \wedge \mathit{EX}(\mathit{EG}(\neg n_j))).$$

is equivalent to the CTL formula over execution traces (ϕ_{trace}):

$$n_i \overset{ntscd}{\rightarrow} n_j = (G, n_i) \models \mathit{EX}(\mathit{A}[\neg n_i \mathit{U} n_j]) \wedge \mathit{EX}(\mathit{E}[\neg n_j \mathit{W}(\neg n_j \wedge n_i)]).$$

PROOF It suffices to prove that the pairs of sub-formulae under the EX operators in the two formulae are equivalent.

We prove that ϕ_{CFG} implies ϕ_{trace} in two steps:

1. $\mathsf{EG}(\neg n_j)$ implies $\mathsf{E}[\neg n_j \mathsf{W}(\neg n_j \wedge n_i)]$:
   The definition of EW requires its left operand to be true until the right operand holds. Thus

if the left operand holds throughout the trace, by the definition of $\mathsf{EG}(\neg n_j)$, then $\neg n_j$ must hold until $\neg n_j \wedge n_i$).

2. $\mathsf{AF}(n_j)$ implies $\mathsf{A}[\neg n_i \mathsf{U} n_j]$):
   The AU operator requires that a n_j state is reached, which holds by the definition of $\mathsf{AF}(n_j)$, and that all prefixes of traces ending in n_j must be free of n_i states.

   In the following, we use regular expressions over CFG node names, n_j, to describe the structure of CFG paths. In this context, negation, $\neg n_j$, is used to denote the absence of a particular control point, n_j.

   Every path from a CFG node n_i either has a prefix that is cyclic in n_i, $n_i(\neg n_i)^* n_i$, or is a path that is acyclic in n_i, $n_i(\neg n_i)^*$. All proper suffixes of paths that are acyclic in n_i are free of n_i by definition. If there exists a path with a prefix that is cyclic in n_i, then there must exist a CFG path of the form $(n_i(\neg n_i)^* n_i)^*$. If $\mathsf{AF}(n_j)$ holds on such a path then it must be the case that n_j appears in the body of the cycle, $(\neg n_i)^*$. Thus, all paths that satisfy $\mathsf{AF}(n_j)$ and begin with a prefix that is cyclic in n_i must begin with a prefix of the form $n_i(\neg n_i)^* n_j$.

   Thus ϕ_{CFG} implies ϕ_{trace}.

We prove that ϕ_{trace} implies ϕ_{CFG} in two steps:

1. $\mathsf{A}[\neg n_i \mathsf{U} n_j])$ implies $\mathsf{AF}(n_j)$:
   The AU operator requires that eventually its right operand n_j becomes true which is the definition of $\mathsf{AF}(n_j)$.

2. $\mathsf{E}[\neg n_j \mathsf{W}(\neg n_j \wedge n_i)]$ implies $\mathsf{EG}(\neg n_j)$:
   If the right operand of the EW never becomes true in a trace then $\neg n_j$ must hold throughout the trace which is equivalent to enforcing $\mathsf{EG}(\neg n_j)$.

   The EW operator, however, only requires $\neg n_j$ to hold up along the trace to the point where n_i holds. For the implication to hold we must show that that $\neg n_j$ will persist through the rest of the path.

   Consider a CFG path from n_i that is free of n_j up to the first occurrence of n_i; this satisfies the above EW. This path has a prefix of the form $n_i(\neg n_j)^* n_i$ and by iterating that prefix we can construct a path $(n_i(\neg n_j)^* n_i)^*$ that satisfies $\mathsf{EG}(\neg n_j)$.

   Thus ϕ_{trace} implies ϕ_{CFG}. ■

Theorem 3 (Coincidence Properties, II) *For all CFGs with the unique end node property, and for all nodes $n_i, n_j \in N$, $n_i \overset{wcd}{\rightarrow} n_j$ if and only if $n_i \overset{ntscd}{\rightarrow} n_j$.* □

PROOF By Lemma 2, we can prove the equivalence by showing that Podgurski-Clarke's weak control dependence from Definition 6 is equivalent to Definition 16.

For readability, we restate Podgurski-Clarke's definition of weak control dependence; we have $n_i \overset{wcd}{\rightarrow} n_j$ iff:

pcwcd(i) n_i has at least two successors, n_k and n_l.

pcwcd(ii) n_j strongly post-dominates n_k.

pcwcd(iii) n_j does not strongly post-dominate n_l.

There are four steps.

1. **pcwcd(ii)** implies **(b)**: Let π be a maximal path from n_k. We must show that n_j occurs in π. There are two possibilities:
   π *is finite*: The last node of π must be an end node. Since n_j post-dominates n_k, this shows that n_j occurs in π.
   π *is infinite*: We know that there exists q such that all paths from n_k longer than q contain n_j; in particular, π will contain n_j since π is infinite, hence longer than q.

2. **(b)** implies **pcwcd(ii)**: First let us show that n_j post-dominates n_k; so let π be a path from n_k to an end node. We must show that π contains n_j, but this follows from **(b)** since π is maximal.

   Next we must find a q such that all paths from n_k longer than q contain n_j; we claim that we can choose q to be one more than the number of nodes in the CFG. For let π be a path from n_k longer than q: it contains a repetition, so if n_j does not occur in π we can construct a maximal path from n_k with n_j not occurring, yielding a contradiction.

3. **pcwcd(iii)** implies **(c)**: Here we have two cases.
   n_j *does not post-dominate* n_l: Then there exists a path π from n_l to an end node such that n_j does not occur in π. The claim now follows since π is maximal.
   For all q*, there exists a path from* n_l *longer than* q *where* n_j *does not occur*: With q the number of nodes in the CFG, we infer that there exists a path from n_l containing repetitions but not containing n_j; this shows that we can construct a maximal (infinite) path from n_l on which n_j does not occur.

4. **(c)** implies **pcwcd(iii)**: Our assumption is that there exists a maximal path π from n_l with n_j not occurring in π. Now there are two cases:
   π *is finite, with the last node being an end node*: But then n_j does not post-dominate n_l, in particular n_j does not strongly post-dominate n_l.
   π *is infinite*: But then for any q, π will be a path from n_l of length q not containing n_j, again showing that n_j does not strongly post-dominate n_l.

This concludes the proof of Theorem 3. ■

Non-termination sensitivity relates more nodes. For an arbitrary CFG, direct non-termination insensitive control dependence (Definition 10) implies the *transitive closure* of direct non-termination sensitive control dependence.

Theorem 4 *For all nodes* $n_i, n_j \in N$, $n_i \overset{nticd}{\rightarrow} n_j$ *implies* $n_i \overset{ntscd^*}{\rightarrow} n_j$. □

Note that this result is supported by the examples in Tables 2.1 and 2.2. For example, in Figure 2.1 (a), $a \overset{nticd}{\rightarrow} d$ holds but $a \overset{ntscd}{\rightarrow} d$ does not. But $a \overset{ntscd^*}{\rightarrow} d$ holds as both $a \overset{ntscd}{\rightarrow} c$ and $c \overset{ntscd}{\rightarrow} d$ hold.

PROOF Our assumption is that n_i has successors n_k, n_l such that **(i)** n_j occurs on all sink-bounded paths from n_k and **(ii)** there exists a sink-bounded path from n_l on which n_j does not occur.

Now consider a sink-bounded path π from n_i via n_k (there exists such a path, by Lemma 1). We can write $\pi = [u_0, u_1, \ldots, u_m, \ldots]$ where $m \geq 1$, $u_0 = n_i$, $u_1 = n_k$, $u_m = n_j$, $u_p \neq n_j$ for $1 \leq p < m$.

Observe that for all $i = 1 \ldots m$, n_j occurs on all sink-bounded paths from u_i (otherwise **(i)** would be contradicted). So, if all sink-bounded paths from n_l would contain u_i, all sink-bounded paths from n_l would contain n_j, contradicting **(ii)**. Thus for all $i = 1 \ldots m$, there exists a sink-bounded path from n_l not containing u_i.

Now define predicates Q_p such that $Q_p(i)$ holds iff $0 \leq i \leq p \leq m$ and all maximal paths from u_i contain u_p. Observe that if $Q_p(i)$ does not hold but $Q_p(i+1)$ holds, then $u_i \overset{ntscd}{\rightarrow} u_p$ (cf. Definition 16). Also observe that $Q_p(p)$ holds for all $p \leq m$, but if $u_p \neq u_0$ then $Q_p(0)$ does not hold (for if all maximal paths from u_0 contain u_p then all maximal paths from n_l contain u_p so also all sink-bounded paths from n_l contains u_p, contradicting the above).

Now we are ready for the construction: if $u_m = u_0$, we are done. Otherwise, we can find j_1 such that $Q_m(j_1)$ does not hold but $Q_m(j_1 + 1)$ holds, showing that $u_{j_1} \overset{ntscd}{\rightarrow} u_m$. If $u_{j_1} = u_0$, we are done. Otherwise, since $Q_{j_1}(j_1)$ holds but $Q_{j_1}(0)$ does not hold, we can find j_2 such that $Q_{j_1}(j_2)$ does not hold but $Q_{j_1}(j_2 + 1)$ holds, showing that $u_{j_2} \overset{ntscd}{\rightarrow} u_{j_1}$. Now we can repeat as desired. ■

Order dependency is relevant for irreducible graphs only.

Lemma 3 *For a reducible CFG, the relations* $\overset{dod}{\rightarrow}$ *and* $\overset{sod}{\rightarrow}$ *are empty.* □

PROOF Assume that $n_1 \overset{sod}{\rightarrow} n_2 \leftrightharpoons n_3$ or $n_1 \overset{dod}{\rightarrow} n_2 \leftrightharpoons n_3$. Thus n_1,n_2,n_3 are distinct, and

od(i) all maximal paths from n_1 contain both n_2 and n_3, and

od(ii) in one maximal path from n_1, n_2 occurs before the first occurrence of n_3, and

od(iii) in one maximal path from n_1, n_3 occurs before the first occurrence of n_2.

We shall show that from these assumptions, a contradiction can be derived when the CFG is reducible. First observe that

n_1 is reachable from neither n_2 nor n_3. (*1)

For otherwise, we could wlog. assume that there is a path from n_2 to n_1 not containing n_3, which by **od(ii)** entails that there exists a maximal path from n_1 not containing n_3, contradicting **od(i)**.

Since the CFG is assumed reducible, its edges E can be partitioned into forward edges E_f and back edges E_b. Here E_f forms an acyclic graph, so wlog. we can assume that n_2 is *not* reachable in E_f from n_3. Since by **od(iii)** and **od(i)**, n_2 is reachable in E from n_3, there exists a node n_4 and

in E_f, a path $[n_3..n_4]$ not containing n_2 (*2)

and a back edge

$n_4 \rightarrow n_5$ where n_5 dominates n_4. (*3)

With n_0 the start node of the CFG, due to (*1) there exists

a path $[n_0..n_1]$ not containing n_2. (*4)

Also, by assumption **od(iii)**, there exists

a path $[n_1..n_3]$ not containing n_2. (*5)

From (*4), (*5), (*2) we see that there is

a path $[n_0..n_4]$ containing n_1 but not containing n_2.

By (*3) we infer that n_5 is on that path, and that there is a path from n_4 to n_4 not containing n_2. Thus we can construct a maximal path from n_1 not containing n_2, contradicting **od(i)**. ■

Observables. For the (bisimulation-based) correctness proof in Section 2.4.1, we shall need a few results about slice sets, members of which are termed "observable". Typically, these results require slice sets Ξ to be closed under non-termination sensitive control dependency, i.e., if $n_1 \stackrel{ntscd}{\rightarrow} n_2$ and $n_2 \in \Xi$ then also $n_1 \in \Xi$. For certain weaker results, it is sufficient to demand that Ξ is closed under non-termination insensitive control dependency, i.e., if $n_1 \stackrel{nticd}{\rightarrow} n_2$ and $n_2 \in \Xi$ then also $n_1 \in \Xi$. (By Theorem 4, the latter closedness property is weaker than the former.) For the main result (Theorem 5), we shall also demand Ξ to be closed under (decisive) order dependency, i.e., if $n_i \stackrel{dod}{\rightarrow} n_j \leftharpoondown n_k$ with $n_j, n_k \in \Xi$ then also $n_i \in \Xi$.

A key feature of our development is the notion of "first observable", where we now present a "may" definition:

Definition 17 For a node n, $obs^1_{may}(n)$ is the set of nodes $n' \in \Xi$ with the property that there exists a path $[n..n']$ where all nodes except n' are not in Ξ. □

Clearly, if $n \in \Xi$ then $obs^1_{may}(n) = \{n\}$. Next, a "must" definition of "subsequent observable":

Definition 18 For a node n, $obs^*_{must}(n)$ is the set of nodes $n' \in \Xi$ with the property that all maximal paths from n contain n'. □

A crucial property of a slice set is that "may" implies "must", i.e., the first observable on any path will be encountered sooner or later on all other paths:

Lemma 4 *Assume the node set Ξ is closed under non-termination sensitive control dependency. Then for all nodes n, $obs^1_{may}(n) \subseteq obs^*_{must}(n)$.* □

PROOF Assume, in order to arrive at a contradiction, that there exists a node n_0 such that $obs^1_{may}(n_0)$ is not a subset of $obs^*_{must}(n_0)$; thus there exists $n_1 \in \Xi$ with $n_1 \in obs^1_{may}(n_0)$ but $n_1 \notin obs^*_{must}(n_0)$. The situation is that there is a path π from n_0 to n_1 where all nodes except n_1 do not belong to Ξ. We infer that $n_0 \notin \Xi$, as otherwise we would have $n_1 = n_0$, contradicting $n_1 \notin obs^*_{must}(n_0)$. We define a predicate Q such that

$Q(n)$ holds iff $n_1 \in obs^*_{must}(n)$.

By our assumption, $Q(n_0)$ does not hold; clearly, $Q(n_1)$ holds. Therefore, π can be written as $[n_0..n_2 n_3..n_1]$ where $Q(n_2)$ does not hold but $Q(n_3)$ holds (that is, there is an edge from n_2 to n_3; note that n_2 may equal n_0 and that n_3 may equal n_1 but we know that $n_1 \neq n_2$).

We shall show that $n_2 \overset{ntscd}{\rightarrow} n_1$; then from $n_1 \in \Xi$ and from Ξ being closed under $\overset{ntscd}{\rightarrow}$ we get $n_2 \in \Xi$ which contradicts n_1 being the only node in π which is also in Ξ.

Since $Q(n_2)$ does not hold, there exists a maximal path starting at n_2 not containing n_1; that path has to have at least two elements (since n_2 has an outgoing edge) and the second element cannot be n_3 (as $Q(n_3)$ holds). Therefore, the second element is some node n_4 with $n_3 \neq n_4$, and there exists a maximal path from n_4 which does not contain n_1. Our final obligation (cf. Definition 16) is to prove that all maximal paths from n_3 contain n_1, which follows since $Q(n_3)$ holds. ■

In a similar way we can show:

Lemma 5 *Assume Ξ is closed under $\overset{nticd}{\rightarrow}$. Assume $n_1 \in obs^1_{may}(n_0)$. Then all sink-bounded paths from n_0 will contain n_1.* □

As a consequence we have the following result, giving conditions to preclude the existence of infinite un-observable paths:

Lemma 6 *Assume that $n_0 \notin \Xi$, but that there is a path π starting at n_0 which contains a node in Ξ.*

- *If Ξ is closed under non-termination insensitive control dependency, then all sink-bounded paths starting at n_0 will reach Ξ.*
- *If Ξ is also closed under non-termination sensitive control dependency, then all maximal paths starting at n_0 will reach Ξ.* □

Our main result, Theorem 5 given below, states that from a given node there is a unique first observable. This does not hold without extra assumptions, however, as demonstrated by the (irreducible) CFG in Figure 2.2(b) where $\Xi = \{b, c\}$ is closed under non-termination sensitive control dependency (since $a \overset{ntscd}{\not\rightarrow} b$ and $a \overset{ntscd}{\not\rightarrow} c$) and provides a with *two* possible first observables. Our remedy is to demand that the slice set Ξ is closed under decisive order dependency, as defined in Definition 12. Recall (Lemma 3) that a reducible graph is vacuously closed under decisive order dependency.

Theorem 5 *If Ξ is closed under $\overset{ntscd}{\rightarrow}$ and $\overset{dod}{\rightarrow}$, then for all nodes n it holds that $obs^1_{may}(n)$ is at most a singleton.* □

PROOF Assume the contrary, and let n_0 be such that $|obs^1_{may}(n_0)| > 1$, implying (by Lemma 4) that $|obs^*_{must}(n_0)| > 1$. Then there cannot exist a maximal path π from n_0 such that $|obs^1_{may}(n)| > 1$ holds for all n occurring in π, for then π would contain no nodes in Ξ, contradicting $obs^*_{must}(n_0)$ being non-empty. Thus there exists a node n_1 such that $|obs^1_{may}(n_1)| > 1$, and thus $n_1 \notin \Xi$, but for all n which are successors of n_1, $obs^1_{may}(n)$ is (at most) a singleton. Since $|obs^1_{may}(n_1)| > 1$, we can

find $n_2, n_3 \in obs^1_{may}(n_1)$ with $n_2 \neq n_3$. Clearly, n_1 has a successor u_2 with $obs^1_{may}(u_2) = \{n_2\}$, and a successor u_3 with $obs^1_{may}(u_3) = \{n_3\}$.

We shall now argue that $n_1 \overset{dod}{\rightarrow} n_2 \leftharpoondown n_3$, which since Ξ is closed under $\overset{dod}{\rightarrow}$ and since $n_2, n_3 \in \Xi$ will imply $n_1 \in \Xi$, yielding the desired contradiction. Looking at Definition 12, we see that for reasons of symmetry, it is sufficient to show the following items:

from n_1, all maximal paths contain n_2: this follows since $n_2 \in obs^1_{may}(n_1) \subseteq obs^*_{must}(n_1)$.

from a successor of n_1, all maximal paths contain n_2 before n_3: u_2 is a successor of n_1 where $obs^1_{may}(u_2) = \{n_2\}$ so there is no way that a path from u_2 can contain n_3 before n_2. ■

Note: Theorem 5 will *not* hold if we assume only that Ξ is closed under non-termination *in*sensitive control dependency. To see this, consider the following reducible graph: in E_f, there is an edge from n_2 to n_1 and an edge from n_1 to n_3 and also a direct edge from n_2 to n_3; in E_b, there is an edge from n_1 to n_2 and an edge from n_3 to n_2. The only control sink is thus the whole graph, so the relation $\overset{nticd}{\rightarrow}$ is empty; also the relation $\overset{dod}{\rightarrow}$ is empty (by Lemma 3). All node sets are thus vacuously closed under $\overset{nticd}{\rightarrow}$ and $\overset{dod}{\rightarrow}$. Assume Ξ is such that $n_2, n_3 \in \Xi$ but $n_1 \notin \Xi$. Then $obs^1_{may}(n_1)$ contains *two* elements: n_2 and n_3. (On the other hand, n_3 is non-termination sensitive dependent on n_1, so Ξ is not closed under non-termination sensitive control dependency, and the scenario is thus *not* a counterexample to the actual Theorem 5.)

2.4 Slicing

We now describe how to slice a CFG G wrt. a slice set S_C, the smallest set containing C which is closed under data dependence $\overset{dd}{\rightarrow}$ and also under $\overset{ntscd}{\rightarrow}$ and under $\overset{dod}{\rightarrow}$.

Definition 19 (Slicing Transformation) The result of slicing is a program with the same CFG as the original one, but with the code map $code_1$ replaced by $code_2$. Here for $n \in S_C$ we have $code_2(n) = code_1(n)$, and for $n \notin S_C$ we have

- if n is a statement node then $code_2(n)$ is the statement `skip`;
- if n is a predicate node then $code_2(n)$ is `cskip`, the semantics of which is that it non-deterministically chooses one of its successors. □

The above definition is conceptually simple, so as to facilitate the correctness proofs. Of course, one would want to do some post-processing, like eliminating `skip` commands and eliminating `cskip` commands where the two successor nodes are equal; we shall not address this issue further but remark that most such transformations are trivially meaning preserving.

2.4.1 Correctness Properties

The main intuition behind our notion of slicing correctness is that the nodes in a slicing criterion C represent "observations" that one is making about a CFG G under consideration. Specifically,

for an $n \in C$, one can observe that n has been executed and also observe the values of any variables referenced at n. Execution of nodes not in C correspond to *silent moves* or non-observable actions. The slicing transformation should preserve the behavior of the program with respect to C-observations, but parts of the program that are irrelevant with respect to computing C observations can be "sliced away". The slice set S_C built according to Definition 4 represents the nodes that are relevant for maintaining the observations C. Thus, to prove the correctness of slicing we will establish the stronger result that G will have the same S_C observations wrt. the original code map $code_1$ as wrt. the sliced code map $code_2$, and this will imply that they have the same C observations.

The discussion above suggests that appropriate notions of correctness for slicing reactive programs can be derived from the notion of weak bisimulation found in concurrency theory, where a transition may include a number of τ-moves [Mil89]. Recall from Section 2.1.2 that a state s is a pair (n, σ) where σ is a store mapping variables into values.

Definition 20 For $i = 1, 2$ we write

- $i \vdash s \rightarrow s'$ to denote that wrt. code map $code_i$, the program state s rewrites in one step to s',
- $i \vdash s \stackrel{n}{\longmapsto} s'$ if $i \vdash s \rightarrow s'$ and $n \in \Xi$ where $s = (n, \sigma)$,
- $i \vdash s \stackrel{\tau}{\longmapsto} s'$ if $i \vdash s \rightarrow s'$ and $n \notin \Xi$ where $s = (n, \sigma)$,
- $\stackrel{\tau}{\Longrightarrow}$ for the reflexive transitive closure of $\stackrel{\tau}{\longmapsto}$, and
- $i \vdash s \stackrel{n}{\Longrightarrow} s'$ if there exists s_1 such that $s \stackrel{\tau}{\Longrightarrow} s_1$ and $s_1 \stackrel{n}{\longmapsto} s'$. □

Definition 21 A binary relation ϕ is a weak bisimulation if for all $i \in \{1, 2\}$, we have the following properties where $j = 3 - i$.

1. If $s_1 \, \phi \, s_2$ and $i \vdash s_i \stackrel{\tau}{\longmapsto} s_i'$ then there exists s_j' such that $j \vdash s_j \stackrel{\tau}{\Longrightarrow} s_j'$ and $s_1' \, \phi \, s_2'$.
2. If $s_1 \, \phi \, s_2$ and $i \vdash s_i \stackrel{n}{\longmapsto} s_i'$ then there exists s_j' such that $j \vdash s_j \stackrel{n}{\Longrightarrow} s_j'$ and $s_1' \, \phi \, s_2'$. □

Remark 2 The notion of weak bisimulation just defined is slightly different from what is mostly seen in the literature, in that $\stackrel{n}{\Longrightarrow}$ does not allow silent moves *after* the observable action. □

Remark 3 If Ξ is closed under $\stackrel{ntscd}{\rightarrow}$ and $\stackrel{dod}{\rightarrow}$, we know from Theorem 5 that for any node n, $obs^1_{may}(n)$ is either a singleton set or empty. With abuse of notation, we shall write $obs^1_{may}(n) = n_1$ for $obs^1_{may}(n) = \{n_1\}$. Also, we know from Lemma 4 that if $obs^1_{may}(n) = n_1$ then all maximal paths from n will contain n_1. □

Definition 22 For each node n in G, we define $relv(n)$, the set of relevant variables at n, by stipulating that x in $relv(n)$ iff there exists a node $n_k \in \Xi$ and a path π from n to n_k such that $x \in ref(n_k)$, but for all nodes n_j occurring before $n_k \in \pi$, $x \notin def(n_j)$. □

Strictly speaking, we should have defined (for i = 1,2) functions $ref_i(n)$ to return the variables referenced at node n wrt. code map $code_i$, functions $def_i(n)$ to return the variables defined at node n

wrt. code map $code_i$, and functions $relv_i(n)$ and relation $\stackrel{dd}{\rightarrow}_i$ parametrized wrt. ref_i and def_i. However, the following result shows that we can safely ignore the subscripts since the slicing transformation applied to S_C yields a node set that is also closed under data dependence and has the same set of relevant variables for each node.

Lemma 7 *Assume, with $\stackrel{dd}{\rightarrow}_i$ etc. as defined just above, that Ξ is closed under $\stackrel{dd}{\rightarrow}_1$. Then*

1. *Ξ is closed also under $\stackrel{dd}{\rightarrow}_2$;*
2. *for all n, $relv_1(n) = relv_2(n)$.* □

PROOF To show (1), assume the contrary; then there exists a path π from $n_j \notin \Xi$ to $n_k \in \Xi$ such that $x \in ref_2(n_k)$ and $x \in def_2(n_j)$, but for all n' interior in π: $x \notin def_2(n')$. Observing that all variables in $code_2$ also occur in $code_1$, we see that $x \in ref_1(n_k)$ and $x \in def_1(n_j)$. Since Ξ is closed under $\stackrel{dd}{\rightarrow}_1$, we can infer that there exists a node n' interior in π with $x \in def_1(n')$; let n_1 be the last such n'. Then $n_1 \in \Xi$ and $x \in def_1(n_1)$ and (cf. above) $x \notin def_2(n_1)$. But since $n_1 \in \Xi$ we have $code_1(n_1) = code_2(n_1)$ which yields the desired contradiction.

To show (2), assume that $x \in relv_i(n)$ with $i \in \{1, 2\}$; we must prove that $x \in relv_j(n)$ where $j = 3 - i$. Our assumptions are that there exists a path π from n to $n_k \in \Xi$ such that $x \in ref_i(n_k)$, but for all nodes n' occurring before n_k in π, $x \notin def_i(n')$. Now, since $n_k \in \Xi$, $code_i(n_k) = code_j(n_k)$, so $x \in ref_j(n_k)$. We are done if we can prove that $x \notin def_j(n')$ for all nodes n' occurring before n_k in π. In order to arrive at a contradiction, assume that this is not the case. Let n_1 be the last node n' occurring before n_k in π with $x \in def_j(n')$. Then $n_1 \stackrel{dd}{\rightarrow}_j n_k$; since $n_k \in \Xi$ which by (1) is closed under $\stackrel{dd}{\rightarrow}_j$, this implies $n_1 \in \Xi$. But then $code_j(n_1) = code_i(n_1)$ which gives the desired contradiction since $x \in def_j(n_1)$ but $x \notin def_i(n_1)$. ■

After this digression, we return the the main development, where a key property is that the set of relevant variables is determined by the first observable.

Lemma 8 *Assume that Ξ is closed under $\stackrel{ntscd}{\rightarrow}$, $\stackrel{dod}{\rightarrow}$, and $\stackrel{dd}{\rightarrow}$. Assume that n_1 and n_2 are such that $obs^1_{may}(n_1) = obs^1_{may}(n_2)$. Then $relv(n_1) = relv(n_2)$.* □

PROOF If $obs^1_{may}(n_1)$ and $obs^1_{may}(n_2)$ are both empty, no node in Ξ is reachable from n_1 nor from n_2, and therefore $relv(n_1) = relv(n_2) = \emptyset$.

Otherwise, let $n_3 = obs^1_{may}(n_1) = obs^1_{may}(n_2)$; for reasons of symmetry, it is sufficient to prove that $relv(n_1) \subseteq relv(n_2)$. So let $x \in relv(n_1)$ be given, we must prove $x \in relv(n_2)$. There exists a path π from n_1 to $n_k \in \Xi$ such that $x \in ref(n_k)$, but $x \notin def(n_j)$ for any node n_j occurring before n_k in π. Since $n_3 = obs^1_{may}(n_1)$, we can split π into $\pi_1 = [n_1..n_3]$ and $\pi_0 = [n_3..n_k]$. Since $n_3 = obs^1_{may}(n_2)$, there exists a repetition-free path $\pi_2 = [n_2..n_3]$, and thus a path $\pi' = \pi_2\pi_0$ from n_2 to n_k. Towards proving our goal $x \in relv(n_2)$, we are left with showing that $x \notin def(n_j)$ for all nodes n_j occurring before n_k in π'. Assume the contrary, and let n' be the last node in π' serving as a counterexample. Since Ξ is closed under $\stackrel{dd}{\rightarrow}$, we infer that $n' \in \Xi$; also, due to the properties of π, we infer that n' does not occur in π_0, and therefore, n' occurs before n_3 in π_2. But this contradicts $n_3 = obs^1_{may}(n_2)$. ■

We need one more auxiliary result.

Lemma 9 *Assume that* Ξ *is closed under* $\overset{ntscd}{\rightarrow}$, $\overset{dod}{\rightarrow}$, *and* $\overset{dd}{\rightarrow}$. *If* $i \vdash s_1 \overset{\tau}{\longmapsto} s_2$ *where* $s_1 = (n_1, \sigma_1)$, $s_2 = (n_2, \sigma_2)$, *and* $i \in \{1, 2\}$ *then*

1. $obs^1_{may}(n_1) = obs^1_{may}(n_2)$ *and*
2. *there exists* V *such that*
   (a) $V = relv(n_1) = relv(n_2)$ *and*
   (b) $\sigma_1 =_V \sigma_2$. □

Here we write $\sigma_1 =_V \sigma_2$ when for all $x \in V$, $\sigma_1(x) = \sigma_2(x)$.

PROOF First observe that $n_1 \notin \Xi$. For (1), clearly $obs^1_{may}(n_2) \subseteq obs^1_{may}(n_1)$, so by Theorem 5 it is sufficient to prove that it cannot be the case that $obs^1_{may}(n_2) = \emptyset$ while $obs^1_{may}(n_1)$ is a singleton $\{n_3\}$. But if so, Lemma 4 would tell us that $n_3 \in \Xi$ occurs on all maximal paths from n_1, and thus also on all maximal paths from n_2, contradicting $obs^1_{may}(n_2) = \emptyset$.

Now (a) follows from Lemma 8. For (b), in order to arrive at a contradiction, we assume that $\sigma_1 =_V \sigma_2$ does not hold. For this to be the case, there must exist $x \in V$ with $x \in def(n_1)$. Since $x \in relv(n_1)$, there exists a path from n_1 to a node $n_k \in \Xi$ with $x \in ref(n_k)$, along which x is not defined. But since x is defined at n_1, this yields the desired contradiction. ■

We now stipulate when a program state in the original program is related to a program state in the sliced program.

Definition 23 For Ξ closed under $\overset{ntscd}{\rightarrow}$, $\overset{dod}{\rightarrow}$, and $\overset{dd}{\rightarrow}$, we define a relation $R : s_1 \, R \, s_2$ iff

- $obs^1_{may}(n_1) = obs^1_{may}(n_2)$ and
- $\sigma_1 =_V \sigma_2$

where $s_1 = (n_1, \sigma_1)$ and $s_2 = (n_2, \sigma_2)$ and $V = relv(n_1) = relv(n_2)$. □

By Lemma 8, this is well-defined. We now state the key part of the correctness result:

Theorem 6 *If* Ξ *is closed under* $\overset{ntscd}{\rightarrow}$, $\overset{dod}{\rightarrow}$, *and* $\overset{dd}{\rightarrow}$, *then the relation* R *from Definition 23 is a weak bisimulation (cf. Definition 21).* □

PROOF For $i \in \{1, 2\}$ and $j = 3 - i$, we must show

1. If $s_1 \, R \, s_2$ and $i \vdash s_i \overset{\tau}{\longmapsto} s_i'$ then there exists s_j' such that $j \vdash s_j \overset{\tau}{\Longrightarrow} s_j'$ and $s_1' \, R \, s_2'$.
2. If $s_1 \, R \, s_2$ and $i \vdash s_i \overset{n}{\longmapsto} s_i'$ then there exists s_j' such that $j \vdash s_j \overset{n}{\Longrightarrow} s_j'$ and $s_1' \, R \, s_2'$.

For (1), assume that $i \vdash s_i \xrightarrow{\tau} s'_i$. Choose $s'_j = s_j$. The claim then trivially follows from Lemma 9.

For (2), assume that $i \vdash s_i \xrightarrow{n} s'_i$. Thus s_i is of the form (n, σ_i); also let $s_j = (n_j, \sigma_j)$ and $s'_i = (n', \sigma'_i)$. We have $n = obs^1_{may}(n) = obs^1_{may}(n_j)$; let $V = relv(n) = relv(n_j)$. Since by Lemma 4, $n \in obs^*_{must}(n_j)$, any execution sequence starting from n_j will sooner or later hit n; also, since n is the only node in $obs^1_{may}(n_j)$, that execution sequence will contain no other nodes in Ξ. All this shows that there exists $s''_j = (n, \sigma''_j)$ such that $j \vdash s_j \stackrel{\tau}{\Longrightarrow} s''_j$. By repeated application of Lemma 9 we infer $\sigma''_j =_V \sigma_j$ and since $\sigma_i =_V \sigma_j$ thus also $\sigma_i =_V \sigma''_j$. In particular,

σ_i and σ''_j agree on $ref(n)$. (*)

Therefore, s''_j will choose the same branch as s_i (if n is a predicate node, otherwise vacuously). That is, there exists s'_j of the form (n', σ'_j) such that $j \vdash s''_j \xrightarrow{n} s'_j$ and thus $j \vdash s_j \stackrel{n}{\Longrightarrow} s'_j$. We are left with showing that with $V' = relv(n')$ we have $\sigma'_i =_{V'} \sigma'_j$. So let $x \in V'$, we must prove $\sigma'_i(x) = \sigma'_j(x)$. If $x \in def(n)$ (and n is thus a statement node) then the claim clearly follows from (*). Otherwise, if $x \notin def(n)$, then $x \in relv(n) = V$ and the claim follows from $\sigma_i =_V \sigma''_j$ since $\sigma'_i(x) = \sigma_i(x) = \sigma''_j(x) = \sigma'_j(x)$. ■

Observe that R is reflexive. Therefore, by Theorem 6, the initial state of the original CFG is weakly bisimilar to the initial state of the sliced CFG. Also, since two states that are related by R produce the same "output", and since bisimulation generalizes Weiser's notion of projection [Wei84] to infinite traces, this demonstrates that

Theorem 7 *If Ξ is closed under $\stackrel{ntscd}{\rightarrow}$, $\stackrel{dod}{\rightarrow}$, and $\stackrel{dd}{\rightarrow}$, then the sliced program has the same "observable behavior" as the original program.* □

2.5 Algorithms

In this section we present algorithms to calculate various forms of control and order dependences that were presented earlier. Each algorithm is accompanied by an overview, a proof of correctness, and the complexity analysis of the worst-case time requirement. The algorithms are presented to suggest that the proposed dependences can be calculated by algorithms with time complexity that is polynomial in the number of nodes/edges. We conjecture that more optimal algorithms can be designed to calculate the same information.

2.5.1 Non-Termination Sensitive Control Dependence (NTSCD)

We adopt an approach similar to symbolic data-flow analysis to calculate control dependences. Basically, control dependences are determined by reasoning about properties of sets of CFG paths; those sets are represented symbolically in our algorithm. Specifically, for each node n with more than one successor in G, the set of all maximal paths that start with $n \rightarrow m$ is represented by a symbol t_{nm}. The algorithm propagates these symbols to collect the effects of particular control flow choices at program points in the CFG. For each node p, a set of symbols S_{pn} is maintained for every node n in the CFG that has more than one successor; these sets record the maximal paths that

originate from n and contain p. Hence, based on the interpretation, $t_{nm} \in S_{pn}$ indicates that all maximal paths starting with $n \rightarrow m$ contain p. We shall use T_n to denote the number of successors ($|succs(n, G)|$) of node n in G. Also, $condNodes(G)$ denotes the set of nodes in G that have multiple successors. The algorithm is presented in Figure 2.4.

Proof of correctness

The correctness of the algorithm (Figure 2.4) is presented as the following theorem.

Theorem 8 *Upon the termination of phase (2) of the algorithm, $t_{nm} \in S_{pn}$ iff all maximal paths starting with $n \rightarrow m$ contain p.* □

PROOF We shall use "only-if" direction as an invariant on the loops in phase (2). We shall then prove the "if" direction via contradiction.

"only-if" direction The finiteness of N ensures the termination of phase (1). Upon the completion of phase (1), the invariant is trivially established at the beginning of phase (2). If n has only one successor m then all maximal paths containing n will contain m. Hence, the assignment at line 21 establishes the invariant at the end of the loop at line 19 (and conditional at line 17). If n has multiple successors and all maximal paths through the successors contain m then all maximal paths containing n will also contain m. This is captured by the assignment at line 29 and the invariant is established at the end of the loops at line 25 and 27.

As the graph has finite number of nodes, the number of successors of a node is finite. Hence, the total number of symbols (t_{nm}) in the G is finite as well. This implies that the size of S_{nm} has a finite bound for every pair of nodes, n and m. In each iteration of the `while` loop at line 14, either a symbol set S_{nm} increases in size or all of the symbol sets remain unchanged. The former case contributes an iteration (line 22 and line 30). As the size of the symbol set is finitely bound, the `while` loop in line 14 will terminate establishing the "only-if" direction.

"if" direction Suppose there are nodes n, m, and p such that all maximal paths starting with $n \rightarrow m$ contain p but $t_{nm} \notin S_{pn}$. This implies that, in every maximal path starting with $n \rightarrow m$, ending with p, and containing nodes q and r (in the given order), $t_{nm} \in S_{in}$ for every node from m to q and $t_{nm} \notin S_{jn}$ for every node from r to p. We consider two cases.

- *r is the only successor of q.* In this case, when t_{nm} is injected into S_{qn}, q will be marked for processing (line 21). Upon processing, t_{nm} will be injected into S_{rn}. Hence, the supposition cannot be true.
- *q has multiple successors.* By the supposition, there should be a node that is the first common node to occur on all maximal paths originating from the successors of q. Let r be this common node. Also assume there are no conditional nodes in the paths from q to r. From the previous results and non-branching property of the paths between q and r, $|S_{rq}| = T_q$. This implies $S_{qn} \subseteq S_{rn}$, hence, the supposition is falsified. Similar reasoning can be applied inside-out when conditional nodes occur on the paths from q to r.

The above reasoning can be applied inductively when r is not the immediate successor of q or when r is not the first common node to occur on all maximal paths originating from the successors

```
NON-TERMINATION-SENSITIVE-CONTROL-DEPENDENCE(G)
 1  G(N, E, n_0) : a control flow graph
 2  S[|N|, |N|] : a matrix of sets where S[p, n] represents S_pn
 3  CD[|N|] : a sequence of sets
 4  workbag : a set of nodes
 5
 6   # (1) Initialize
 7  workbag ← ∅
 8  for each n in condNodes(G)
 9  do for each m in succs(n, G)
10      do S[m, n] ← {t_nm}
11          workbag ← workbag ∪ {m}
12
13   # (2) Calculate all-path reachability
14  while workbag ≠ ∅
15  do n ← remove(workbag)
16       # (2.1) One successor case
17      if T_n = 1 and n ∉ succs(n, G)
18        then m ← select(succs(n, G))
19               for p in condNodes(G)
20               do if S[n, p]\S[m, p] ≠ ∅
21                     then S[m, p] ← S[m, p] ∪ S[n, p]
22                          workbag ← workbag ∪ {m}
23       # (2.2) Multiple successors case
24      if |succs(n, G)| > 1
25        then for m in N
26               do if |S[m, n]| = T_n
27                     then for p ∈ condNodes(G)\{n}
28                          do if S[n, p]\S[m, p] ≠ ∅
29                               then S[m, p] ← S[m, p] ∪ S[n, p]
30                                    workbag ← workbag ∪ {m}
31   # (3) Calculate non-termination sensitive control dependence
32  for each n in N
33  do for each m in condNodes(G)
34      do if 0 < |S[n, m]| < T_m
35           then CD[m] ← CD[m] ∪ {n}
36
37  return CD
```

Figure 2.4: Algorithm to calculate non-termination sensitive control dependence

of q. ■

Based on the interpretation attached to t_{mn} and S_{pn} and Theorem 8, it is trivial to see that phase (3) correctly calculates non-termination sensitive control dependence.

Complexity analysis

Phases (1) and (3) of the algorithm (Figure 2.4) have a worst case complexity of $O(|N|^2 \times \lg |N|)$ where $\lg |N|$ is the complexity of set operations. The complexity of the loop at line 25 is $O(|N|^2 \times \lg |N|)$ and it dominates the complexity of the loop at line 14. In the worst case in phase (2), for a node p, all token sets $S[p, i]$ of p will stabilize in $\sum T_n$ iterations. Hence, the overall complexity of phase (2) will be $O(\sum T_n \times |N|^3 \times \lg(|N|)$. This will also be the overall complexity of the algorithm.

2.5.2 Non-Termination Insensitive Control Dependence (NTICD)

The proposed algorithm (Figure 2.5) to calculate non-termination insensitive control dependence is very similar to the NTSCD algorithm. The only differences being the presence of phase (2.3) and the interpretation attached to t_{nm}. In the NTSCD algorithm, any token t_{nm} injected into S_{nn} is not propagated to non-m successors of n, hence, preserving non-termination sensitivity. Phase (2.3) in NTICD algorithm induces non-termination insensitivity by undoing this preservation. Also, t_{nm} represents all extensible finite paths starting with $n \rightarrow m$ in NTICD algorithm.

Proof of correctness

Given the similarity of NTSCD and NTICD algorithms, we prove the correctness of NTICD algorithm by proving that phase (3) along with the interpretation attached to t_{nm} calculates non-termination insensitive control dependence.

The key observation being that phase (2.3) induces non-termination insensitivity. Succinctly, if $t_{nm} \in S_{nn}$ then t_{nm} is added to S_{pn} where p is a successor of n. Thus establishing that all finite paths that start with $n \rightarrow m$ and that reach n can be finitely extended to reach p, hence, inducing non-termination insensitivity.

Lemma 10 *If $t_{nm} \in S_{pn}$ and p belongs to a control sink then for all nodes $q \in$ c-sink(p).$t_{nm} \in S_{qn}$* □

Proof If $|$c-sink$(p)| \leq 1$ then we are done. If $|$c-sink$(p)| > 1$, then let q be a node such that $q \in$ c-sink(p) and $t_{nm} \notin S_{qn}$. Since q and p belong to the same control sink, all finite paths from p can be extended to q. Hence, $S_{pn} \subseteq S_{qn}$. Similarly, we can prove $S_{qn} \subseteq S_{pn}$. Hence, $S_{pn} = S_{qn}$. ■

Theorem 9 *Phase (3) of NTICD calculates non-termination insensitive control dependence.* □

Proof $t_{nm} \in S_{pn}$ implies that all finite paths starting with $n \rightarrow m$ can be extended to p. Hence, $0 < |S_{mn}| < T_n$ implies that there are some successors m of n for which all finite paths starting at m can be extended to p while, for some successors q, not all finite paths starting at q can be extended to p. Hence, $n \overset{ntscd}{\rightarrow} p$.

```
NON-TERMINATION-INSENSITIVE-CONTROL-DEPENDENCE(G)
 1  G(N, E, n_0) : a control flow graph
 2  S[|N|, |N|] : a matrix of sets where S[p, n] represents S_pn
 3  CD[|N|] : a sequence of sets
 4  workbag : a set of nodes
 5
 6   # (1) Initialize
 7  workbag ← ∅
 8  for each n in condNodes(G)
 9  do for each m in succs(n, G)
10      do S[m, n] ← {t_nm}
11          workbag ← workbag ∪ {m}
12
13   # (2) Calculate all-path reachability
14  while workbag ≠ ∅
15  do n ← remove(workbag)
16       # (2.1) One successor case
17      if T_n = 1 and n ∉ succs(n, G)
18        then m ← select(succs(n, G))
19             for p in condNodes(G)
20             do if S[n, p]\S[m, p] ≠ ∅
21                  then S[m, p] ← S[m, p] ∪ S[n, p]
22                       workbag ← workbag ∪ {m}
23       # (2.2) Multiple successors case
24      if |succs(n, G)| > 1
25        then for m in N
26             do if |S[m, n]| = T_n
27                  then for p in condNodes(G)\{n}
28                       do if S[n, p]\S[m, p] ≠ ∅
29                            then S[m, p] ← S[m, p] ∪ S[n, p]
30                                 workbag ← workbag ∪ {m}
31       # (2.3) Erase non-termination sensitivity
32      if |S[n, n]| > 0
33        then for m in succs(n, G)\n
34             do if S[n, n]\S[m, n] ≠ ∅
35                  then S[m, n] ← S[m, n] ∪ S[n, n]
36                       workbag ← workbag ∪ {m}
37
38   # (3) Calculate non-termination insensitive control dependence
39  for each n in N
40  do for each m in condNodes(G)
41      do if 0 < |S[n, m]| < T_m
42           then CD[m] ← CD[m] ∪ {n}
43
44  return CD
```

Figure 2.5: Algorithm to calculate non-termination insensitive control dependence

When $|S_{pn}| = 0$ or $|S_{pn}| = T_n$, it implies that for all successors of n either none or all finite paths can be extended to contain p. Hence, $n \overset{ntscd}{\not\rightarrow} p$. Also, by Lemma 10, $|S_{pn}| = T_n$ for all conditional nodes n in the control sink of p, hence, $n \overset{ntscd}{\not\rightarrow} p$.

So, phase (3) correctly calculates non-termination insensitive control dependence. ■

Complexity analysis

Phase (2.3) of NTICD algorithm contributes $O(\sum T_n \times |N| \times \lg(|N|))$ to the overall complexity of phase (2) of NTSCD algorithm. As $O(\sum T_n \times |N|^3 \times \lg(|N|)$ dominates $O(\sum T_n \times |N| \times \lg(|N|)$, the overall complexity of NTICD algorithm is identical as that of NTSCD algorithm.

2.5.3 Decisive Control Dependence (DCD)

As Definition 11 implies Definition 7, we calculate decisive control dependence by pruning non-termination sensitive control dependence. It is evident that clause (2) in Definition 11 is a stronger than that in Definition 7. Hence, we use the negative form of clause (2) in Definition 11 – for all successors n_l of n_i, there exists a maximal path such that n_j occurs before any occurrence of n_i – to prune non-termination sensitive control dependence to calculate decisive control dependence.

In the algorithm (Figure 2.6), t_{nm} represents a path π that starts with $n \rightarrow m$ and is maximal or terminates with n while $t_{nm} \in S_{pn}$ represents that a path starting with $n \rightarrow m$ that can be extended to contain p. In phase (2) of the algorithm, tokens are propagated to calculate reachability between conditional nodes and other nodes of the G. This information is later used in phase (3) to calculate decisive control dependence.

Proof of correctness

To prove the correctness of the DCD algorithm, it is sufficient to prove that phase (2) of the algorithm calculates reachability between the successors of the conditional nodes and the other nodes of G.

Theorem 10 *At the end of phase(2) in the DCD algorithm, $t_{nm} \in S_{pn}$ iff there exists a path starting with $n \rightarrow m$ that can be extended to p.* □

Proof We shall use the "only-if" direction as an invariant on the loop in phase (2). We shall then prove the "if" direction via contradiction.

"only-if" direction As the number of edges in the G is finite, phases (1) will terminate. The invariant is trivially established at the beginning of phase (2). The loops at line 18 and 19 extend a path starting with $n \rightarrow m$ and leading to p to every successor q of p, if it has not already been extended. Also, q is queued for processing at line 22. Hence, at the end of the loop, the invariant is established.

Each iteration of the outer `while` loop at line 16 in phase (2) will either result in the increase in size of a symbol set while contributing an iteration or there will be no change in the data. The size

```
DECISIVE-CONTROL-DEPENDENCE(G)
 1  G(N, E, n_0) : a control flow graph
 2  S[|N|, |N|] : a matrix of sets where S[n_1, n_2] represents S_{n_1 n_2}
 3  T[|N|] : a sequence of integers where T[n_1] denotes T_{n_1}
 4  CD[|N|] : a sequence of sets
 5  workbag : a set of nodes
 6
 7   # (1) Initialize
 8  workbag ← ∅
 9  for each n in condNodes(G)
10  do succs ← succs(n, G)
11      for each m in succs
12      do workbag ← workbag ∪ {m}
13          S[m, n] ← {t_nm}
14
15   # (2) Calculate exists-a-path reachability
16  while workbag ≠ ∅
17  do n ← remove(workbag)
18      for each m in succs(n, G)
19      do for each p in condNodes(G)
20          do if S[n, p]\S[m, p] ≠ ∅
21                then S[m, p] ← S[m, p] ∪ S[n, p]
22                     workbag ← workbag ∪ m
23
24   # (3) Calculate decisive control dependence
25  CD ← NON-TERMINATION-SENSITIVE-CONTROL-DEPENDENCE(G)
26  for each n in N
27  do for each m in CD[n]
28      do if |S[n, m]| = T_m
29            then CD[n] ← CD[n]\{m}
30
31  return CD
```

Figure 2.6: Algorithm to calculate decisive control dependence

of the symbol sets are finite as the tokens/symbols in the G are finite. Hence, the outer `while` loop in phase (2) will terminate.

"if" direction Upon termination of phase (2), suppose that there are nodes n, m, and p such that there exists a path starting with $n \rightarrow m$ that contains p but $t_{nm} \notin S_{pn}$. This implies that, along a path starting with $n \rightarrow m$ and containing p, there should be two consecutive nodes q and r, in the given order, such that $t_{nm} \in S_{qn}$ and $t_{nm} \notin S_{rn}$. However, this leads to a contradiction as, upon termination of phase (2), the condition on line 20 will evaluate to false for all nodes in the G. Hence, the supposition cannot be true. ■

Complexity analysis

Based on the structure of phase (2), it is trivial to see that the complexity of DCD algorithm is identical to that of NTSCD algorithm.

2.5.4 Decisive Order Dependence (DOD)

Given nodes $n = n_1, m = n_2$, and $p = n_3$, we need to check if the three clauses in the definition of decisive order dependence[7] are satisfied. We can use information from graph reachability algorithm to check if m and p satisfy first clause in Definition 12 (as done in the first and second conjuncts on line 6 of `order-dependence()`).

As for the second and third clauses, we encode the order dependence calculation as a problem of constructing colored bound directed acyclic graph (DAG). The bounding condition is that out-going edges of m and p are not explored. The coloring condition contains three parts: (1) m and p are assigned colors *white* and *black*, respectively; (2) Every node in the DAG is colored *white(black)* iff only if all its children are colored *white(black)*; and (3) Nodes with children of different colors, all uncolored children, and/or nodes that are sources of back edges are *uncolored*.

Given such a colored bound DAG rooted at n, it is trivial to observe that, for an acyclic graph, a node q will be colored *white(black)* only if all of its successors are colored *white(black)*. Given the encoding, this implies all maximal paths from q contain $m(p)$ before any occurrence of $p(m)$. Hence, we can conclude that m and p are *decisively order dependent* on any node n that has at least one *black* child and at least one *white* child.

In case of a cyclic graph, the source q of a back edge is *uncolored* indicating the existence of a maximal path that does not contain $m(p)$. In such cases, given the coloring condition, every ancestor of q will be *uncolored*, hence, falsifying clauses (2) and (3) of Definition 12.

Proof of correctness

Based on the above description/intuition, we need to prove that the coloring and bounding of the DAG does indeed capture the information required to decide if $n \overset{dod}{\rightarrow} m \leftrightharpoons p$. We shall prove the correctness of the algorithm by proving the following theorems.

[7] In this subsection, we shall refer to decisive order dependence as order dependence.

```
ORDER-DEPENDENCE()
 1  OD[|N|][|N|] : a matrix that captures order dependence
 2  G(N, E, n_0) : a control flow graph
 3  for each n in condNodes(G)
 4  do for each m in N
 5     do for each p in N\{m}
 6        do if REACHABLE(m, p, G) ∧ REACHABLE(p, m, G) ∧ DEPENDENCE(n, p, m, G)
 7              then OD[m][p] = OD[m][p] ∪ {n}
 8  return OD

DEPENDENCE(n, m, p, G)
 1  color[|N|] : a sequence of values ranging over {unknown, white, black}
 2  for each q in N
 3  do color[q] ← uncolored
 4  color[m] = white
 5  color[p] = black
 6  visited ← {m, p}
 7  COLORED-DAG(G, n, color, visited)
 8  whiteChild ← false
 9  blackChild ← false
10  for each q in succs(n, G)
11  do if color[q] = white
12        then whiteChild ← true
13     if color[q] = black
14        then blackChild ← true
15  return whiteChild ∧ blackChild

COLORED-DAG(G, n, color, visited)
 1  if n ∉ visited
 2    then visited ← visited ∪ {n}
 3         for each q in succs(n, G)
 4         do COLORED-DAG(G, q, color, visited)
 5         c ← color[select(succs(n, G))]
 6         for each q in succs(n, G)
 7         do if color[q] ≠ c
 8               then c ← uncolored
 9                    break
10         color[n] ← c
11  return
```

Figure 2.7: Algorithm to calculate decisively strong order dependence

Theorem 11 *Given a CFG G, a white node, and a black node,* `colored-dag()` *creates a colored bound DAG such that*

1. *a node is colored white if all its immediate successors are colored white,*
2. *a node is colored black if all its immediate successors are colored black,*
3. *a node is uncolored if (1) all its immediate successors are uncolored, it has at least two children of different colors, or it is the source of a back edge in G.* □

PROOF It is trivial to see (by induction) that `colored-dag()` will visit all unvisited nodes reachable from the given node n as in a depth-first search. As each visited node is recorded in `visited`, the bounding condition is established by the addition of m and p to `visited` at line 4 and 5 of `dependence()` and maintained by the check at line 1 of `colored-dag()`. This record keeping along with the finiteness of nodes in the CFG ensures the termination of `colored-dag()`.

After every child of node n has been fully explored in the loop at line 3 in `colored-dag()`, the color of n is determined by the loop at line 6 in the same procedure. The loop will terminate normally only when the color of every child of n is the same as the color of an arbitrarily chosen child at line 5. The abnormal termination of the same loop (via `break`) indicates that there are at least two children of the node that have different colors. In situations where one of the successor q is a visited but partially explored node, the color of q will be *uncolored* due to initialization at line 3. Hence, either the loop at line 6 will terminate abnormally or terminate normally (when every child of n was *uncolored*) and color n as *uncolored*. ■

Lemma 11 *In the colored bound DAG constructed by* `colored-dag()`*, a node n is white(black) iff all nodes reachable from n in the DAG are white(black).* □

Theorem 12 *Given a colored bound DAG created by* `colored-dag()` *from CFG G,* `dependence()` *returns true iff clauses (2) and (3) of Definition 12 are satisfied in G.* □

PROOF At the beginning of `dependence()`, m and p are designated as the *white* and *black* nodes, respectively. After `colored-dag()` returns on line 7 of `dependence()`, let q and r be immediate successors of n such that q is *white* and r be *black*.

"only-if" direction From Lemma 11, on all paths in the DAG from $q(r)$, $m(p)$ will be encountered before any $p(m)$ is encountered. The absence of *uncolored* nodes on such paths rules out the possibility of an infinite path from $q(r)$ that does not contain the $m(p)$. Hence, for all maximal paths from $q(r)$ in G, $m(p)$ will be encountered before any $m(p)$ is encountered. Thus q and r satisfy clauses (2) and (3) of Definition 12, respectively, when `dependence()` returns true.

"if" direction Suppose all maximal paths from $q(r)$ contain $m(p)$ before any occurrence of $p(m)$. This implies that there can be no node n_i on any path between $q(r)$ (inclusive) and $m(p)$ (exclusive) such that n_i has an out-going edge that can lead to a cycle not containing $m(p)$. Hence, all nodes on these paths can be colored *white(black)*. As a DAG rooted at n will not contain back-edges leading to infinite paths and as no such edges emanate from nodes on the paths between $q(r)$ (inclusive) and $m(p)$ (exclusive), `colored-dag()` will achieve the coloring as described above. Hence, `dependent()` will return true when q and r satisfy clauses (2) and (3) of Definition 12. ■

Complexity analysis

`colored-dag()` will be executed at least for every edge in the graph. As line 7 in `colored-dag()` can be executed $|N|$ times for each execution of `colored-dag()`, the worst-case complexity of *colored-dag()* will be $O(|E| \times |N| \times \lg(N))$.

The conditional at line 11 in `dependence()` can execute $|N|$ times for each execution of `dependence()`. By factoring in the complexity of `colored-dag()`, the worst-case complexity of `dependence()` will be $O(|N| + |E| \times |N| \times \lg(N)) = O(|E| \times |N| \times \lg(N))$.

The worst-case complexity of graph reachability algorithm is $O(|N|^3)$. The loops at line 3, 4, and 5 in `order-dependence()` will contribute $|N|^3$ iterations. Hence, the worst-case complexity of `order-dependence()` will be $O(|N|^3 + |N|^3 \times |E| \times |N| \times \lg(N)) = O(|N|^4 \times |E| \times \lg(N))$.

2.6 Related Work

Fifteen years ago, control dependence was rigorously explored by Podgurski and Clarke [PC90]. Since then there has been a variety of work related to calculation and application of control dependence in the setting of CFGs that satisfy the unique end node property.

In the realm of calculating control dependence, Johnson et al.[JP93] proposed an algorithm that could be used to calculate control dependence in time linear in the number of edges. Later, Bilardi et al.[BP96] proposed new concepts related to control dependence along with algorithms based on these concepts to efficiently calculate weak control dependence. In comparison, in this chapter we sketch a feasible algorithm in a more general setting.

In the context of slicing, Horwitz, Reps, and Binkley [HRB90] presented what has now become the standard approach to inter-procedural slicing via dependence graphs. However, in the last decade, C++, Java, and other languages that support semantically different exit points (exceptional and normal) to a procedure have become prominent. Hence, the work of Horwitz et al.cannot be applied directly as data dependence changes due to the semantic differences between the exit points/statements. This issue was recently addressed by Allen and Horwitz [AH03]. In their effort, they extend the previous work [HRB90] to handle exception-based inter-procedural control flow. In their work, they inject normal exit nodes and exceptional exit nodes in the CFG, but then preserve the *unique exit node* property by connecting the normal and exceptional exit node to the unique exit node. They also consider the first statements of `try` and *catch* blocks and *throw* statements as predicate statements. In contrast, our approach is simpler as the CFG is unaltered even in case of exceptional exit nodes and/or multiple normal exit nodes.

As for control dependence across procedure boundaries, the naive approach of considering the invocation site as a predicate (Soot [8] and [AH03]) and relating the `catch` statement with the corresponding `throw` statement via data dependence would suffice. If more accuracy is required, then our definitions can be trivially applied to a collection of CFGs by tweaking the proposed algorithms to utilize the information about the connectivity between the nodes of different CFGs being considered.

In relevant efforts concerning slicing correctness, Horwitz et al.[HPR89] use a semantics based multi-layered approach to reason about the correctness of slicing in the realm of data dependence. Alternatively, Ball et al.[BH93] used a program point specific history based approach to prove the

[8] http://www.sable.mcgill.ca/soot/

correctness of slicing for arbitrary control flow. We build off of that work to consider arbitrary control flow without the unique end-node restriction. Their correctness property is a weaker property than bi-simulation – it does not require ordering to be maintained between observable nodes if there is no dependence between these nodes – and it holds for irreducible CFGs. For irreducible graphs, we need the extra notion of "order-dependency" to achieve the stronger correctness property.

In terms of handling dependences in a concurrent setting, Krinke [Kri98] considered static slicing of multi-threaded programs with shared variables, and focused on issues associated with inter-thread data dependence but did not consider non-termination sensitive forms of control dependence. Millett and Teitelbaum [IT98] studied static slicing of Promela (the model description language for the model-checker SPIN) and its application to model checking, simulation, and protocol understanding. They reused existing notions of slicing that, as we argue in this chapter, do not account for the subtleties of multi-threaded execution. They did not discuss the appropriateness of those notions for an inherently multi-threaded language like Promela, nor did they formalize a notion of correct slice for their applications. Hatcliff et al.[HCD+99] presented notions of dependence for concurrent CFGs to capture Java-like synchronization primitives. They proposed a notion of bi-simulation as the correctness property, but they did not provide a detailed definition or proof of correctness as has been done in this chapter.

Chapter 3

Data-based Dependence

Data dependence is a notion that relates program points based on the data defined and used in them. This notion is also referred to as *def-use* relation, and it has been studied for over three decades by the program analysis community. Data dependence has been heavily used in compiler optimization, testing, refactoring, and other program analysis enabled applications.

As computer languages have evolved, the focus of the program analysis community has shifted from purely scalar-based sequential programs to scalar- and heap-based sequential programs. One such specific focus that is relevant in the context of this dissertation is the *design of a scalable and accurate static alias analysis for heap allocated data — calculate if two variables may refer to the same run- time object.* Over the past decade, the same focus has been extended to concurrent programs as well. In this pursuit, various efforts have proposed *escape analysis — detect if objects escape their creation scope in concurrent programs* — that can be leveraged in the calculation of aliasing information in concurrent programs.

This chapter describes an escape analysis that is focused on capturing inter-thread interactions that are based on shared data; hence, the title of the chapter. The chapter also presents details of how the escape analysis can be leveraged to improve the accuracy of data dependence in concurrent inter-procedural programs in the presence of dynamically allocated data. Various optimizations and extensions to the analysis are also presented. The scalability and accuracy of the analysis and its optimization and extensions are empirically illustrated.

An preliminary exposition about the analysis presented in this chapter was published under the title *Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs – Venkatesh Prasad Ranganath and John Hatcliff* [RH04] at the *13th International Conference on Compiler Construction (CC)* held as part of ETAPS 2004.

3.1 Background

3.1.1 Identifier-based Data Dependence (IBDD)

Compiler optimizations such as constant propagation rely on *def-use* information. Consider the program in Figure 3.1 as an example. The variable `c` is *defined* upon entering the procedure

```
1   public class Example1 {
2     void tempConv(int c) {
3       float f = c * 9 / 5 + 32;
4       if (f < 100) {
5         System.out.println (" Good Weather " + f);
6       } else {
7         System.out.println (" Hot Weather " + c);
8       }
9       c = 20;
10    }
11  }
```

Figure 3.1: A trivial example to illustrate data dependence.

`tempConv(int)`. The definition of `c` at procedure entry will be used/accessed in the definition of `f` at line 2 as it will be executed immediately upon entering the procedure. In other words, the use of `c` at line 2 is dependent on the definition of `c` at procedure entry. Similarly, the use of `f` at line 3, 4, and 6 is dependent on the definition of `f` at line 2. However, none of the uses of the variable `f` depends on the definition of `f` at line 7 due to the lack of a control flow path along which this definition can *reach* any of these use sites.

As mentioned in Definition 2 in Chapter 2, a program point e_2 is *data dependent* on program point e_1 ($e_1 \overset{dd}{\rightarrow} e_2$) if e_1 defines a variable v, e_2 uses the variable v, and there exists a control flow path in the program from e_1 to e_2 such that none of the intermediate program points on this path define variable v.

In terms of implementation, data dependence analysis will need to verify the existence of a control flow path without any "killing" definitions of the variable of interest. However, this approach can be simplified if a program can be represented in *single static assignment (SSA)* form – *every variable is defined utmost once* [Muc97]. In this form, *every use of variable v depends on the only definition of variable v.* In cases where ϕ-nodes[1] occur in the SSA form of a program, the use nodes dependent on the phi-node can be made to depend on the source nodes of the ϕ-node without loss of accuracy.

In the presence of procedures, data dependence can get complicated as a procedure may be invoked at multiple locations. However, the above approach to analysis will suffice if the *call graph* of the program and the variable renaming across procedure boundaries are available and leveraged by the analysis. The accuracy of such an approach will depend on the accuracy of the call graph while its complexity will depend on the number of inter-procedural control flow paths to be evaluated.

Independent of the procedural-ness of the program, the approach primarily relies on the *identifiers of the variables* occurring in various definitions and use sites. Hence, such data dependence is referred to as *identifier-based data dependence.*

3.1.2 Effects of Aliasing

In object oriented languages, *aliasing is the situation when more than one variable refers to an object.* In the presence of aliasing, the changes made to an object o through a variable v can be

[1] A ϕ-node represents the merging of two version of a variable at control flow join points, e.g. at the end of *if-else* statements.

```
1   class Name {
2     String  first ;
3     String  last ;
4   }
5
6   public class Example2 {
7     public static void main(String[] s) {
8       Name n = new Name();
9       Name a = name;
10      n. first  = ‘‘John’’;
11      n.second = ‘‘Doe’’;
12      System.out.println(a.last + ‘‘, ’’ + a.first);
13    }
14  }
```

Figure 3.2: An example to illustrate the effects of aliasing on data dependence in an *intra-procedural* setting.

"viewed" via an alias a to o.

For example, in the program in Figure 3.2, the changes to the field *first* of the object referred to by variable n at Line 12 is "visible" via the alias a (via $a.first$) at line 12.

In the presence of aliasing, data dependence analysis cannot be based on the above the approach used for IBDD involving scalar variables. Instead, the approach should be extended to consider all definitions via identifiers of equivalent (based on aliasing) variables, e.g. $n.first$ and $a.first$ should be considered as equivalent variables. However, unlike in the case of IBDD involving scalar variables, aliasing can cause the violation of the single static assignment (SSA) property required by the approach. Hence, to be accurate, the data dependence analysis will need to resort to an expensive control flow path based approach. Even this approach will suffer from the common inaccuracies stemming from the combination of the abstraction of run time objects and multiple execution (e.g. being embedded in a loop, being executed by multiple invocations of the enclosing method) of object allocation sites.

In the rest of this chapter, we shall refer to the aliasing based data dependence in sequential contexts as *aliasing-based data dependence (ABDD)*.

In Inter-procedural Contexts

In the presence of procedures, aliasing may span across procedural boundaries. For example, in Figure 3.3, the variable `n/m` in the method `setName(Name)/printName(Name)` is an alias to the variable `name` in the method `main(String[])`. Consequently, the variable `m` in the method `printName(Name)` will be an alias to the variable `n` in the method `setName(Name)`. The former form of alias (e.g. between `name` and `n/m`) is referred to as *vertical alias* as it occurs along the length of a call stack while the latter form of alias (e.g. between `n` and `m`) is referred to as *horizontal alias* as it spans across two call stacks.[2]

Suppose `setName(Name)` was inlined Figure 3.3. Now vertical aliasing exists between the inlined code and code in `printName(Name)` and, hence, the resulting data dependence could be detected

[2]Defining these aliases using the relationship captured in a call graph can be complicated and/or ambiguous.

```
1   public class Example3 {
2     public static void main(String[] s) {
3       Name name = new Name();
4       setName(name);
5       printName(name);
6     }
7
8     void setName(Name n) {
9       n. first  = ‘‘John’’;
10      n.second = ‘‘Doe’’;
11    }
12
13    void printName(Name m) {
14      System.out.println(m.last + ‘‘, ’’ + m.first);
15    }
16  }
```

Figure 3.3: An example to illustrate the effects of aliasing on data dependence in an *inter-procedural* setting.

by tracking the variable renaming across procedure boundaries. However, this approach will suffer from the same drawbacks as in the intra-procedural context.

In the absence of such inlining (dealing with the unmodified program), the above approach fails to handle horizontal aliasing. The main issue being the disconnect of variable renaming across different procedure boundaries. This can be addressed by maintaining the history of variable renamings at various procedure invocation sites along a inter-procedural control flow path during the analysis. For example, in Figure 3.3, the analysis will need to record that the fields of object referred to by variable `name` in the method `main(String[])` are vertically aliased by variable `n` in the method `setName(Name)` during the definition of `first` and `name` field at lines 9 and 10. The analysis will then need to combine the recorded information along with a valid control flow path to establish the data dependence of `m.last` and `m.first` at line 14 in the method `printName(Name)` on lines 9 and 10.

The accuracy of the above approach will be proportional to the accuracy of the information used to statically determine if variables involved at def-use sites are aliases. The accuracy of the call graph information will affect the number of inter-procedural control flow paths considered to calculate data dependence and, hence, the complexity of the approach.

In Concurrent Contexts

In concurrent programs, data dependence that spans across process/thread boundaries is referred to as *interference dependence (ID)* [Kri98]. These dependences occur due to the sharing of data between processes/threads via aliasing. Although an approach similar to that used in sequential inter-procedural contexts can be used in concurrent inter-procedural contexts, the accuracy and the complexity of such an approach will depend on the cost of calculating information about features such as aliasing, dynamic process creation, and process-specific call graphs.

Another form of dependence occurs in concurrent contexts due to aliasing, shared data, and *locking* and *conditional waiting* based synchronization. This form of dependence is referred to as

```
1   class Acct {
2     protected int balance;
3     Acct() {
4       balance = 100;
5     }
6   } // End of class Acct
7
8   class Husband extends Thread {
9     protected Acct savings;
10    protected Acct checking;
11    Husband(Acct act) {
12      this.savings = act;
13      checking = new Acct();
14    }
15    public void run() {
16      savings.balance += 20;
17      synchronized(savings) {
18        savings.notify();
19      }
20      checking.balance += 10;
21    }
22  } // End of class Husband
23
24  public class Home {
25    public static void main(String[] s) {
26      Acct savings = new Acct();
27      Thread wife, husband;
28      wife = new Wife(savings);
29      husband = new Husband(savings);
30      wife.start();
31      husband.start();
32    }
33
34    public static void mainAlt(String[] s) {
35      Acct savings = new Acct();
36      Thread wife, husband;
37      wife = new Wife(savings);
38      husband = new Husband(savings);
39      for(int i = 2; i <= 3; i++) {
40        if (i % 2 == 0)
41          wife.start();
42        if (i % 3 == 0)
43          husband.start();
44      }
45    }
46  } // End of class Home
47
48  class Wife extends Thread {
49    protected Acct savings;
50    protected Acct checking;
51    Wife(Acct act) {
52      this.savings = act;
53      checking = new Acct();
54    }
55    public void run() {
56      Acct newAcct = new Acct();
57      synchronized(savings) {
58        savings.wait();
59      }
60      savings.balance -= 20;
61      synchronized(checking) {
62        checking.wait();
63      }
64      checking.balance -= 10;
65      newAcct.balance += 10;
66    }
67  } // End of class Wife
```

Figure 3.4: A simple Java program illustrating object sharing and synchronization between threads.

ready dependence and it was introduced by Hatcliff et al. [HCD+99] in the context of Java.

3.2 Motivation

3.2.1 Data flow across threads in Java

Interference dependences can only arise when an update is made to a data item that is "shared" between two or more threads. Thus, let us begin by considering how an object can become shared by flowing from one thread to another using the simple program given in Figure 3.4 (ignore the `mainAlt(String[])` method for now). Java[GJS00] supports threads via `java.lang.Thread` objects.[3] The example program has three threads: the main thread which executes the method

[3]The term *thread* indicates an execution entity and the term *thread object* indicates an instance of `java.lang.Thread` or any of it's subclasses.

`main(String[])` in class `Home`, a thread associated with an instance of `Husband`, and a thread associated with an instance of `Wife`. In Java, a thread comes into existence when `java.lang.Thread.start()` is executed on the associated Thread object (e.g. the calls to `start()` at line 30 and line 31). This dispatch results in the invocation of `run()` on the receiver of the `start()` method (e.g. the call to `start()` at line 31 causes the newly created thread to execute the `run()` method at line 15).

Let us now consider all possible ways in which a heap object o created by a thread t can be communicated to a newly created thread t'. Java disallows any arguments to `run()` or `start()` methods of `java.lang.Thread`. Thus, the only way to provide data to the *run* method of t' is to make the data reachable[4] from the `this` variable of the `Thread` instance of t' (e.g. from the instance of `Husband/Wife` class) or to have it be reachable from static fields (in essence, global variables). Note also that there are only two ways to assign data/values to instance fields of an object. One is via direct assignments to reachable fields. The other is via invocation of an method in which the fields reachable via the arguments (including the receiver in instance methods) will be assigned data reachable from the arguments to the method. For instance, in the example program, the `main` thread communicates the *savings* account object to the `run()` method of the `Wife` thread by invoking the constructor of `Wife` with the *savings* account as an argument. The constructor performs the actual communication by assigning the account argument to the `savings` field reachable from `this`. The main properties of this example that we will use to describe the analysis are: (a) an escaping *savings* account object ends up being shared between `Husband` and `Wife` threads and (b) a `newAcct` account object is created and used only in the `Wife.run()` method.

3.2.2 Interference Dependence

Definition 24 (interference dependence) Let P be a program, o be an object with field f, and t_1 and t_2 be threads such that $t_1 \neq t_2$. If there exists an execution trace of P such that f is written at trace state s_m by a statement at program point e_m executed by t_1 and read at state s_n by a statement at program point e_n executed by t_2 when s_n occurs after s_m and no write to $o.f$ occurs between s_m and s_n, then n is interference dependent on m. □

Given the above definition of interference dependence in terms of traces of concurrent programs, it is obvious that accurate static calculation of interference dependence is infeasible due to the possible exponential number of interleaved execution traces. Therefore, we need to resort to safe yet cheap approximations such as the one given below.

> *If a field f of object o is being written and read at program points e_m and e_n, respectively, and e_m and e_n occur in different threads t_m and t_n, then e_n is interference dependent on e_m.*

The accuracy of such safe approximations can then be improved at additional cost by checking if m and n may "execute in parallel" according to some static approximation, i.e. as computed by *may-happen-in-parallel* analysis [ACSE99, NAC98].

By referring to the example of Figure 3.4, we now explain (1) how interference dependences are detected in previous presentations of slicing [HCD+99, Zha99], (2) how our modification of Ruf's escape analysis (introduced later in Section 3.3) can be used to safely reduce the number of dependences, and (3) how simple extensions to track aliasing gives even further reduction. We shall

[4] From hereon, "reachable" will signify reachability by following references held in accessible fields.

Interference dependence

D : set of all field/array definition sites.
U : set of all field/array use sites.
I : empty set of interference dependences.
for all u ∈ U **do**
  for all d ∈ D **do**
    if $(\Gamma(u) \equiv \Gamma(d))^*$ **then**
      I = I ∪ {(u,d)}

ReadyDA dependence

W : set of all `wait()` call sites.
N : set of all `notify()/notifyAll()` call sites.
R : set of ready dependences.
for all n ∈ N **do**
  for all w ∈ W **do**
    if $(\Gamma(n) \equiv \Gamma(w))^\dagger$ **then**
      R = R ∪ {(w,n)}

Figure 3.5: Algorithm to calculate the approximate interference dependence and ready dependence based on conditions 3 and 4 given in Section 3.2.3. Γ maps a given definition/use site to a property of the variable being defined or used, and ≡ represents a test for equality between the property.

illustrate how existing approaches and our approach can be seamlessly enabled in the algorithm given in Figure 3.5 by choosing an appropriate property to be associated with variables via Γ and suitably defining ≡.

Type-based approach

A simple approach is to classify field/array read/writes as interfering if e_m represents a read of an expression such as `a1.f`, e_n is a write to `a2.f`, and `a1.f` and `a2.f` have identical signature (i.e. both instances of `f` represent the same field of a class) then e_m is interference dependent on e_n. While this approximation of interference dependence is easy to compute, it includes inference dependences even for objects that are not shared. Hence, it leads to spurious interference dependence dependences such as ones between line 20 and both line 60 and line 65 and between line 60 and line 65.

This approach can be realized by the algorithm in Figure 3.5 by defining Γ as the mapping def/use sites to type of the field being defined/used and ≡ as the type equality test.

Prior to Java 1.2, field resolution was based on the static type of the primary in the field access expression, i.e. the static type of the expression `a` in `a.f`. However, in Java 1.2 and above, field resolution was based on the dynamic type of the primary. Type based approach can be directly applied to programs written in 1.1 or prior versions of Java. In case of programs written in 1.2 or later version of Java, the approach needs to consider all possible field signatures that could fit the field access expression in the def/use sites (i.e. use a static approximation of the possible object types the primary can refer to at runtime and consider the fields with matching signatures in these types).

In the rest of this chapter, for simplicity, we shall assume the input programs were written in Java 1.1.

Leveraging escape analysis

Objects that are accessible outside the scope of their creation context are referred to as *escaping*. If the thread in which objects are created is considered as the creation context, then an object "escapes" the creating thread if it is accessible in a different thread.

In terms of escape information, the definition of interference dependence implies that *only es-*

caping objects induce interference dependences. Escape analyses (such as the one proposed by Ruf [Ruf00]) can determine which objects are accessed by at most one thread, and such objects can be excluded from the calculation of interference dependence. In Figure 3.4, typical escape analyses will mark the local variable `newAcct` in `Wife.run()` as referring only to thread-local/non-escaping objects. This allows the elimination of a spurious dependence between line 20 and line 65 that generated by the type based approach. However, simple escape analysis would mark `checking` and `savings` in `Wife.run()` as escaping since they are both accessed in the constructor for `Wife` which is executed by the `main` thread – *not* the `wife` thread. This means that the dependence between line 20 and line 60 cannot be removed.

In terms of the algorithm, Γ can be defined as mapping from def/use sites to a boolean value. If a site is mapped to *true* then it indicates that the primary of the field/array access expression at that site is shared; *false*, otherwise. $\equiv$ is defined as simple boolean equality test.

Leveraging alias information

Although `checking.balance` and `savings.balance` involve escaping objects (accessed in the main thread and its children threads), `checking` and `savings` will never refer to the same object. In other words, they will never be aliases. Hence, alias information can be used to prune dependences relating `checking.balance` and `saving.balance`. For example, the knowledge that the primaries in the expressions at line 20 and line 60 are not aliases can be used to remove the interference dependence between these lines.

In terms of the algorithm, Γ can be defined as a mapping from a pair of def-use site to a boolean value such that a def-use site pair will be mapped to *true* only if the primaries in the sites may be aliases at runtime.[5] The accuracy of the information returned by Γ is dependent on the points-to/alias analysis but nevertheless it can lead to more accurate dependences.

3.2.3 Ready Dependence

In simple words, a statement m is ready-dependent on a statement n if the execution of m can be infinitely delayed due to the fact that n fails to complete its execution and n is a synchronization related construct. This notion of dependence is relevant when using slicing to generate reduced program models for purposes of temporal property verification(specifically, liveness properties) [HCD+99].

Although we can perceive ready dependence as a form of control dependence and question its discussion in this chapter, we think ready dependence is a form of data-based dependence. The rational being that intra-procedural control dependence stems from the control flow of the program as encoded in the static structure of the program and not the data elements of the program while ready dependence stems from the influence of shared data (e.g. objects on which synchronization operations are performed) on the control flow of the program and not the static structure of the program. Further, we think other interesting concurrency specific dependences in the context of dynamic memory allocation and aliasing will be data-based as opposed being control-based.

We shall use `notify()` to denote both `notify()` and `notifyAll()` methods in `java.lang.Object` and `wait()` to denote all overloaded versions of `wait()` in `java.lang.Object`.

Definition 25 If m and n are program points in a given Java program then m is ready dependent

[5] This will change the signature of Γ in Figure 3.5.

on n if any one of the following is true.

1. m and n occur in the same thread and m is reachable from n and n is the start of a `synchronized` statement.
2. m and n occur in the same thread and m is reachable from n and n is an invocation of `wait()`.
3. m and n occur in different threads and m is the start of a `synchronized` statement on object o and n is the finish of a `synchronized` statement on object o.
4. m and n occur in different threads and n is an invocation of `notify()` on object o and m is an invocation of `wait()` on object o. □

In Java, a synchronized statement is statement block adorned with the keyword `synchronized` along with a variable referring to the lock object. Hence, by *start of a synchronized statement*, we mean the point in the statement block where a monitor corresponding to the lock object is entered by the executing thread. Similarly, by *finish of a synchronized statement*, we meant the point in the statement block where the monitor corresponding to the lock object is exited by the executing thread.

Condition 1 and 2 are uninteresting in the context of this chapter as they can be handled via a simple intra-thread analysis of the control flow graph of the methods. However, condition 3 and 4 provide interesting cases as they relate entities that occur in different threads. From hereon, by "ready dependence" we mean *"ready dependence resulting from conditions 3 or 4 only"*. Techniques proposed to prune interference dependence can also be used to prune ready dependence (e.g. if an object o is not shared then it cannot generate a ready dependence via conditions 3 or 4).

In Figure 3.4, a naive type-based ready dependence analysis will report ready dependence dependences between line 18 and both line 58 and line 62. However, the latter dependence is a spurious one which cannot be eliminated by escape information. However, it can eliminated using alias information computed using *ready entities* as described in Section 3.3.

3.3 Equivalence-based Escape Analysis

Instead of starting from scratch, we adapted an existing escape analysis. We choose to work with Ruf's escape analysis [Ruf00] because of its implementation simplicity, ease of comprehension, scalability, and accuracy in comparison with other approaches (see [Ruf00] for details).

Our adaptation of Ruf's analysis calculates information required to answer the following questions.

- Given a field/array read expression e_m and a field/array write expression e_n, is m interference dependent on n?
- Given a `entermonitor` statement e_m and a `exitmonitor` statement e_n, is e_m ready dependent on e_n under condition 3 in the definition of ready dependence?
- Given a `wait()` call-site e_m and a `notify()` call site e_n, is e_m ready dependent on e_n under condition 4 in the definition of ready dependence?

The analysis proceeds in three phases. The first phase collects information about the system to be used by the later phases. The second phase summarizes information both inter-procedurally and intra-procedurally in a bottom-up fashion. In phase three, the summarized information is disseminated to the methods in a top-down fashion. After presenting the details of the analysis, we summarize how it differs from Ruf's original version.

3.3.1 Alias sets

Each program variable v is associated with an *alias set* – an abstract object that summarizes properties of concrete objects that v may reference at runtime. Contrary to intuition, an alias set is structurally not a set of aliased variables on which membership operation can be performed to determine aliasing. However, conceptually an alias set is a set of aliased variables as an alias set can be constructed by identifying variables associated with the alias set.

An alias set for v is either $\perp$ (indicating that only null reference values are assigned to v at runtime, or that v is of non-reference type) or a tuple of properties as given below.

$$\textit{aliasSet} ::= \perp \mid \langle \textit{fieldMap, escapes, waits, notifies, rdEntities} \rangle.$$

fieldMap associates each field f of an abstract object with an alias set abstracting the concrete objects that may be referenced by f at run-time, i.e. it maps fully quantified field names to alias sets. `$ELT` is a special field used to represent all cells in a dimension of an array.

escapes is a boolean that when *true* indicates that the abstract object is visible in multiple threads.

waits is a boolean that when *true* indicates that the abstract object received a `wait()`.

notifies is a boolean value that when *true* indicates that the abstract object received a `notify()`.

rdEntities is a set of abstract object tokens drawn from a domain E which we call *entities*. An entity represents an equivalent set of alias sets that represent concrete objects that could participate in a wait/notify relationship (leading to a ready dependence). Specifically, if the intersection of the ready entities for some alias sets a_1 bound to variable v_1 and a_2 bound to v_2 is non-empty, then invocations of `wait`/`notify` on v_1 and v_2 can lead to a ready dependence.

The significance of *waits*, *notifies*, and *rdEntities* will be explained in Section 3.6.

Upon creation of an alias set, all boolean elements of the alias set are set to *false*, *rdEntities* is set to $\emptyset$, and *fieldMap* is empty. Alias sets of fields are created on demand. The following operations operate on alias sets.

clone operation creates a new alias set isomorphic to input alias set.

markAsEscapes operation sets the *escapes* element of all alias sets reachable (inclusive) from the input alias set to *true*.

unify (θ) operation merges the information represented by elements of the given pair of alias sets (except for *rdEntities* as explained below). The unification of the boolean and map values is defined as the join under the boolean lattice (with *true* as the top element) and function lattice, respectively. The domain of the resulting *fieldMap*s is the union of the domains of the maps being unified. The alias sets of the field names occurring in the domains of both the maps are also (recursively) unified.

$\theta(\textbf{arg1}, \textbf{arg2})$.**escapes**	**arg1.waits**	**arg2.notifies**	$\theta(\textbf{arg1}, \textbf{arg2})$.**rdEntities**
true	*true*	*true*	$arg1.rdEntities \cup arg2.rdEntities \cup \{newEntity\}$
false	-	-	$arg1.rdEntities \cup arg2.rdEntities$
-	*false*	-	$arg1.rdEntities \cup arg2.rdEntities$
-	-	*false*	$arg1.rdEntities \cup arg2.rdEntities$

Table 3.1: Rules to unify *rdEntities*.

The *rdEntities* of the alias sets being unified is modified only when the *waits* element of one alias set is *true* and the *notifies* element of the other alias set is *true*. In such cases, the *rdEntities* set of the unification will be the union of the *rdEntities* sets of the argument alias sets and a singleton set containing a fresh distinct entity (as given by Table 3.1). In all other cases, the unification of *rdEntities* set is defined as the set union operation.

3.3.2 Alias Context

An *alias context* is an aggregate data structure that summarizes aliasing based information at method interface. An alias context is a tuple of alias sets corresponding to `this` (in case of instance methods), method arguments $(a_1, a_2, a_3, \ldots)$, return value (r), and the exceptions thrown by the method (e).

$$aliasContext ::= \langle this, \langle a_1, a_2, a_3, \ldots \rangle, r, e \rangle$$

Like alias sets, alias contexts support the following operations.

clone operation creates a new alias context isomorphic to input alias context.

markAsEscapes operation sets the *escapes* element of all alias alias sets reachable (inclusive) from the input alias context to *true*.

unify operation is a point-wise extension of alias set unification to alias contexts tuples. In other words, this operation merely unifies corresponding (according to structural isomorphism) alias sets (as explained earlier) in the given pair of alias contexts.

For the sake of simplicity, we use the the term *method context* to indicate the alias context of a method at its entry point and the term *site context* to indicate the alias context at a call-site.

Both alias set and alias context can be implemented using union-find data structures [VSD86] (as done in Indus) for the purpose of fast unification.

3.3.3 Algorithm

Phase One

In the first phase, the analysis constructs a call graph that presents a unified view of all the threads in the system. Specifically, besides the syntactically and semantically explicit caller-callee edges,

there is path in the call graph from the parent thread to the child thread via the call site that creates the child thread by invoking `java.lang.Thread.start()`.[6] A reasonably accurate approach to build such a call graph would be to rely to use the result from a points-to analysis such as object flow analysis [Ran02] or rapid-type analysis [BS96].

Phase Two

This phase operates in inter-procedural mode by processing each method in each strongly connected component (SCC) in a bottom-up fashion on the call graph. The processing of the methods includes the creation of the method context and performing flow-insensitive intra-procedural analysis on the method as described below. The effect of this phase is to accumulate information about object accesses in a method and its callees, calculate new information (such as ready entities) based on the accumulated information, and continue this process by bubbling up this information along the edges of the call-graph of threads and the program.

Intra-procedural analysis The rules for processing the statements of a method during intra-procedural analysis are given in Figure 3.6.

Before describing the rules, a brief description of the mappings is given below.

AS returns the alias set associated with the given variable.

MC returns the method (alias) context associated with the given method.

CALLEES returns the callees (invoked methods) that will be invoked at the call site involving the given receiver variable in the given method.

SCC returns the methods in the strongly connected component (SCC) of the call graph that contains the given method.

MULTI_EXEC returns *true* if the given call site may be executed multiple times via looping or execution of the enclosing method.

The rules ensure that the aliasing of data inside a method is represented appropriately in its method context. During processing, if an alias set for a variable/field does not exist, a new instance is created.

The interesting rule is the one for method invocation. At a call-site, if the caller and the callee exist in the different strongly connected components in the call graph then the method context of the callee is cloned and the clone is unified with the site context to achieve context sensitivity. When both caller and callee occur in the same SCC, the method context of the callee is unified with the site context to achieve an effect similar to simple fixed-point iteration. Using the method context instead of its clone in the unification is an optimization to avoid repetitive unification along the paths of the SCC until a fixed point is reached. Depending on the callee methods, *waits and notifies* elements of the alias set associated with the receiver variable are updated. Similarly, the *escapes* element of the alias sets reachable from the alias sets in the method context are also updated.

If the method being invoked is the `start()` method, then all the references being accessed in the child thread may escape as they may be reachable from the reference to the thread object. Hence,

[6] There are no separate call graphs for each thread in the system.

Domain

$v \in V$	set of variables
$f \in F$	set of fields
$m, p \in M$	set of methods
$a, r, e \in A$	set of alias sets
$mc, sc \in C$	set of alias contexts
$s \in CS$	set of call sites

Mappings

$AS : V \rightarrow A$	alias set lookup
$MC : M \rightarrow C$	method context lookup
$CALLEES : M \times V \rightarrow \wp(M)$	callee lookup
$SCC : M \times \wp(M)$	SCC lookup
$MULTI\_EXEC : CS \rightarrow \{true, false\}$	multiply executed call-site lookup

Rules

statement	action
$v_1 = (t)v_2$	$\theta(AS(v_1), AS(v_2))$
$v_1 = v_2.f$ $v_2.f = v_1$	$\theta(AS(v_1), AS(v_2).\mathit{fieldMap}(f))$
$v_1[] = v_2$ $v = v_1[]$	$\theta(AS(v_1).\mathit{fieldMap}(\$ELT), AS(v_2))$
return v	$\theta(AS(v), AS(r))$
throw t	$\theta(AS(t), AS(e))$

$v_r = v_t.m(a_1, \ldots a_n)$	*let* $sc \leftarrow \langle AS(v_t), \langle AS(a_1), \ldots AS(a_n)\rangle, AS(v_r), e\rangle$
	$\forall p \in CALLEES(m, v_t).$
	let $mc \leftarrow MC(p)$
	if $p =$ *java.lang.Object.wait* then
	$AS(v_t).waits \leftarrow true$
	if $p =$ *java.lang.Object.notify* then
	$AS(v_t).notifies \leftarrow true$
	if $p =$ *java.lang.Thread.start* then
	$markAsEscapes(mc)$
	if $p \in SCC(m)$ then
	$\theta(sc, mc)$
	else
	$\theta(sc, clone(mc))$
	if $MULTI\_EXEC(v_t.m(a_1, \ldots a_n)) \wedge$
	$m =$ *java.lang.Thread.start* then
	$\theta(sc, sc)$

Figure 3.6: Domains, mappings, and rules used in intra-procedural analysis. In these rules, m represents the method in which the rules are being applied, e represents the alias set corresponding to the exceptions thrown in m, and r represents the alias set corresponding to the return value of m.

the *markAsEscapes* operation is used to mark all alias sets reachable from the method context as escaping. However, this is done *before* unification to affect the unification of elements sensitive to multiple threads.

If the call-site invokes `start()` and is determined to be executed multiple times, the site context is unified with itself via *unify* to account for the effect of multiple executions as described in Section 3.5.1.

Handling Static fields Unlike instance fields, static field (global variables) are associated with classes (types) and not with objects. Hence, the information propagation as described above will result in incorrect information pertaining global variables. This is addressed by using canonical representative alias sets for global variables, i.e. every occurrence of a global variable is associated with the same alias set. In terms of realization, a new boolean element *global* is injected into alias sets. The containing alias sets are marked as global by setting the *global* elements to *true* when the containing alias sets are associated with a global variable. As data reachable from a global variable can be accessed without traversing an object exposed via interfacial elements such as parameter variables, the alias sets reachable from global alias sets[7] are also marked as global. As for the unification of the *global* element, it is defined as the join operation under the boolean lattice (with *true* as the top element). The cloning of global alias set is defined as the identity operation, i.e. the input global alias set (not a copy) is returned as the clone. This simulates the effect of instance independent access to static fields/global variables as the same alias set is accessed across different methods.

It is possible that a static field may be accessed in a method but the alias set associated with that static field may not reachable from the alias context of the method. This can occur when the references reachable from static field are only assigned to local variables to the method. In such cases, the rules for phase one will fail to record such access as contributing to inter-thread data sharing. To address this issue, for each static field, the representative alias set is marked as escaping via *markAsEscapes* and unified with itself at the end of phase two. The marking and unification will force any marking and unification sensitive processing that was skipped as a result of these alias sets not being exposed at call-sites.

In Ruf's analysis, alias sets corresponding to static fields are pessimistically marked as escaping as they are accessible from all threads. However, doing so may result in inaccuracies, e.g. some static fields may not be accessed in multiple threads. Hence, in our analysis, we rely on self unification to trigger the calculation of information according to the rules and improve the accuracy of the analysis.

Phase Three

In this phase, the call graph is traversed in a top-down fashion by disseminating the calculated information from the caller methods/creator threads to callee methods/created threads.

Call-sites are the only points of processing in this phase. At each call-site encountered in the top-down traversal of the call graph, each alias set in the site context is visited recursively. On visiting an alias set of a site context, the *escapes* element of the alias set is joined with the *escapes* element of the alias set's counterpart in the method context. As for *rdEntities*, the sets are not union-ed (which would destroy context sensitivity), but rather elements from the *rdEntities* set in the site context alias sets are injected into the *rdEntities* of method context alias sets.

[7] We shall refer to alias sets with *global* element set to *true* as *global alias sets*.

3.3.4 Complexity

The algorithm visits each node in the call graph twice and at each node it visits the statements of the methods once. The worst case processing time for unification is dictated by the recursive nature of the data structures used in the program, in particular when propagating information across method boundaries. Hence, the worst case time complexity for the analysis is in the order of $O(N+S)$ where N is the number of nodes in the call graph and S is the total number of statements in the system if time complexity of object construction and alias set processing is considered as a constant.

A recursive data structure with m fields being passed along n call chains of length l can lead to alias sets whose accumulated size is in the order of $O(n * m^l)$. However, in reality, large recursive data structures are also processed and constructed recursively rather than a single big chunk. Hence, reaching worst case space complexity would be a rarity.

3.3.5 Example

In this section, we shall walk through the application of the analysis to the program in Figure 3.4.

Phase One

The analysis calculates the call graph (as depicted in Figure 3.7) for the program. The call graph has two important aspects not discussed earlier. The first aspect is that the invocations/calls inside non-Java/native methods are not captured by the call graph. This approach to call graph construction is common, but it can lead to inaccurate results. We assume that the number of method calls from native methods that can result in call graph edges is low. If this is not the case, the call graph should be augmented to account for such edges. The second aspect is that one such augmentation is necessary and should be performed in case of `java.lang.Thread.start()` method. Despite `start()` being a native method, the call graph is augmented with outgoing edges from `start()` to invoked methods.

Phase Two

Given the call graph of the system, the bottom-up processing of the methods will start with the class initializers (written as `<clinit>`) and `Acct.Acct()`. The methods `wait()` and `notify()` are not processed as they are native methods and lack Java method bodies; however, method contexts are created for these methods for the purpose of unification at call sites.

On processing `Acct.Acct()`, a method context $\rho_1 = \langle \alpha_0, \langle\rangle, \alpha_0, \perp \rangle$ is created where and $\alpha_0 = \langle \langle\rangle, \mathit{false}, \mathit{false}, \mathit{false}, \emptyset \rangle$ is a newly created alias set representing the `this` variable in `Acct.Acct()`. We use the term *newly created alias set* to indicate an alias set unaltered after creation. α_0 occurs in the position of the return value alias set to capture the information that the new object created by the invocation of the constructor is also returned.

The method context at the start of the processing of `Wife.Wife(Acct)` will be $\rho_2 = \langle \alpha_1, \langle \alpha_2 \rangle, \alpha_1, \perp \rangle$ where α_1 and α_2 are newly created alias sets corresponding to `this` variable and the argument `Acct` object. While processing the method, new alias sets α_3 and α_4 corresponding to `savings` and `checking` fields, respectively, are created. The assignment of the `act` to `savings` leads to the unification of α_2 and α_3 ($\alpha_5 = \theta(\alpha_2, \alpha_3)$) and the result of this unification will be identical to α_2

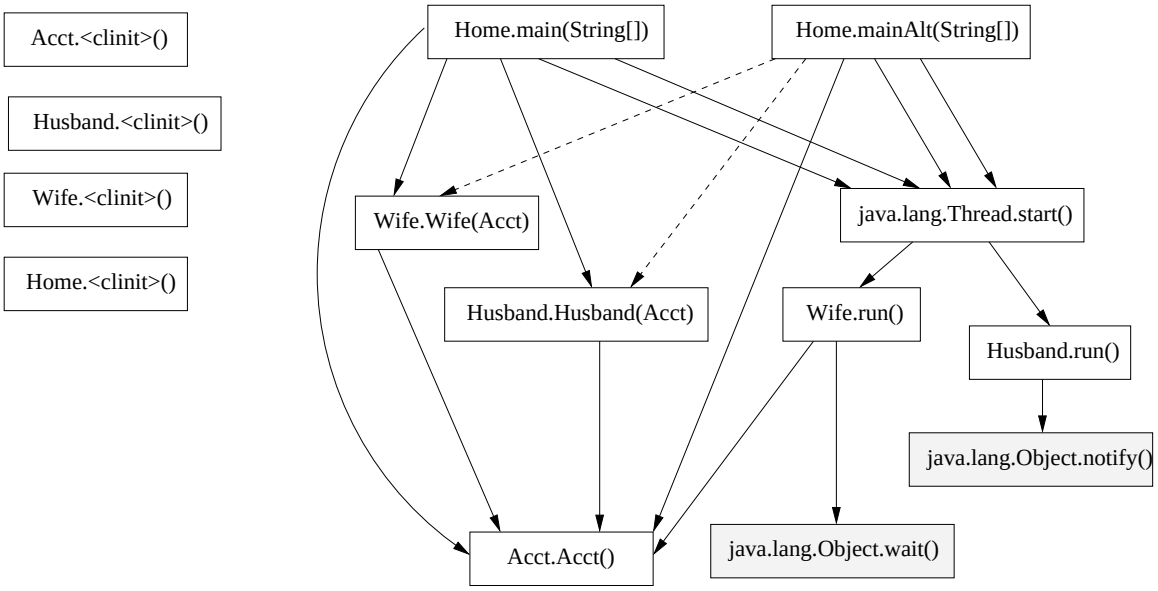

Figure 3.7: The call graph for the program in Figure 3.4. The solid lines represent an invocation via a particular call site in the direction of the arrow. The dashed lines represent invocations that may be executed multiple times in the direction of the arrow. Boxes represent the named methods. Shaded boxes represent named native methods.

and α_3. After processing the method, $\alpha_1 = \langle\{savings \rightarrow \alpha_5, checking \rightarrow \alpha_4\}, false, false, false, \emptyset\rangle$ and $\rho_2 = \langle\alpha_1, \langle\alpha_5\rangle, \alpha_1, \perp\rangle$. A similar method context $\rho_3 = \langle\alpha_6, \langle\alpha_7\rangle, \alpha_6, \perp\rangle$ is generated for `Husband.Husband(Acct)` where $\alpha_6 = \langle\{savings \rightarrow \alpha_7, checking \rightarrow \alpha_8\}, false, false, false, \emptyset\rangle$. This operation is illustrated in Figure 3.8.

Upon processing `Wife.run()`, its method context will be $\rho_4 = \langle\alpha_9, \langle\rangle, \perp, \perp\rangle$ where $\alpha_9 = \langle false, \{savings \rightarrow \alpha_{10}, checking \rightarrow \alpha_{11}\}, false, false, false, \emptyset\rangle$ and $\alpha_{10/11} = \langle\langle\rangle, false, true, false, \emptyset\rangle$. A similar method context ρ_5 is created for `Husband.run()` except that the *notifies* element in the alias sets of `savings` and `checking` fields is set to *true* and *false*, respectively.

Although the local variable *newAcct* in `Wife.run()` will be associated with an alias set, the alias set is not visible from the method context as the data in the local variable is never exposed beyond the method via assignment to fields in the arguments or the receiver or those reachable from static fields.

Next, the processing of the line 26 in `Home.main(String[])` creates a new alias set, α_{12}, associated with the local variable `savings`. As the line also contains a method invocation, `Acct.Acct()`, a site context ρ_6 is created and unified with the alias context ρ'_1, a clone of ρ_1,[8] resulting in $\langle\alpha'_0, \langle\rangle, \alpha'_0, \perp\rangle$.

While processing the constructor `Wife.Wife(Acct)` invocation at line 28, a site context $\rho_5 = \langle\alpha_{13}, \langle\alpha'_0\rangle, \alpha_{14}, \perp\rangle$ is created. Upon unifying it with ρ'_2, a clone of ρ_2, the result will be $\langle\alpha'_1, \langle\alpha'_0\rangle, \alpha'_1, \perp\rangle$. As α'_1 is used in positions corresponding to α_{13} and α_{14}, the context unification results in the unification of α_{13} and α_{14} via α'_1. Further, $\alpha'_1 = \langle\{savings \rightarrow \alpha'_0, checkings \rightarrow \alpha'_4\}, false, false, false, \emptyset\rangle$. The processing of line 29 will result in a site context $\rho_6 = \langle\alpha'_6, \langle\alpha'_0\rangle, \alpha'_6, \perp\rangle$ where $\alpha'_6 = \langle\{savings \rightarrow$

[8] We shall use α'/ρ' to denote the clone of α/ρ.

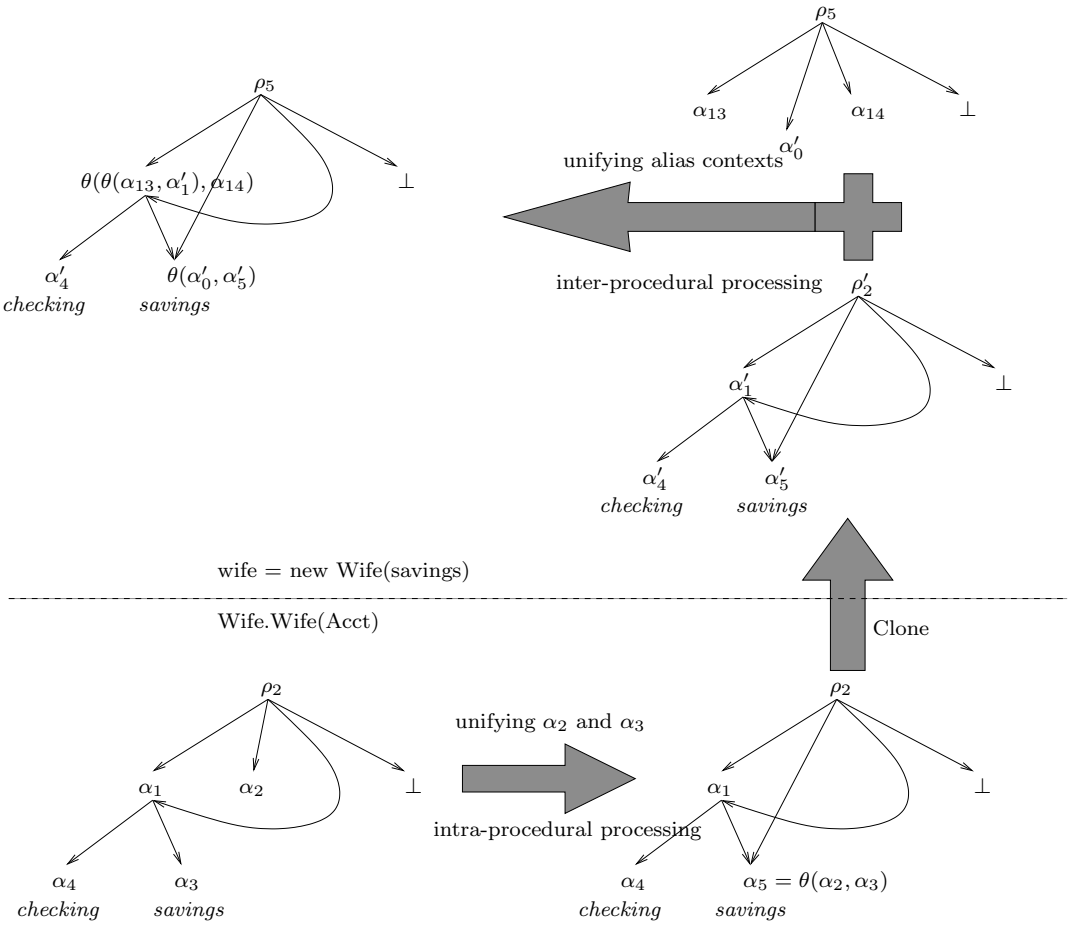

Figure 3.8: Graphical illustration of the alias set and alias context unification during intra- and inter-procedural processing in the second phase of the analysis. The dotted line denotes the boundary of the method.

$\alpha'_0, checkings \rightarrow \alpha'_8\}, false, false, false, \emptyset\rangle$.

The processing of line 30 will start with the creation of the site context $\rho_7 = \langle \alpha'_1, \langle \alpha'_0 \rangle, \alpha'_1, \perp \rangle$ followed by its unification with the clone of the method context of `java.lang.Thread.start()` method resulting in $\langle \alpha'_1, \langle \alpha'_0 \rangle, \alpha'_1, \perp \rangle$ where $\alpha'_0 = \langle \langle \rangle, \underline{true}, \underline{true}, false, \emptyset \rangle$. The important results of this unification are the propagation of the information about the waiting on the object referred to by `Wife.savings` field into `main(String[])` and the marking of the object referred to by the same field as escaping.

Similar processing occurs while handling line 31. However, in this case, the *notifies* element of the alias set corresponding to the `savings` field in the clone of the participating method context is set to *true*. Hence, according to the rules in Table 3.1, a new entity is injected into the *rdEntities* set of α'_0. Hence, $\alpha'_0 = \langle \langle \rangle, true, true, \underline{true}, \{newEntity1\} \rangle$.

Phase Three

In the last phase, information is propagated in the top-down direction. Specifically, *escapes* and *rdEntities* of alias sets $\alpha_0, \alpha_5, \alpha_7, \alpha_{10}$, and corresponding counterparts in the method context of `Wife.run()` and `Wife.Wife(Acct)` are updated with the values from α'_0.

3.3.6 In Comparison with Ruf's analysis

We now summarize how our escape analysis differs from the original version proposed by Ruf.

- Ruf's analysis is geared towards the removal of unnecessary synchronization operations. Hence, Ruf's alias sets have elements *synchronized* and *syncThreads* to capture information indicating whether or not objects represented by the alias set are used in synchronization and the number of threads that synchronize on the objects, respectively. However, the alias sets in our analysis have other fields geared towards general escape information and interference and ready dependence instead of the above mentioned fields.
- The unification rules in Ruf's analysis mark an alias set as synchronized only at synchronization expressions. However, in our analysis, the decision to mark the alias set as escaping is deferred to the unification operation.
- Our analysis provides more accurate information as required by interference/ready dependence (explained in sections 3.6.2 and 3.6.3) as it maintains ready/share entities that account for dependence-specific sharing while being calling contexts sensitive.
- Thread creating call sites are handled conservatively in Ruf's analysis by marking every alias set reachable from such a site context as escaping. In contrast, our analysis employs an optimization to improve the accuracy (see Section 3.5.1).
- As mentioned in Section 3.3.3, our analysis unifies these alias sets with themselves to capture the effect of multi-thread access and computing data sharing information based on the rules instead of pessimistically marking these alias sets and reachable alias sets as escaping as done in Ruf's analysis.

3.4 Extensions

In this section, we shall describe three extensions to the analysis to calculate aliasing, lock coupling, and side-effect information. These extensions are not directly related to program dependence calculation.

3.4.1 Aliasing

A simple extension to our analysis can calculate *if two reference variables are aliases*. The extension is required for two reasons.

1. The algorithm does not unify alias sets across method boundaries, neither in phase two nor in phase three. Hence, a trivial query such as "do the concerned reference variables have the same corresponding representative alias set?" will fail except in cases of mutually recursive methods.
2. The data structures used in the analysis do not store any information pertaining to aliasing.

Based on the above reasons, we extend the data structures and the operations to capture aliasing information and extend the algorithm to calculate the aliasing information.

The extension to the data structure is the addition of *intraThreadEntities* set to the alias set. This elements of this set will be used to check if two alias sets indeed correspond to aliasing variables.

As for the *unify* operation, the unification of *intraThreadEntities* sets is defined as the set union operation followed by the injection of a new distinct element into the union. Hence, new entities injected at unification are propagated to the call sites via the unify operation. As for the *clone* operation, the clones of non-global alias sets will have empty *intraThreadEntities* sets.

Given a vertical alias, the extension to *unify* captures vertical aliasing information and exposes it in the context of caller method, but not in the context of the callee methods. It also captures horizontal aliasing information in the caller, but it does not expose this information to either of the callee methods in which the aliasing occurs. The missing exposure is achieved in phase three by injecting the elements of *intraThreadEntities* sets of the site context alias sets into the *intraThreadEntities* sets of the corresponding method context alias sets.

Uncontrolled unification at thread creation sites can pollute the intra-thread aliasing information with inter-thread aliasing information. To prevent such pollution, the *intraThreadEntities* sets in clone of the method context (or in the method context) alias sets are emptied before the unification at thread creation sites. Unlike *rdEntities*, in phase three, the analysis does not inject the elements in *intraThreadEntities* sets in site context alias sets into the *intraThreadEntities* sets of the corresponding method context alias sets at thread creation sites. Once again, this is to prevent the pollution of intra-thread aliasing information with inter-thread aliasing information.

Interestingly, this approach of calculating aliasing information provides accurate results that comparable to that provided by relatively expensive iterative flow analysis based approaches [Ran02, LH03] provided the call graph used for the analysis is accurate.

$\theta(\mathbf{arg1}, \mathbf{arg2})$.**escapes**	**arg1.locked**	**arg2.locked**	$\theta(\mathbf{arg1}, \mathbf{arg2})$.**lkEntities**
true	*true*	*true*	$arg1.lkEntities \cup arg2.lkEntities \cup \{newEntity\}$
false	-	-	$arg1.lkEntities \cup arg2.lkEntities$
-	*false*	-	$arg1.lkEntities \cup arg2.lkEntities$
-	-	*false*	$arg1.lkEntities \cup arg2.lkEntities$

Table 3.2: Rules to unify *lkEntities*.

3.4.2 Lock Coupling

Ruf's analysis targeted the detection of synchronized blocks involving uncontested locks. As our analysis is an extension of Ruf's analysis, similar information can be calculated along with the information required to answer the *lock-coupling* question – *do the given two synchronized blocks contend for the same lock?*

As in the previous extension, we extend the data structures and the operations to capture locking information and extend the algorithm to leverage and propagate the locking information and, consequently, calculate lock-coupling information.

Following suite from the previous extension, we extend the alias set with a new boolean element named *locked*. When *locked* element of an alias set is *true*, the object referred to by the corresponding variables is involved as a *lock object* in a synchronization block. Hence, during phase two, the *locked* element of the alias set corresponding to variable providing the lock object in a synchronization block is set to *true*.

Another extension to alias set is the addition of a new set element named *lkEntities*. The elements of *lkEntities* set are used alike *intraThreadEntities* to represent lock coupling. However, the unification of *lkEntities* is defined by the rules given in Table 3.2.

As for the processing of *lkEntities* in the algorithm, the proposed unification rules are used in phase two and the entities are propagated as in case of *intraThreadEntities* in phase three.

Given these modifications and two variables, the lock coupling question can be answered by checking if the *lkEntities* set in corresponding alias sets have a common element.

3.4.3 Side-Effect Analysis

Our analysis maintains method contexts that summarize access information about various data reachable via the arguments to and the receiver of the method. Further, as the access information is accumulatively propagated bottom-up in the second phase, the method contexts summarize the access information about data reachable via the arguments and receiver in the corresponding method and the methods called, directly or indirectly, from the corresponding method. Hence, the algorithm and the data structure provide an ideal setup to calculate *side-effect information – detect if data reachable from the arguments or the receiver is written by a method.*

Given method m and an argument/receiver a/r to the method, specific questions that can be answered by side-effect information are:

1. Does m modify the state of a/r, i.e. write to any field reachable from a/r?

2. Does m depend on the state of a/r, i.e. read any fields reachable from a/r?
3. Does m modify the global state of the system, i.e. write to any static fields (global variables)?
4. Does m depend on the global state of the system, i.e. read any static fields (global variables)?
5. Does m write to the field at the end of a given access path that is a chain of fields and that is rooted at a/r?
6. Does m read the field at the end of a given access path that is a chain of fields and that is rooted at a/r?

The answers to the first four of the above questions can be used to optimize method invocations without exploring the programs points that may be executed as a result of the invocation. For example, if the analysis indicates that the method does not write to any static fields or to any fields reachable from any of the arguments/receiver to the method, then the method can be considered as *pure – a code block that does not modify the state of the system that existed immediately prior to the execution of the code block*. The last two questions can be used in driver and stub generation in the context of program testing and program verification[9].

To calculate the required information, the analysis needs to record the fields read and written via every variable in a method and propagate it to every caller of that method. This can be achieved by extending the alias sets with new entity sets *readFields* and *writeFields*.

As for processing, upon reading or writing a field, the signature of the field is added to the *readFields* and *writeFields* sets, respectively, of the alias set corresponding to primary of the processed field access expression. The unification of these sets is defined as set union. Also, in phase three, the elements from these sets in the site context alias sets are not propagated into the corresponding sets in the corresponding method context alias sets. This prevents the read-write information of the caller method from polluting the read-write information of the callee method.

3.4.4 Generalization

Given the number of extensions that can be layered on top of the vanilla version of the analysis, we believe other interesting analysis can be layered as well. For this purpose, we generalize the approach to layer extensions on top of the analysis.

For each dimension of the information, the extension should inject a scalar element (e.g. *escapes* in case of uni-dimensional information, *read* and *write* in case of multi-dimensional information) into the alias set along with appropriate rules for modifying the element during intra- and inter-procedural processing and during alias set/context unification. Depending on the direction of the information (e.g. vertical in case of side-effect analysis, horizontal in case of escape analysis), the extension may choose to restrict or allow the propagation of the information in the third phase. In cases such as read-write coupling, the extension should introduce an element for each of the two dimension of the information and appropriately affect the processing rules.

To calculate inter-procedural call-path sensitive information, the extension should introduce an entity set (e.g. *rdEntities*) and provide rules that govern how and when the entity sets should be modified either independently or based on the elements introduced to track associated dimensions of the information. The extension can also provide unification rules and propagation constraints

[9] http://beg.projects.cis.ksu.edu

Information	Dim	*Elements*	*Entity Sets*	Call Path Sensitive
Escape	1	*escapse*	-	N
Aliasing	1	-	*intraThreadEntities*	Y
Interference	2	*read, written*	*rwEntities*	Y
Ready	2	*waits, notifies*	*rdEntities*	Y
Locking Coupling	1	*locked*	*lkEntities*	Y
Side Effect	2	*readFields, writeFields*	-	N
Write Coupling	2	*readFields, writeFields*	*signatureOfWWFields*	Y
Field-based Sharing	2	*readFields, writeFields*	*signatureOfRWFields*	Y

Table 3.3: Summary of various realized refinements of the generalization described in Section 3.4.4.

to control the influence of intra- and inter-procedural processing and information direction on the calculated information, respectively.

Various refinements of this generalization realized in this effort is summarized in Table 3.3. Some of the mentioned realizations are explained later in the chapter.

3.5 Optimizations

In this section, we describe three optimizations to the analysis: two to improve accuracy and one to improve performance.

3.5.1 Multiple Executions of Thread Creation Sites

Thread creating call-sites that are executed multiple times pose obstacles to the accuracy of static analyses such as escape analysis.

For example, if `Home.mainAlt()` was considered as the entry point to the system, then the abstract thread corresponding to the thread allocation site in the loop at lines 41 and 43 in Figure 3.4 represents more than one runtime thread. If unification at such thread creation call sites are processed only once, then the analysis fails to correctly calculate information based on the sharing of objects between the threads created at such call sites. Ruf's analysis handles this situation by marking all alias sets reachable from such site contexts as escaping. Although correct, this approach is conservative – *what if the threads created at such thread creating call sites do not share data?*

To address this pessimism, consider unrolling the enclosing loop once. Upon analyzing the resulting program using the rules given earlier, each alias set reachable from the site context will be unified with itself if the variable names are reused in the loop unrolling. Hence, in our analysis, we rely on this observation in Phase 2 to unify the site context with itself via *unify* operation when the method invoked at the call-site is `java.lang.Thread.start()` and the call-site is executed multiple times (note that due to the definition of unification on escape and ready information, unifying an alias set with itself does not necessarily correspond to the identity function). This captures the effect of the loop on the flow of information.

This optimization is already encoded in the rules described in Figure 3.6.

Thread Allocation Sites Another similar situation arises due to multiple execution of thread allocation sites. Although similar, the allocated threads need to be created via thread creating call sites – either through multiple executions of thread creation sites or execution of separate thread creation sites involving the allocated Thread objects. Previously described optimization handles the former case while the rules of the analysis handles the latter case. Hence, thread allocation sites that are executed multiple times do not require special handling.

3.5.2 Static Field Access

Although our analysis takes extra steps to improve the accuracy of information involving static fields, the resulting accuracy can be less than the expected accuracy: *self unification of global alias sets at the end of phase two calculates information that is independent of the different threads and methods in which the associated global variables are accessed.*

This oversight is addressed by capturing and exposing the access to static fields via the alias contexts in a thread sensitive manner. For this purpose, the alias contexts are extended with an alias set g that corresponds to an abstract variable that is a constant and refers to an abstract object that contains every static field in the program as its member. We shall overload the term global alias set and use it to refer to g.

$$aliasContext ::= \langle this, \langle a_1, a_2, a_3, \ldots \rangle, r, e, g \rangle$$

The analysis will process global alias sets as any other member alias sets of alias contexts with the following changes:

- The rules given in Figure 3.9 are used process static fields during intra-procedural processing.
- Unlike the handling of static fields as described in Section 3.3.3, none of the alias sets are marked as *global* in phase two. As a consequence, clones of alias sets reachable from global alias sets are created (as opposed to using canonical representatives) when containing alias contexts are cloned.[10]
- At the end of the intra-procedural processing of a method m in phase two, for every site context occurring in m, the contained global alias sets are unified with the global alias set with the method context of m. This ensures the propagation of static field access information to the caller.
- At the end of phase two, the global alias sets are processed on a per thread basis. Specifically, methods that serve as the entry point for threads in the given program are identified (i.e. class initializers, implementations of `java.lang.Runnable.run()`, and methods with the signature `public static void main(String[])`), global alias sets in the alias contexts of these methods are marked as *escaping* via *markAsEscapes*[11], and unified with each other. Unlike in the

[10] To facilitate the calculation of side-effect information, *global* elements of alias sets reachable from global alias sets can be set to *true* before processing the methods in phase three.

[11] The alias sets exposed via the method contexts of class initializers are marked as escaping because 1) they represent escaping entities and 2) they would not have been marked as escaping at the (implicit) thread creation sites corresponding class initialization threads.

Domain

$f, f_g \in F$	set of fields
$a, r, e, g \in A$	set of alias sets

Rules

statements	action
$v_1 = f_g$	$\theta(AS(v_1), g.fieldMap(f_g))$
$f_g = v_1$	

Figure 3.9: Additions/Changes to the domains and rules presented in Figure 3.6 used in intra-procedural phase of the analysis.f_g denotes a static field.

approach described in Section 3.3.3, global alias sets are not unified with themselves as the exposure of global alias sets in the alias context would have subjected them to marking and unification sensitive processing.

- As the canonical representative is not used for global alias sets, the information pertaining to static fields is propagated in phase three. Specifically, the value of *global* element is propagated from the caller-side alias sets to the callee-side alias sets via the join operation. The other parts of the alias sets are handled as in the unoptimized case.

To preserve space, the reference to global alias sets can be dropped from the alias context after the corresponding method has been processed in phase three as the information pertaining to static field access is captured by the alias sets corresponding to the local variables.

In summary, the optimization partitions the call graph into overlapping maximal sub-graphs rooted at nodes corresponding to the each non-invoked methods (no incoming edges). The data structures are modified such that escape information is maintained local to each of these sub-graphs in phase two. At the end of phase two, the global alias sets exposed at class initializers are marked as escaping and then unified with a common alias set that summarizes escape information for all static fields in the program.

3.5.3 Type Filtering

In the Java class library, `String.valueOf(o)` is commonly used to construct string constants like in expressions `"Error : param1=" + p1`. Internally, `String.valueOf(Object)` returns the string constant `"null"` if the input argument is `null`; otherwise, it returns the result of invoking `toString()` on the input argument. Hence, many `toString()` methods will occur in the same strongly connected component (SCC) in the call graph as most `toString()` implementation use the string concatenation operation to construct the string representation of the receiver object. Hence, many fields belonging to different classes are confounded into a large alias set corresponding to the input argument of `toString()`. Due to non-cloning of method contexts when caller and callee belong to the same SCC, this large alias set will be cloned at the edges of the containing SCCs. Further, if such an SCC occurs lower/earlier in the bottom-up topological ordered list of SCCs in the call graph, then such large computation and resource intensive cloning can occur repetitively. This can cause major time and space bottleneck.

A similar situation can also occur in applications that use AWT/Swing event framework. Specifically, the situation stems from the use of `java.lang.Object` as a "general" type for the event source

in the constructor of `java.awt.Event/java.awt.AWTEvent` and the deep class hierarchy spanning AWT and Swing classes.

These bottlenecks can be addressed by relying on the covariance property of the Java type system – *given the type τ of a parameter position of a method, only instances of subtypes of τ can occur as arguments in that position.*. In other words, the types of the arguments limit the features of the argument objects that are accessed in the methods. Hence, while propagating the information from the callees to the callers in phase two, *only information pertaining to fields determined to be accessible by the declared static type of the concerned parameter position is propagated.* Accessible fields of a static type include the fields declared in the super-types and the sub-types of the static type. The former inclusion follows from the support for inheritance in Java while the latter inclusion follows from the support for downcasting (i.e. cast from a super-type to a sub-type).

The specific change to the algorithm is that, while propagating the values from callee to caller at a call site, the field map of the alias sets corresponding to the parameters (including the receiver) to the callee are trimmed/filtered based on the accessible fields of the static type of the corresponding parameters. This notion of reducing the size of the alias set is known as *type filtering*.

In cases where the callee and the caller occur in the same SCC in the call graph, it is likely that parameters/variables of sibling types may be associated with the same alias sets due to flow through a variable of a common super-type and direct unification (without cloning) of alias contexts. In such cases, type filtering can incorrectly eliminate valid accessible fields belonging to the subtypes and retain only the fields declared in common super-types. To avoid such issues, type filtering is only applied (to the clone of method contexts) when the callee and caller occur in different SCCs in the call graph.

Although rare, this issue can also occur in cases where the caller and the callee occur in different strongly connected components. Hence, this optimization should be considered to be unsound and applied with care.

In most general situations, we strongly believe that the optimization will be sound due to type correctness of Java programs, single implementation inheritance in Java, and cloning of alias contexts before type filtering. The cloning requirement implies that it more likely for this optimization to yield unsound results when used with the static field access handling approach described in Section 3.3.3 as opposed to with the approach described in Section 3.5.2.

As the optimization does improve the performance by manifold on medium- to large-scale programs (as demonstrated in Section 3.7.8), we propose that this optimization be applied only when the proposed optimizations fail to scale.

3.6 Applications

We now describe how the information computed by our escape analysis can be used to calculate interference and ready dependence.

3.6.1 Using Escape Information

As each variable and field of reference type in the program is associated with an alias set, each such variable and field is also associated with escape information. Hence, as mentioned in Section 3.2.2,

during the calculation of interference dependence, the dependence analysis can check if the primaries of the expressions involved in the candidate dependence refer to escaping objects by checking if the *escapes* element of corresponding alias sets is set to *true*. If so, the candidate dependence is retained; otherwise, it is discarded. By candidate dependence, we mean the pair of program entities that are considered by the dependence analysis as leading to dependence (e.g. by starting from type-based information).

Similarly, for ready dependence, the same check is performed on the receiver variables involved in the `wait()` and `notify()` invocations involved in the candidate dependence.

3.6.2 Accurate Ready Dependence via Ready Entities

As mentioned in Section 3.2.2, escaping information can be used to detect spurious dependences involving non-escaping objects; however, there are cases where aliasing information can be used to detect spurious dependences involving different escaping objects, i.e. two variables that may refer to escaping object but are not aliases. This situation can be addressed by leveraging an alias analysis. Nevertheless, this can be imprecise in cases where the concerned variables are aliases and refer to escaping objects, but occur in the same thread. This case cannot be addressed even by alias analysis. However, this can be achieved by leveraging *waits*, *notifies*, and *rdEntities* elements of the alias sets.

For two expressions to be related by ready dependence according to rule 4 in the definition of ready dependence, one of the expression should invoke `wait()` while the other expression invokes `notify()` and both invocations should be on the same receiver object in different threads. The last requirement can be approximated as both invocations should involve receiver variables that are mutual aliases occurring in different threads.

The wait-notify coupling implied by the first two requirements is captured by the *waits* and *notifies* elements of the alias set along with the rules to set them during the second phase of the analysis. As for the calculation of mutual aliasing, we can rely on the fact that unified alias sets corresponding to aliasing variables. However, we need to record this unification as proof of aliasing. For this, we can rely on using entities. Specifically, if two alias sets correspond to variables that are mutual aliases, then there should be a entity common to both alias sets. However, the question in case of ready dependence – "how to capture *inter-thread* aliasing?"

Observe that, given two alias sets that are to be unified, the aliasing spans thread boundaries if the *escapes* element of either of the alias sets is set to *true*.

Hence, given two alias sets that are to be unified with *waits* element set to *true* in one and *notifies* element set to *true* in the other, the corresponding variables can contribute a ready dependence via wait-notify coupling if the *escapes* element of either of the alias sets is set to *true*. This is the knowledge encoded in the unification rules in Table 3.1. Once this information is calculated in the second phase of the analysis, it is disseminated to the call sites that participate in ready dependence in the last phase of the analysis.

The benefits of this approach is empirically illustrated in Section 3.7.

Optimizing Rule 3 based Ready Dependence Similarly, the information about lock coupling (see Section 3.4.2) can be used detect and eliminate candidate ready dependences based on rule 3 if two escaping but non-aliasing variables provide the object to be locked/unlocked while entering/exiting synchronized blocks.

$\theta(\mathbf{arg1}, \mathbf{arg2}).\mathbf{escapes}$	**arg1.read**	**arg2.written**	$\theta(\mathbf{arg1}, \mathbf{arg2}).\mathbf{rwEntities}$
true	*true*	*true*	$arg1.rwEntities \cup arg2.rwEntities \cup \{newEntity\}$
false	-	-	$arg1.rwEntities \cup arg2.rwEntities$
-	*false*	-	$arg1.rwEntities \cup arg2.rwEntities$
-	-	*false*	$arg1.rwEntities \cup arg2.rwEntities$

Table 3.4: Rules to unify *rwEntities*.

3.6.3 Accurate Interference Dependence via Read-Write Entities

Similar to ready dependence, detection of interference based on escape information can be further improved to prune out dependences stemming from non-aliasing escaping primaries. As in case of ready dependence, the alias set is extended with the following three elements.

read is a boolean that when *true* indicates that a field of the variable/object was read,

written is a boolean that when *true* indicates that a field of the variable/object was written, and

rwEntities is a set of object entities that represent read-write coupling between the enclosing alias set and other alias sets. It is similar to *rdEntities* in meaning. From the nature of interference dependence, $rwEntities \neq \emptyset \Rightarrow escapes = true$.

Similar to the unification rule of *rdEntities*, the result of unifying *rwEntities* is determined by the *read* and *written* elements of the alias sets being unified. The *rwEntities* of the alias sets being unified is modified only when *read* element of one alias set is *true* and the *written* element of the other alias set is *true*. In this case, a fresh entity is injected into the *rwEntities* set. In all other cases, *rwEntities* set is unmodified in the result of unification.

With this information, access expressions can be considered for interference dependence only when the intersection of *rwEntities* sets of the alias sets corresponding to the primaries is non-empty. We refer to this notion of variables being related by possible concurrent read and write of a common field as *read-write coupling*.

Field-based Sharing Information

In the above extension, the fields that cause read-write coupling are not captured. This information may be useful to answer the questions – *does the object referred to by the given variable participate in the read-write coupling via the given field?* and *what fields of the object referred to by the given variable lead to read-write coupling?*

This information can be captured by adding new entity sets *readFields*, *writeFields*, and *signatureOfRWFields* to alias sets.

As for processing, upon reading or writing a field, the signature of the field is added to the *readFields* and *writeFields* sets, respectively, of the alias set corresponding to primary of the processed field access expression. Upon unifying an alias set a_1 with its *read* element set to *true* with an alias set a_2 with its *written* element set to *true*, the result of unification $\theta(a_1, a_2).signatureOfRWFields$ will be $a_1.readFields \cap a_2.writeFields$ if $\theta(a_1, a_2).escapes = true$.

As in case of side-effect information calculation (see Section 3.4.3), in phase three, the elements from these sets in the site context alias sets are not propagated into the corresponding sets in the corresponding method context alias sets. This prevents the read-write information of the caller method from polluting the read-write information of the callee method.

The elements of *signatureOfRWFields* of the alias set corresponding to the given variable can then be used to answer the above questions.

Write-Write Coupling Information

While the previous two extensions capture read-write coupling between objects, it may be useful to detect *write-write coupling* – the notion of relating variables via possible concurrent writes to common fields. This information can enable client analysis to easily identify an optimal collection of related concurrent writes for purposes such as synchronization injection, synchronization sensitive refactoring, calculation of may-happen-in-parallel (MHP) information, etc.

The extension to capture the required information is the addition of a new entity set *signatureOfWWFields* to alias sets that is similar to *signatureRWFields*. Drawing from the similarity, upon unifying an alias set a_1 with its *written* element set to *true* with an alias set a_2 with its *written* element set to *true*, the result of unification $\theta(a_1, a_2).signatureOfWWFields$ will be $a_1.writeFields \cap a_2.writeFields$ if $\theta(a_1, a_2).escapes = true$. The elements of this entity set can be used by the above mentioned purposes.

3.6.4 Aliasing-based Data Dependence

Another trivial application of the equivalence class based analysis is in the calculation of (intra-thread) aliasing-based data dependence. Basically, the dependence calculation can use the aliasing information calculated by the extension (Section 3.4.1) instead of the information calculated by an iterative flow analysis based alias/points-to analysis [Ran02, LH03, And94]. This approach will be cost efficient in the context of sizable input programs as the extension has linear time complexity while the iterative flow analysis based approach generally have cubic time complexity.

However, as the core escape analysis relies on a pre-computed call graph, the accuracy of the aliasing information provided by the extension will be directly dependent on the accuracy of the call graph. Hence, inaccurate call graphs (such as call graphs based on rapid type analysis (RTA) [BS96] as opposed to call graph based on points-to (object flow) analysis [Ran02]) can lead to inaccurate aliasing information.

So, depending on the accuracy of the available call graph and the desired accuracy for aliasing information, the proposed extension can be trivially leveraged to compute aliasing-based data dependence.

3.6.5 Atomicity and Independence

While analyzing concurrent systems, recent analysis try to optimize the processing by not analyzing *atomic methods — no shared data is accessed upon entering the method and until exiting the method.* It is hard to detect this information as an object that is not shared till the method exits may subsequently become shared. A dynamic analysis or an expensive path- and context sensitive static

analysis can detect such situation by keeping track of the accurate variable to object mappings. A path insensitive and flow insensitive analysis such as our escape analysis cannot provide highly accurate information.

As our escape analysis calculates may-sharing information — *a escaping variable may refer to a shared object*, we can leverage the negation of this information — an non-escaping variable will not refer to a shared object. Specifically, for any given method, we can explore every alias sets reachable (inclusive) from the corresponding alias context and check if any of them are marked as shared. If not, then the method can be safely flagged as atomic.

We can extend this technique to statements as well. For a given statement, we can explore alias sets reachable (inclusive) from the alias sets corresponding to the variables occurring in the given statement and check if any of them are marked as shared. If not, then the statement is *independent — the execution of the statement neither depends on nor influences the concurrent execution of statements*. This information can be used in program verification and similar contexts where the behavior of concurrent programs is analyzed by exploring all possible interleaved execution traces of the programs. Specifically, independent statements can be executed atomically with either their predecessor (or successor) and avoid an interleaving of concurrent threads between independent statement and it's predecessor (or successor).

As sharing of objects in Java leads to interaction between threads via locking-unlocking and reading-writing, we could leverage more accurate information pertaining to various couplings to further improve the accuracy of atomicity and independence information.

3.6.6 Property-sensitive Program Slicing

As the described analysis maintains contextual information pertaining to call paths, the maintained information is also used in property-sensitive program slicing. Details about this new form of slicing and how the information from the analysis is leveraged is explained in Chapter 5.

3.6.7 Partial Order Reductions

Read-write, wait-notify, and locking coupling are the only forms of inter-process communications via shared objects in Java programs. Hence, this information can be leveraged in various contexts to detect possible interaction points between threads of concurrent Java programs. One such context is using partial order reductions to reduce the number of interleavings considered while exploring the behavioral space of a Java program in applications such as program verification via model checking. The details of how the information from the analysis can be used to achieve partial order reduction is explained in Chapter 6.

3.7 Empirical Evaluation

In this section, we shall describe the experiments conducted to evaluate the analysis along with a discussion of the results from the experiments.

3.7.1 Implementation

All variants of the analysis described in the previous sections were implemented as part of the StaticAnalyses module of Indus, a project aimed at providing a program analysis and transformation toolkit for Java[12], and exercised in realizing a full-featured Java Slicer also provided by Indus. Deviating from the descriptions in the previous chapters, Indus operates on the Jimple representation of the Java programs and not directly on Java programs. As Jimple [VR00] is a 3-address based intermediate representation that supports OO features, Java programs can be translated into an equivalent Jimple program; hence, the results of the experiments apply directly to the subject Java programs.

Ruf's analysis was based on a SSA representation. Instead we achieve a similar effect by using *local variable splitting* transformation of Soot. This transformation ensures that variables are defined only once along every non-cyclic path beginning from the start node in the intra-procedural control flow graph of the containing method.

3.7.2 Experimental Setup

All experiments were conducted on a 1.1GHz Linux box with 1GBs of RAM using Sun JDK1.5.0_07 with a maximum heap-size of 512MB.

The timing data is not presented as all of the experiments completed in one wall-clock minute (inclusive of parsing, object-flow analysis, escape analysis, various dependence analysis, and serialization of calculated information).

The time and memory measurements were collected by instrumenting the code via AspectJ[13].

The programs from Java Grande[14] benchmark suite were used as the input/subject programs in the experiments. A Grande application is an application which has large requirements for any or all of: memory, bandwidth, and processing power. Java Grande suite is divided into three sections:

Low-level operations This class of benchmarks measure low-level operations such as thread creation, fork/join (FJ), barrier (Bar), and synchronization (Syn).

Kernels This class of benchmarks encompass specific operations (Ser, LUF, SOR, Crp, SMM) that are frequently used in Grande applications.

Large Scale Applications This class of benchmarks are representative of Grande applications (MD, MC, RT).

The number of classes and methods (both application and library) that make up the benchmarks in each section is given in Table 3.5.

Due to constraints of the presentation medium in displaying the names on the graphs, we shall use abbreviations to identify the programs while presenting the data.

In each experiment, each method with the signature `public static void main(String[])` as representing a possible entry point into the program.

[12] http://indus.projects.cis.ksu.edu
[13] http://www.eclipse.org/aspectj
[14] http://www.epcc.ed.ac.uk/javagrande/

Benchmark	Classes	Methods
Bar	121	301
Crp	133	336
FJ	117	299
LUF	121	316
MD	123	324
MC	250	776
RT	132	363
Ser	119	313
SOR	133	335
SMM	133	337
Syn	119	297

Table 3.5: The size of the benchmarks.

As for the steps in the experiments, object flow analysis was first performed in object-sensitive mode [Ran02] to calculate accurate value flow information to aid the construction of an accurate call graph. The call graph was then used to perform the escape analysis exercising various optimizations. Based on the calculated information, various dependence analysis were executed in different accuracy modes.

3.7.3 Escape Analysis

For each of the benchmarks, escape analysis was executed in two modes: *opt1* escape analysis was executed with only the optimization described in Section 3.5.1, and *opt1+opt2* escape analysis was executed with optimizations described in Section 3.5.1 and Section 3.5.2.[15]

The data from these experiments, both raw numbers and graphical representation, is available in Table 3.6 and Figure 3.10. The data indicates the number of reference type variables in the benchmarks that were flagged as referring to escaping, read-write coupled, write-write coupled, and locking coupled objects by the analysis.

The data from the experiments indicates that the static field access optimization described in Section 3.5.2 provides interesting improvement (ranging from 5% upto 90% reduction in various cases) of the escape, read-write coupling, write-write coupling, and locking information in comparison with the unoptimized mode. The extent of improvement will only increase if compared with the purely unoptimized mode of escape analysis (that would be very close to Ruf's analysis).

The data for method atomicity also follows the cue of the data for other aspects of the analysis and indicates improvement in accuracy. However, the improvement is not emphatic as escape information was used to determine atomicity as opposed to various coupling information.

As opposed to the reduction in the numbers, we see the number of methods flagged as atomic increases. This is consistent with the usage of the negation of the information as described in Section 3.6.5.

The only immediately disturbing data corresponds to wait-notify coupling. The data indicates that the analysis detects wait-notify coupling in only `Bar` benchmark. This is due to the fact that only `Bar` benchmark uses wait-notify pattern of synchronization.

[15] We shall present empirical data for the type filtering optimization in Section 3.7.8.

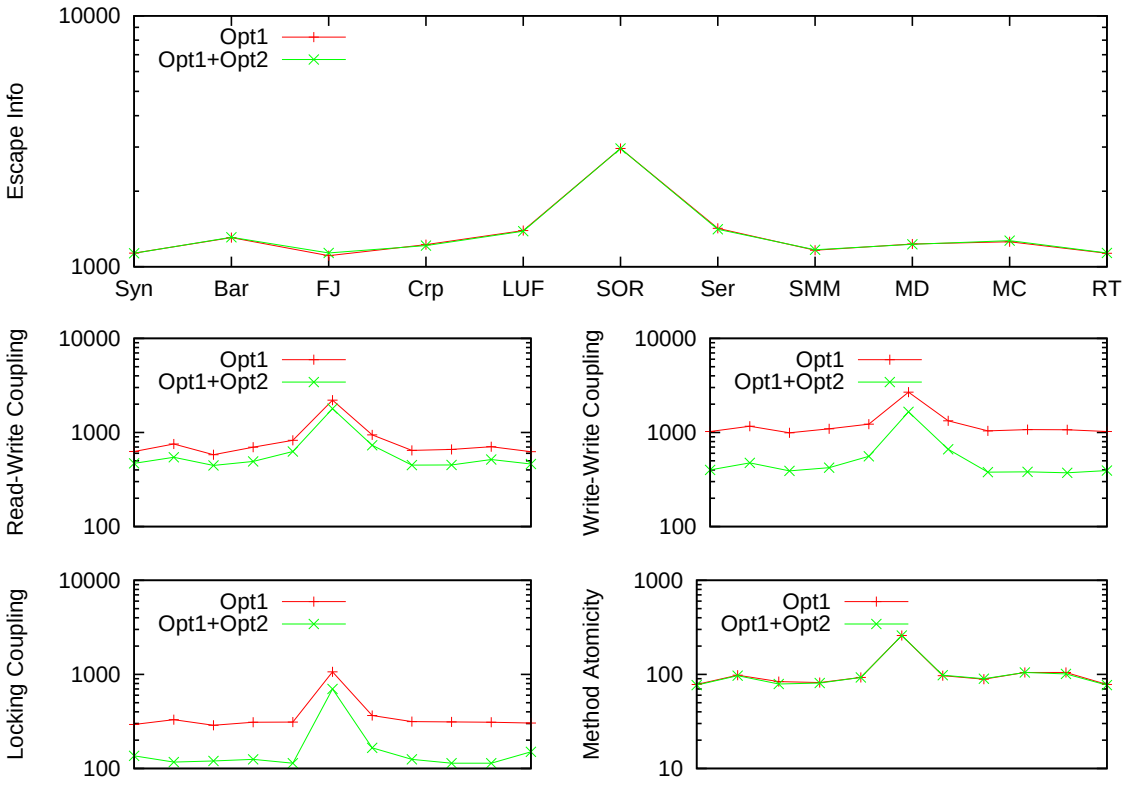

Figure 3.10: Summary of data calculated by equivalence-based escape analysis in various experiments. The raw data is available in Table 3.6. Logarithmic scale is used for data on the y-axis.

Benchmark	Escapes		Read-Write		Write-Write		Locking		Wait-Notify		Atomicity	
	opt1	opt1+2	opt1	opt1+2	opt1	opt1+2	opt1	opt1+2	opt1	opt1+2	opt1	opt1+2
Syn	1132	1134	625	463	1022	393	304	150	0	0	78	77
Bar	1133	1131	628	470	1025	400	293	136	0	0	78	77
FJ	1106	1135	581	447	992	391	288	120	0	0	84	79
Crp	1306	1310	754	545	1166	475	330	117	0	0	98	97
LUF	1224	1214	697	492	1094	422	310	125	0	0	82	81
SOR	1232	1229	660	452	1072	382	312	114	0	0	105	105
Ser	1163	1168	644	449	1041	379	315	125	0	0	89	90
SMM	1257	1270	707	515	1067	373	310	114	0	0	105	101
MD	1393	1385	827	627	1224	557	311	114	0	0	93	93
MC	2958	2958	2204	1792	2685	1658	1064	700	0	0	259	259
RT	1422	1410	943	732	1332	664	366	166	0	0	97	98

Table 3.6: Data calculated by equivalence-based escape analysis in various experiments.

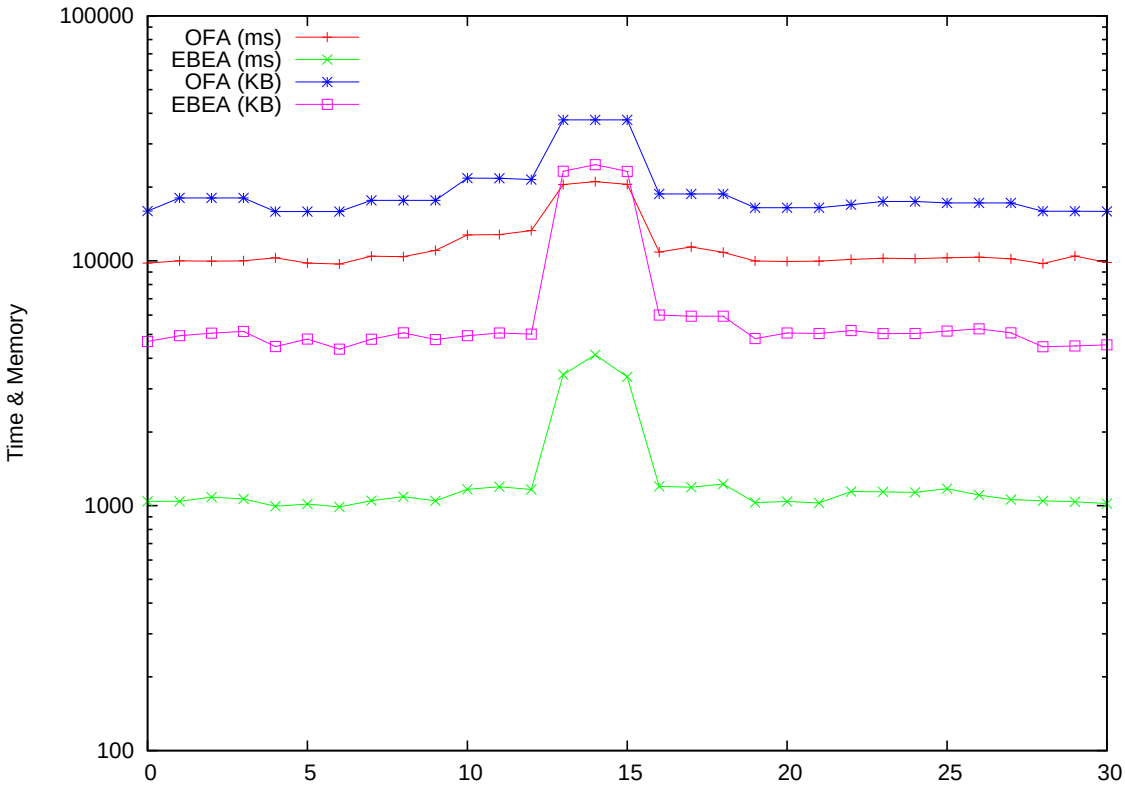

Figure 3.11: Summary of time and memory required by iterative object-flow analysis and by equivalence-based escape analysis in various experiments. The raw data is available in Table 3.7. Logarithmic scale is used for data on the y-axis.

3.7.4 Alias Analysis

As mentioned in the earlier sections, our escape analysis can be extended to provide aliasing information. The data collected in our experiments (Figure 3.11 and Table 3.7) support the claim (not the complexity analysis result) that the time and memory required by the escape analysis (hence, the alias analysis) is small in comparison with that of an iterative object sensitive and flow sensitive flow analysis.

During object flow analysis, every string literal in the input program was abstracted into a single abstract string literal to speed up the execution of the analysis. This inaccurate setting leads to lower/better performance numbers for object flow analysis. Hence, the timing and/or space data pertaining to object flow analysis will only worsen in comparison with escape analysis in more accurate settings.

On more close examination of the data, the processing time required by both the analyses increased rather linearly with the size of the reachable methods in the input programs. Although this observation may not agree with the worst case cubic time complexity of the flow analysis, it agrees with the worst case linear time complexity of our analysis. Comparatively, the time required by our analysis was in the range of 9-20% of the time taken by the object flow analysis.

OFA (ms)	EBEA (ms)	OFA (KB)	EBEA (KB)
9458	1051	15773	4218
8956	979	15774	4189
9263	1002	15773	4218
10209	1141	17879	4731
10177	1144	17879	4731
10223	1139	17879	4731
9412	1060	15719	4089
9363	1026	15719	4089
9387	1020	15719	4089
10061	1142	17533	4369
10049	1141	17533	4369
10041	1155	17533	4350
12867	1269	21488	4809
12844	1232	21488	4820
12852	1232	21488	4820
19387	3690	36991	18955
19421	3657	36991	18961
19370	3723	36991	18955
10148	1254	18534	5275
10108	1268	18536	5277
10136	1265	18534	5305
9545	1103	16194	4182
9532	1132	16194	4167
9533	1114	16194	4182
9869	1182	16754	4617
9888	1167	16754	4617
9373	1136	16754	4617
9938	1248	16975	5168
9942	1243	16975	5168
9988	1278	16975	5168
9483	1070	15684	4265
10370	1080	15684	4273
9534	1053	15684	4273

Table 3.7: Time and memory required by iterative object-flow analysis (OFA) [Ran02] and equivalence-based escape analysis (EBEA) in various experiments.

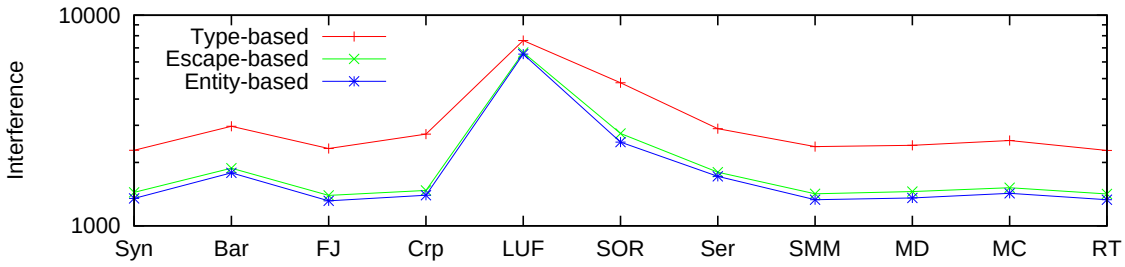

Figure 3.12: Summary of interference dependences calculated based on type, escape, and entity information. Raw data is available in Table 3.8. Logarithmic scale is used for data on the y-axis.

In case of memory requirement, data indicates that the analyses requires memory rather linear in the number of the reachable methods in the input programs. The memory requirement for our analysis increases rather rapidly with the size of the input program. However, this is still below the predicted worst case complexity. Comparatively, the memory required by our analysis was in the range of 23-66% of the memory taken by the object flow analysis.

More interesting data pertaining to memory requirement is presented in Section 3.7.8 in the context of large programs.

Although the data supports our complexity predictions, we believe more empirical study is required to predict costs in general scenarios.

3.7.5 Interference Dependences

In the experiments pertaining to interference dependences, we exercised the algorithm described in Figure 3.5 with three different instantiations of the Γ function.

Type-based This instance used type information to detect if the primaries of field access expressions could lead to interferences.

Escape-based This instance used escape information from our analysis to detect primaries leading to interferences.

Entity-based This instance used relied on entity information (read-write coupling) to detect interference.

We combined these instances with the combination of two modes of escape analysis (*opt1* and *opt1+opt2* (as described in Section 3.7.3) and with the aliasing information calculated based on object flow analysis (*OFA*) (as described in Section 3.2.2). In case of type-based version of Γ, escape analysis and its optimizations do not affect the result; hence, they are not presented in the detailed graph (Figure 3.13).

The interference dependence data in unoptimized mode (given in Figure 3.12) supports our claim that escape information can improves the accuracy of the interference dependence calculation and entity information can contribute further improvements.

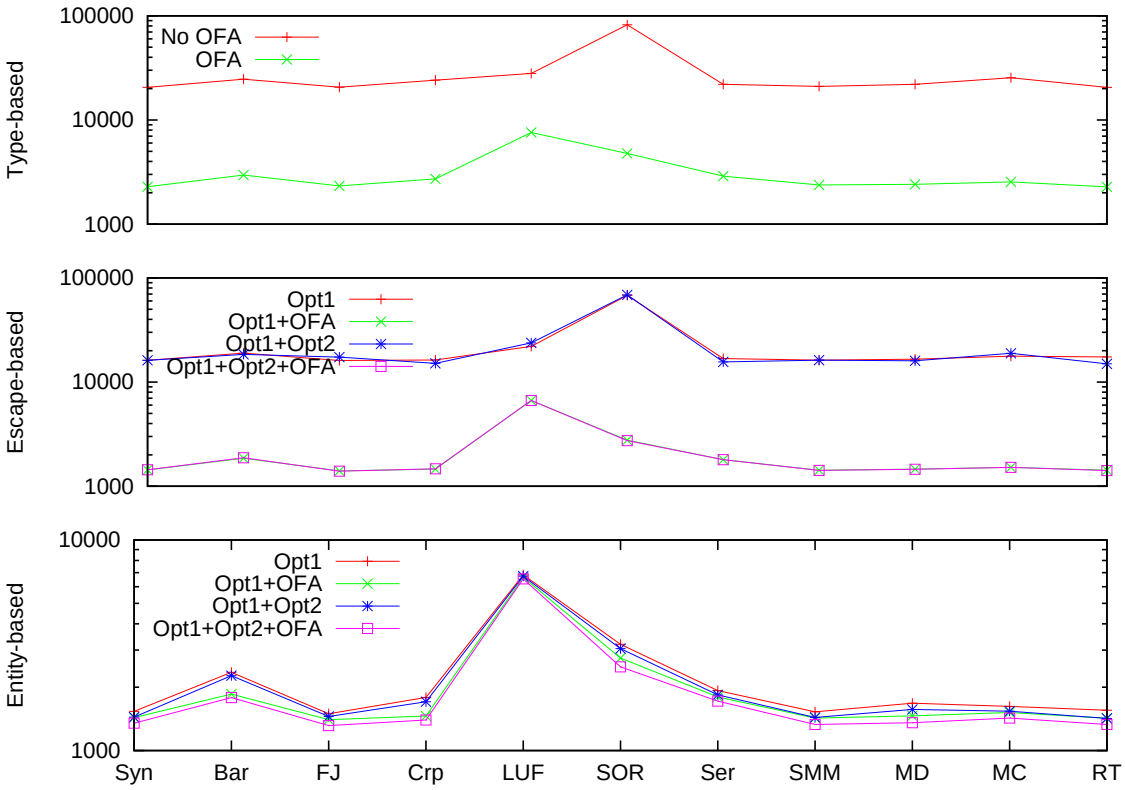

Figure 3.13: Details of interference dependences calculated based on type, escape, and entity information and along with other optimizations. Logarithmic scale is used for data on the y-axis.

Benchmark	Type-based				Escape-based				Entity-based			
	unopt	ofa	opt	ofa+opt	unopt	ofa	opt	ofa+opt	unopt	ofa	opt	ofa+opt
Syn	20581	2278	20581	2278	17409	1423	14987	1416	1555	1423	1421	1329
Bar	20582	2285	20582	2285	16184	1435	16184	1442	1536	1435	1442	1349
FJ	20633	2330	20633	2330	16144	1402	17382	1395	1496	1402	1452	1314
Crp	24664	2967	24664	2967	18989	1854	18420	1877	2351	1854	2266	1786
LUF	24113	2723	24113	2723	16294	1459	15110	1471	1787	1459	1704	1397
SOR	22027	2412	22027	2412	16609	1462	15992	1455	1679	1462	1568	1356
Ser	21037	2378	21037	2378	16240	1426	16240	1419	1526	1426	1437	1331
SMM	25469	2541	25469	2541	17743	1516	18929	1516	1622	1516	1538	1427
MD	27993	7591	27993	7591	21992	6648	23794	6648	6823	6648	6735	6535
MC	82114	4783	82114	4783	67976	2766	68695	2744	3198	2747	3048	2497
RT	22021	2894	22021	2894	16815	1796	15631	1796	1927	1796	1838	1718

Table 3.8: Number of interference dependences calculated based on type, escape, and entity information and along with other optimizations.

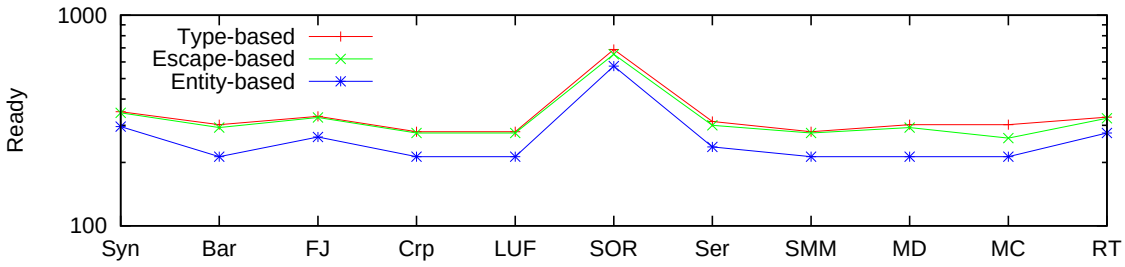

Figure 3.14: Summary of ready dependences calculated based on type, escape, and entity information. Raw data is available in Table 3.9. Logarithmic scale is used for data on the y-axis.

By examining the data in detail (type-based and escape-based) Figure 3.13 and Table 3.8), we find that using aliasing information based on OFA produces a huge improvement that is better than the improvement achieved by only using escape information. However, read-write coupling information based on entities yields greater reduction than that obtained by leveraging OFA information.

3.7.6 Ready Dependences

We conducted ready dependence related experiments in a manner identical to interference dependence related experiments. The data indicates that an improvement in accuracy of ready dependence between escape information based approach and type-based approach.

However, despite using the same techniques and optimization, entity information based approach did not provide extra improvements over the escape information based approach. We believe this could be due to the relatively low number of synchronization statements in the benchmarks.

Interestingly, the improvement in accuracy of ready dependence was the largest when aliasing information from OFA was combined with the entity information from the escape analysis. However, given the restrictive nature of the benchmarks, we cannot conclude that this combination will work the best in all cases.

In short, the data supports our claim that leveraging escape and entity information improves the accuracy of ready dependence, but requires more empirical evaluation to predict expected gains in general scenarios.

3.7.7 Aliasing-based Data Dependences (ABDD)

In our experiments, we used an algorithm similar to the one described in Figure 3.5 to calculate aliasing based data dependence. We ran this algorithm with three different instantiations: *type-based*, *escape-based*, and *entity-based*, of the Γ function as described in Section 3.7.5. However, unlike in case of interference and ready dependence, we only executed entity-based mode of the experiment in the most optimal mode as the proposed optimizations do not affect the accuracy of the calculated aliasing information.

The results from the experiments (Figure 3.16 and Table 3.10) clearly indicate that ABDD based on aliasing information from OFA is more accurate than the ABDD based on type information.

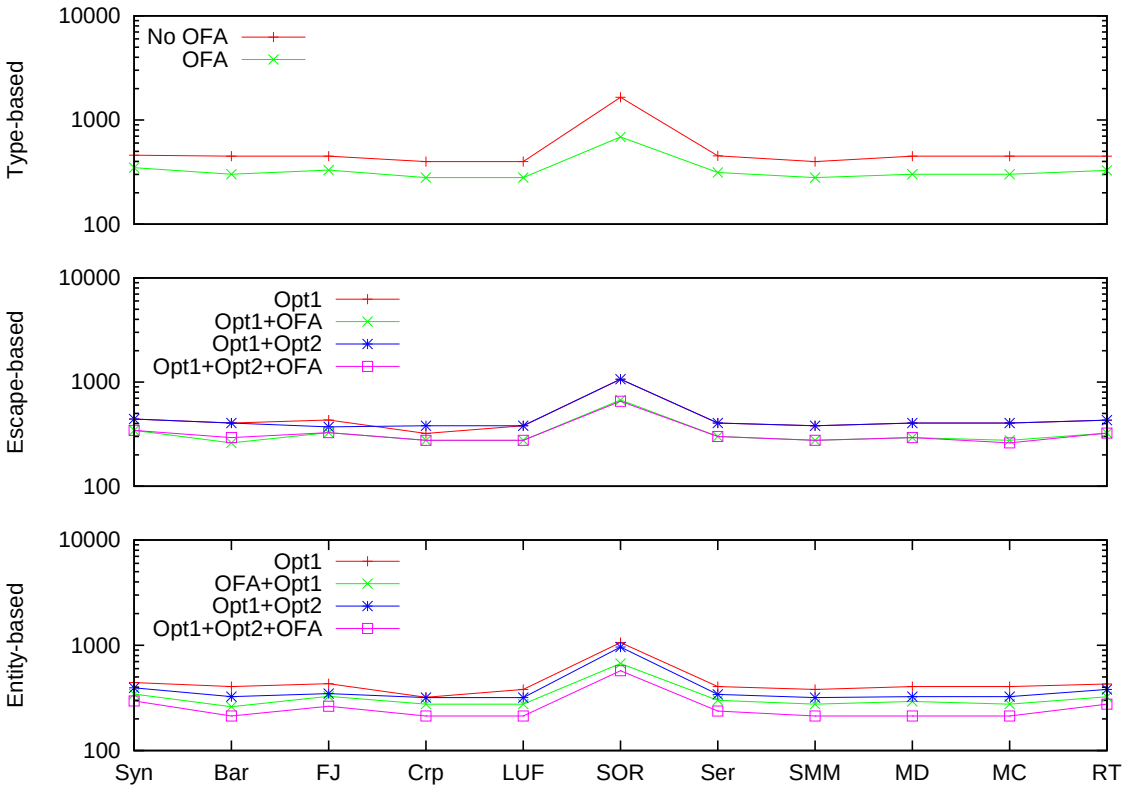

Figure 3.15: Details of ready dependences calculated based on type, escape, and entity information. Logarithmic scale is used for data on the y-axis.

Benchmark	Type-based				Escape-based				Entity-based			
	unopt	ofa	opt	ofa+opt	unopt	ofa	opt	ofa+opt	unopt	ofa	opt	ofa+opt
Syn	449	328	449	328	431	324	431	324	429	324	381	276
Bar	460	348	460	348	442	344	442	344	442	344	394	296
FJ	450	331	450	331	432	327	372	327	432	327	348	264
Crp	449	302	449	302	405	261	405	293	405	261	325	213
LUF	399	280	399	280	321	276	381	276	321	276	318	213
SOR	449	302	449	302	405	293	405	293	405	293	325	213
Ser	399	280	399	280	381	276	381	276	381	276	318	213
SMM	449	302	449	302	405	276	405	261	405	276	325	213
MD	399	280	399	280	381	276	381	276	381	276	318	213
MC	1650	688	1650	688	1065	672	1065	653	1061	672	962	573
RT	453	313	453	313	405	300	405	300	405	300	342	237

Table 3.9: Number of ready dependences calculated based on type, escape, and entity information and along with other optimizations.

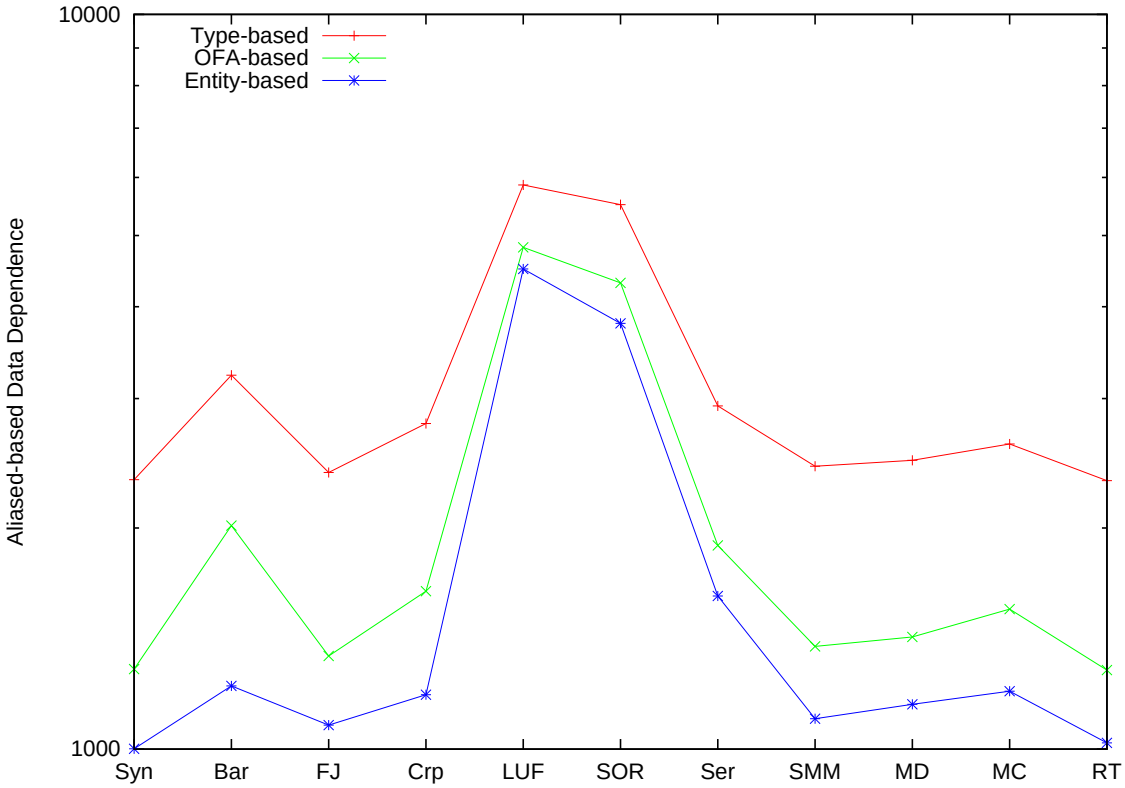

Figure 3.16: Summary of aliasing-based data dependences calculated based on type, OFA, and entity (opt1+2) information. Raw data is available in Table 3.10. Logarithmic scale is used for data on the y-axis.

Benchmark	Type-based	OFA-based	Entity-based
Syn	2320	1281	1020
Bar	2327	1285	1001
FJ	2380	1339	1078
Crp	3228	2015	1219
LUF	2775	1640	1186
SOR	2472	1421	1151
Ser	2427	1380	1100
SMM	2602	1551	1199
MD	5858	4819	4503
MC	5509	4312	3798
RT	2932	1893	1616

Table 3.10: Number of aliasing-based data dependences calculated based on type, OFA, and entity (opt1+2) information.

Further, they indicate that ABDD based on entity information is either equally accurate or more accurate than ABDD based on aliasing information from OFA. Hence, this suggest that aliasing information based on our efficient escape analysis can be used in place of flow analysis based aliasing information with minimal loss of accuracy.[16]

3.7.8 Type Filtering

In the previous experiments, our analysis with and without various optimizations other than type filtering scaled well as the input programs were small. Hence, we did not include the data to illustrating the effect of type filtering (which was minimal).

To illustrate the effect of type filtering, we considered two non-trivial Java applications – JReversePro 1.4.1[17], a Java Classfile Decompiler/Disassembler, and JEdit 4.1[18], a full-featured (multi-threaded) editor. JReversePro is composed of 90 classes, 745 methods, 347 fields, and 216KB of application class bytecodes while JEdit is composed of 808 classes, 4954 methods, 1967 fields, and 2165KB of application class bytecodes.

The data from the experiments is presented in Table 3.11. We analyzed JReversePro and JEdit in four different modes:

vanilla modes exercised the optimization described in Section 3.5.1.

staticField mode exercised the optimizations described in Sections 3.5.1 and 3.5.2.

typeFiltering mode exercised the optimizations described in Sections 3.5.1 and 3.5.3.

both mode exercised the optimizations described in Sections 3.5.1, 3.5.2, and 3.5.3.

The time and memory required by the escape analysis are given column 2 and 3. The number of variables marked as escaping and as participating in concurrent/shared read-write, write-write, lock-unlock, and wait-notify couplings are given in columns 4, 5, 6, 7, and 8, respectively. The number of methods marked as *atomic* is given in columns 9.

The data from the experiment indicate that the improvement in the accuracy of various information calculated by the analysis with each optimization independently is only marginally in comparison with the information when minimal optimizations is used. We suspect that this might be due to the sequential nature of JReversePro.

However, the data indicates remarkable improvement (ranging from 10% to 24%) at nominal increase in cost when the optimizations are combined. This validates the optimizations, but it does not validate the correctness of the optimization. Upon random examination of the data, the data did seem to be valid. Further, this improvement indicates that the marginal improvement in accuracy when optimizations are applied independently may be due to the requirement of not subjecting global alias sets (with *global* element set to *true*) to type filtering in *typeFiltering* mode and not using type filtering optimization (specifically while processing global alias sets) in *staticField* mode.

In summary, it would be beneficial to apply the proposed optimizations to improve the accuracy at a nominal cost; however, techniques and approaches to easily detect if the result is unsound need to be further explored.

[16] Please refer to Section 5.4.1 for information about a simple yet powerful optimization used in the calculation of ABDD in these experiments.
[17] http://jrevpro.sourceforge.net
[18] http://www.jedit.org

Mode	Time	Memory	Escapes	Read-Write	Write-Write	Locking	Wait-Notify	Atomic
	(s)	(MB)						
vanilla	48669(113457)	195(303)	28379	25626	26611	21289	20342	477
staticField	154092(145697)	723(306)	28397	25583	25491	21380	20469	476
typeFiltering	50098(153665)	190(302)	27882	25112	26059	21192	20269	518
both	81950(140091)	242(299)	23831	20962	20663	19442	18772	801

JReversePro v1.4.1

Mode	Time	Memory	Escapes	Read-Write	Write-Write	Locking	Wait-Notify	Atomic
	(s)	(MB)						
vanilla	82786(377718)	266(528)	41069	38421	39028	30790	28974	578
staticField	155171(363511)	613(521)	41055	38373	38271	30741	28947	576
typeFiltering	78466(365857)	266(520)	40878	38228	38831	30650	28846	585
both	105884(379307)	310(519)	34340	31669	31435	28979	27890	823

JEdit v4.1

Table 3.11: Data from the escape analysis of JReversePro and JEdit with various optimizations. The numbers within parentheses in columns 2 and 3 are time and memory data corresponding to object flow analysis. These experiments were conducted by using JVM in Sun JDK 1.5 with 1.7GB of maximum allowed heap space on 2GHz Linux Box with 2GB of RAM.

3.8 Related Work

Numerous escape analyses have been proposed to calculate if an object escapes (i.e. can be accessed outside of) a particular method m and/or thread t. Ruf's equivalence-class-based analysis [Ruf00] for calculating if an object is only accessed in a single thread and Choi et al. fixed-point-based analysis [CGS+99] to calculate if an object escapes a method or a thread are two well-known escape analyses. Aldrich et al. [ACSE99], Blanchet [Bla99], and Bogda and Hölzle [BH99] also proposed similar analyses to improve the runtime performance of Java programs by removing unnecessary synchronization and to enable stack allocation of objects.

Similarly, numerous data-race detection algorithms have been proposed. Choi et al. propose an on-the-fly technique [CLL+02] to detect data-race conditions (i.e. situations where accesses to shared objects are not protected by locks). Unlike our approach, this approach requires instrumentation and various optimizations such as static data-race detection analysis as a pre-phase to the dynamic analysis [CLS01]. Flanagan and Freund proposed an annotated type-based technique [FF00] along with an annotation inference mechanism which is similar to escape analysis [FF01] for the purpose of detecting race-condition by ensuring objects annotated/marked as thread-local are indeed thread-local (i.e. through the entire execution, they are only reachable from a particular thread t). A similar sort of analysis by Boyapati and Rinard [BR01] required extensions to Concurrent Java [FF00] and used the auxiliary information to deduce escape information. In comparison, the proposed approach is fully automatic and requires no user intervention. In terms of scalability, the complexity of these approaches were not described analytically; hence, an analytical comparison of the proposed approach and these previous approaches is not available.

Most of the efforts related to aliasing information has been based on points-to analysis. Numerous iterative [And94, Ste96, Das00, Ran02, LH03, MRR02] and equivalence class based [LH99, LH01] points-to analysis have been proposed. These efforts have been focused on modularity, parameterization, scalability, accuracy, and various forms of sensitivities in the context of sequential programs. Sălcianu and Rinard [SR01] proposed a combined iterative algorithm to calculate points-to and escape information in the context of concurrent programs.

There is a wide body of literature on slicing sequential programs, beginning with Weiser's original paper on slicing [Wei84]. Horwitz et al. [HRB88] proposed a inter-procedural program slicing algorithm which has been extended by others to handle various features such as exceptions and unconditional jumps. There has been effort ([LH96, HPR89, LR94, LH98]) in addressing issues involved in slicing program written in object oriented languages with features such as pointers and/or references.

In 2004, Hammer et al. [HS04] independently proposed an approach to improve the accuracy of Java program slicing by addressing the inaccuracies stemming from aliasing between nodes in the object graphs of method parameters in a sequential setting. In contrast, my effort focuses on the calculation and use of accurate object escape information to improve the accuracy of data (and control) dependences in both sequential and concurrent setting.

Krinke [Kri98] considered intra-procedural slicing for a simple **while**-language with **co-begin/co-end** statements and proposed a form of symbolic execution to prune interference dependences starting from the observation that considering interference dependences to be transitive is overly conservative. Nanda [Nan01] proposed algorithms for context-sensitive algorithms for slicing concurrent programs. However, these algorithms rely on symbolic execution and/or *may-happen-in-parallel(MHP)* [ACSE99, NAC98] information to prune interference dependences. However, in the absence of synchronization operations, the use of MHP algorithms will not provide large reductions in the dependences as it is harder to detect instruction execution ordering. Also, these techniques

does not address pruning of interference dependences arising from language features such as dynamically created objects and threads. As the justification for pruning interference dependences are independent (thread locality of objects vs instruction execution ordering), orthogonal pruning techniques such as ours and those mentioned above can be combined to obtain further reductions.

Specifically in the context of ready dependence, Hatcliff et al. [HCD+99] proposed safe locks based optimization to improve accuracy. A lock is deemed as *safe if the lock will not be held indefinitely at runtime*. We have implemented an analysis to calculate safe locks and have found that it provides good accuracy improvements independently and even better accuracy improvements along with the optimization proposed in this chapter.

Chapter 4

Constrained Java

Although the approaches/techniques described in this dissertation are applicable to Java programming language, they are described in the context of a *Constrained Java* language to simplify their exposition.

Constrained Java (CJava) language is the Java programming language along with few structural constraints and one semantic alteration.

CJava is based on Java language as described in the *second edition of the Java Language specification* [GJS00] with the underlying execution semantics as described in the *second edition of Java Virtual Machine (JVM) specification* [LY99]. The structural constraints restrict the usage of certain grammar rules defined in the Java language specification and mandate the use of certain Java coding patterns. Hence, *every valid CJava program is also a valid Java program.*

The basic purpose of the constraints is to simplify the structure of CJava programs to reproduce the basic property of three-address form code — *there are at most three addresses in a statement with at most one assignment and at most one operation.* The reasons for using CJava instead of using a three-address form based intermediate language are

- to describe the approaches/techniques in a language with well defined and documented syntax and semantics (not mere representation),
- to describe the approaches/techniques in Java (the intended target language) as opposed to three-address code based intermediate languages and, consequently,
- to ease the transition of the proposed approaches/techniques into the Java application domain.

As CJava merely removes most of the existing syntactic sugaring in Java and enforces certain rules to explicate existing implicit syntactic interpretation of Java programs, *the proposed techniques can be directly applied to Java programs.*

4.1 Structural Constraints in CJava

4.1.1 Assignment Constraints

In Java, an assignment statement is also an expression that evaluates to the assigned value. Hence, it is possible to chain assignment statements, e.g. `a = b = c;`. Such constructions complicate the description of algorithms that refer to variable defining sub-expressions. To avoid complications, such constructions are disallowed in CJava programs.

Constraint 1 An expression can have at most one assignment operator.

Hence, the Java expression `a = b = c;` should be written as `b = c; a = b;` in CJava.

A situation similar to assignment chaining occurs when prefix/postfix increment/decrement expressions (`--x/x--`) (involving a hidden assignment to the involved prefix/postfix expression (`x`)) are used with the assignment operator. Hence, such constructions are disallowed in CJava programs as well.

Constraint 2 Pre/Post increment/decrement operators are not supported.

Java expressions such as `a = b++;` should be written as `a = b; b = b + 1;` in CJava.

4.1.2 Array/Field Access Constraints

While accessing a field in Java, the array reference/field (`a`/`b`) of the array/field access expression (`a[i]`/`b.f`) is read first and then the index/field expression (`i`/`f`) is accessed in the read object. As each array/field access expression accesses two address locations (the array reference/primary and the cell/field), multiple array/field access expressions in a statement will violate the desired three-address code form of CJava programs. Hence, such constructions are disallowed in CJava programs.

Constraint 3 Array/Field access expressions can only occur in assignment statements, and only one array/field access expression can occur in an assignment statement.

Java statements such as `a = b.f + c.g;` should be written as `f1 = b.f; g1 = c.g; a = f1 + g1;`.

4.1.3 Invocation Constraints

Java allows chaining of method invocations (e.g. `o = foo().bar()`) and embedding method invocations within method invocations (e.g. `o = foo(bar())`). These features complicate expositions that need to refer to a particular method invocation in an expression. To alleviate the confusion, CJava disallows the use of these features.

Constraint 4 Utmost one method invocation can occur in an expression.

Hence, the Java expression `o = foo().bar();` should be written as `t = foo(); o = t.bar();` in CJava. Similarly, `o = foo(bar());` in Java should be written as `t = bar(); o = foo(t);` in CJava.

4.1.4 Statement Constraints

As opposed to complicating the presentation by handling the syntactic sugar added to the language, the syntactic sugar statements: `for` and `switch`, are disallowed in CJava programs.

Constraint 5 `for` and `switch` statements are not supported.

Java statements such as `for(int i = 0; i < 10; i++) { ... }` will be written as `int i = 0; while (i < 10) { ... i++}` in CJava. Similarly, `switch (i) { case 1: ... ; case 2: ... }` should be written as `if (i == 1) { ... } else if (i == 2) { ... }`.

4.1.5 Method Constraints

Java allows assignment to formal parameters of methods, i.e. erase the binding between the parameter of the method to its argument. As parameters capture the data available at the method interfaces, overriding definitions can complicate the presentation of certain ideas. Hence, such definitions are disallowed in CJava by the following constraint.

Constraint 6 All formal parameters of methods should be declared as `final`.

A Java method `void conv(int temp) { ... }` should be written as `void conv(final int temp) { ... }` in CJava.

In Java methods, when a statement has no successor, the implicit semantics is for the enclosing method to return after executing this statement. In CJava, this implicit semantics is not assumed; hence, the exit/return points of methods should be explicitly provided.

Constraint 7 Every maximal non-repetitive intra-procedural control flow path should be terminated by a `return` statement.

Observe that this constraint allows methods without return points in CJava programs; however, in such methods, there can be no maximal finite intra-procedural control flow path. Hence, this constraint does not in any way limit the expressiveness of CJava in comparison with Java. Instead, it merely explicates the return points in methods.

`synchronized` methods in Java provide syntactic sugar to execute the method within a synchronized statement in which the `this` variable of the method (the `Class` object in case of static methods) is used as the lock expression. Unlike the `synchronized` statement, there is no explicit

executable program point that can be associated with such synchronization/locking operation in `synchronized` methods. For these reasons, `synchronized` methods are disallowed in CJava.

Constraint 8 `synchronized` methods are not supported.

Instead, Java `synchronized` methods should be written in CJava by wrapping the entire body of the method in a `synchronized` statement on the `this` variable in case of instance methods and the `Class` object in case of static methods.

4.1.6 Class Constraints

Java supports static fields and blocks to initialize static fields either when declaring them or in a static block. The initialization-at-declaration option is a syntactic sugar that is usually realized initialization-in-static-block option by a Java compiler. For simplicity, CJava only supports the initialization-in-static-block initialization option for static fields.

Constraint 9 Each class can have utmost one static block; static fields can only be initialized within a static block.

In the rest of this dissertation, (in the spirit of the Java Virtual Machine specification) the static initialization block is assumed to be executed as a method. This method is referred to as *class initializer method*. It has the Java signature `public synchronized static void <clinit>()` which in turn should rewritten according to the method constraint 8.

When inner classes or anonymous classes are used, the method scope needs to be considered to differentiate program points occurring in inner/anonymous classes from those occurring in the containing classes. For this purposes, these features are not supported in CJava.

Constraint 10 Anonymous classes and Inner classes are not supported.

Each of the proposed structural constraints are illustrated in Table 4.1.

4.2 Semantics of Field Resolution

Constraint 11 Contrary to Java, field resolution occurs at compile time in CJava.

To elaborate, consider the programs in figure 4.1. Upon executing the program in the left column as a Java program, the field `x` is resolved at runtime. Hence, the signature of the field `x` accessed in `C.foo()` will be dynamically calculated to be `<int B.x>`. If the same program is executed as a CJava program, then the signature of the field `x` is calculated to be `<int B.x>` at compile time and this information is used at runtime to access field `x`.

This alteration simplifies the presentation of various approaches by removing a source of variance, namely, accounting for runtime field resolution at analysis time.

```
class A {}

class B extends A {
  int x;
}

class C extends B {
  void foo() {
    this.x = 10;
  }
}
```

```
class A {
  int x;
}

class B extends A {}
```

Figure 4.1: Java programs that illustrate field resolution semantics in Java version $\leq$1.1 and version $\geq$1.2.

However, this alteration has a serious flaw. For example, consider the Java program in Figure 4.1. Suppose class `C` is compiled against classes `B` and `A` occurring in the left column but is executed with classes `B` and `A` occurring in the right column. The execution of this combination of classes will fail with a `NoSuchFieldError` exception as the referenced field with the signature `<int B.x>` does not exist (instead field with signature `<int A.x>` exists). On the other hand, such an execution will succeed if runtime field resolution is used (as done in Java programs).

Hence, to address this flaw and to apply the approaches described in this dissertation to Java programs, this semantic alteration should be disregarded. Instead, a static approximation of the runtime field resolution information (calculated possibly by leveraging results of a points-to analysis) should be considered while handling analysis/data pertaining to field access.

As a side note, this altered semantics was an issue in earlier versions ($<$ 1.2) of Java. Specifically, the Java language specification required field resolution to occur at runtime (see Section 8.3.3.2 in [GJS96]) while the Java virtual machined specification only supported compile time field resolution (see Section 5.2 in [LY96]). This discrepancy was later addressed in the Java 1.2 (see Section 5.4.3.2 in [LY99]).

Java	CJava
`a = b = c;`	`b = c;` `a = b;`
`a += b = c`	`b = c;` `a = a + b`
`a = b++;`	`a = b;` `b = b + 1;`

`m = o.f + p.g;`	`int f1 = o.f;`
	`int g1 = p.g;`
	`m = f1 + g1;`

`o = foo().bar();`	`t = foo();` `o = t.bar();`
`o = foo(bar());`	`t = bar();` `o = foo(t);`

`for (int i = 0; i < 10; i++) { ... }`	`int i = 0;`
	`while (i < 10) {`
	`  ...`
	`  i++;`
	`}`

`switch (i) {`
`  case 1:`
`  ...`
`  break;`
`  case 2:`
`  ...`
`}`
	`if (i = 1) {`
	`  ...`
	`} else if (i == 2) {`
	`  ...`
	`}`

`int conv(int f) { ... }`	`int conv(final int f) { ... }`

`void foo() {`
`  System.out.println("Hello");`
`}`
	`void foo() {`
	`  System.out.println("Hello");`
	`  return;`
	`}`

`void foo() {`
`  while(true);`
`}`
	`void foo() {`
	`  while(true);`
	`}`

`synchronized void foo() {`
`  ...`
`}`
	`void foo() {`
	`  synchronized(this) {`
	`  ...`
	`  }`
	`}`

`synchronized static void foo() {`
`  ...`
`}`
	`static void foo() {`
	`  synchronized(Foo.class) {`
	`  ...`
	`  }`
	`}`

Table 4.1: Examples of Java fragments and their equivalent CJava fragments.

Chapter 5

Program Slicing

Program Slicing is a program analysis calculates the parts of a given program that influence (or are influenced by) the program points of interest. Since its invention by Weiser [Wei84] in the early 1980s, various researchers [HRB90, AH90, Ven91, CCL98] have explored techniques, variations, and applications of program slicing for over two decades in sequential contexts. Over this period, the primary applications of program slicing has been in maintenance, debugging, comprehension, and testing of programs.

In the past decade, few efforts [Zha99, HCD+99, CB01, Nan01, Kri03a] have explored program slicing in the presence of aliasing and concurrency. The proposed algorithms have been mostly based on dependence graphs. Further, all algorithms, except Krinke's and Nanda's, have been calling context insensitive.

Although the application domain of program slicing has been expanded to include program verification and program security, program slicing as a technique is not yet widely used. Based on my personal experience from implementing a program slicing framework for Java, I conjecture that this may be due to the lack of simpler slicing algorithms that are scalable, accurate, and flexible (customizable), and the lack of an accessible implementation to experiment with program slicing.

As a novel contribution, I have independently proposed a parametric program slicing algorithm that is based on dependence information in relational form (*not* based on dependence graphs), is efficiently calling context sensitive, and leverages property sensitivity to efficiently improve accuracy in the presence aliasing in both sequential and concurrent programs. Further, as it is parametric, it can be used to realize and reason about different forms of slicing. In the spirit of the theme of the dissertation, I have layered few extensions on top of the algorithm to improve scalability and accuracy in an application/context specific manner. These contributions have been implemented in a publicly available program slicing framework for Java in the Indus project[1].

This chapter contains the description of these contributions along with preliminary empirical evaluation of the contributions based on their realization in the Indus project.

[1] http://indus.projects.cis.ksu.edu

```
1   public class Example1 {
2     void tempConv(int c) {
3       float f = c * 9 / 5 + 32;
4       if (f < 100) {
5         System.out.println (" Good Weather " + f);
6       } else {
7         System.out.println (" Hot Weather " + c);
8       }
9     }
10  }
```

Figure 5.1: A trivial example to illustrate graph based program slicing.

5.1 Motivation

5.1.1 Basics

Formally, *program slicing is an analysis that accepts a program P along with a collection of program points C and provides a collection of program points S such that either S influences the behavior of P at C*[2] *or C influences the behavior of P at S.* The collection of input program points C are referred to as *slice criteria* while the collection of output program points S are referred to as a *program slice of P*. In many literature, the criteria is also accompanied by a collection of variables that determine the subset of the behavior, i.e. the values of the variables, that should be preserved at the criteria in the slice. I consider this to be an engineering/implementation detail; hence, it is not addressed in this dissertation.

Program slices in which the slice influences the criteria is referred to as *backward slices* whereas the slices in which the criteria influences the slice is referred to as *forward slices*. I refer to this aspect of the slice as the *direction* of the slice.[3]

Independent of the direction, to construct a slice, the analysis explores various relations between the criteria and other programs points in the program to determine which of the other program points to include in the slice. As the definition of a variable influences the use of a variable, data dependence relation is considered during slicing. Similarly, as the execution of (control flow to) a program point can be decided by a decision at another program point, control dependence relation is also considered during slicing. Further, as the influence may be indirect (e.g. program point e_1 may influence the control flow to program point e_2 which in turn may influence the data accessed at program point $e_3 \in C$), the analysis will also need to consider the relation between the slice and other program points in the program. Hence, the slice is expanded till a fixed point is reached.

Mathematically, program slicing can perceived as the transitive closure of the union of a collection of dependence relations. Given this perception, if the dependence relations are represented in a directed graph (i.e. program points as nodes and dependence relations as edges), then the algorithm to calculate a program slice is merely a graph search algorithm for a non-existent node starting from the criteria nodes in the appropriate direction (for forward/backward slices). Such graphs that represent the dependence relations of a program are referred to as *program dependence graph (PDG)*.

[2] *A influences B* is to be read as the program points in A influence the program points in B when A and B are collections of program points.

[3] In the rest of this chapter, unless specified, program slice/slicing implies backward program slice/slicing.

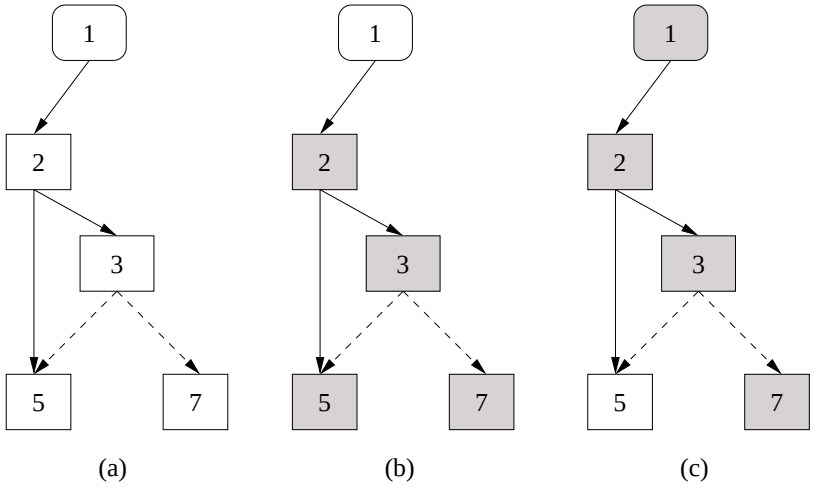

Figure 5.2: PDG and slices of the program in Figure 5.1. Solid lines represent identifier based data dependences while dashed lines represent control dependences. (a) is the PDG of the program, (b) is the forward slice based on the criteria {*line* 3}, and (c) is the backward slice based on the criteria {*line* 7}. In this example, a line in the program is considered as a program point.

For the program in Figure 5.1, identifier-based data dependence and control dependence could be used to construct a PDG as shown in Figure 5.2 (a)) with edges directed from the *dependee* program point to the *dependent* program point. Given the PDG and the slice criteria as line 3, the forward slice can be calculated (as shown in Figure 5.2 (b)) by following the edges along their direction. Given the slice criteria as line 7, the backward slice can be calculated (as shown in Figure 5.2 (c)) by following the edges in the reverse direction.

5.1.2 Inter-procedural Slicing

Accuracy

In a program composed of procedures/methods along with corresponding invocations, each procedure can be sliced according the previous technique whenever a program point enclosed by the procedure is a criteria or is included into the slice. Although simple, it can be complicated in terms of achieving accuracy when dependences vertically span procedural boundaries via parameter-argument bindings and procedure invocations in multiple calling contexts.

For example, consider the method `Acct.transfer(Acct,Acct,int)` in a sequential context. If {line 22} is the criteria, then lines 21 and 16 should be included in the slice. However, not all parameters in line 16 need to be included in the slice. Advancing the slice calculation beyond the boundary of `Acct.transfer(Acct,Acct,int)`, the first argument in line 67 need not be included. Hence, the slice calculation needs to handle the program at finer and varying level of granularity to achieve accuracy, and this is done by the proposed algorithm.

This issue was the primary concern identified in Weiser's slicing algorithm and addressed via a graph based inter-procedural slicing algorithm by Horwitz et al. [HRB90]. The solution was based on a dependence graph (*system dependence graph (SDG)*) composed of dependence graphs of

```
1   class Acct {
2     protected int balance;
3     Acct() {
4       balance = 100;
5     }
6     synchronized void deposit(int amt) {
7       int temp = balance;
8       balance = temp + amt;
9
10    }
11    synchronized void withdraw(int amt) {
12      int temp = balance;
13      balance = temp - amt;
14    }
15    static void transfer(Acct src,
16            Acct dest, int amt) {
17      synchronized(src) {
18        synchronized(dest) {
19          int temp = src;
20          src = temp - amt;
21          temp = dest;
22          dest = temp + amt;
23        }
24      }
25    }
26  } // End of class Acct
27
28  public class Home {
29    public static void main(String[] s) {
30      Acct savings = new Acct();
31      Thread wife, husband;
32      wife = new Wife(savings);
33      husband = new Husband(savings);
34      wife.start();
35      husband.start();
36    }
37  } // End of class Home
38  class Husband extends Thread {
39    protected Acct savings;
40    protected Acct checking;
41    Husband(Acct act) {
42      this.savings = act;
43      checking = new Acct();
44    }
45    public void run() {
46      checking.deposit(5);
47      savings.deposit(20);
48      synchronized(savings) {
49        savings.notify();
50      }
51      savings.withdraw(10);
52      checking.deposit(10);
53    }
54  } // End of class Husband
55
56  class Wife extends Thread {
57    protected Acct savings;
58    protected Acct checking;
59    Wife(Acct act) {
60      this.savings = act;
61      checking = new Acct();
62    }
63    public void run() {
64      synchronized(savings) {
65        savings.wait();
66      }
67      Acct.transfer(savings, checking, 10);
68    }
69  } // End of class Wife
```

Figure 5.3: A simple stripped-down concurrent Java program containing intra- and inter-thread and intra- and inter-procedural dependences.

various procedures extended with additional nodes at procedure boundaries to summarize the data dependences induced by the procedure between the arguments to the procedure. This extension disables the direct use of dependence graphs for purposes other the program slicing. This drawback is addressed by the proposed algorithm by relying on dependence relations and not their graph-based representation.

Calling Context Sensitivity

Further, suppose {line 52} is the slice criteria. The slice algorithm should descend into `Acct.deposit(int)` to include any parts of the method that may affect behavior of the program at line 52. As `checking.balance` is changed in `Acct.deposit(int)`, the entire body of the method is added to the slice. However, while ascending from `Acct.deposit(int)`, the slicing algorithm has two

options:

- ascend along all call paths/invocations leading to the current method/procedure or
- ascend along the call path/invocation that initially triggered the descent into the current method/procedure.

Comparing the options, the first option is clearly inaccurate as it may lead to the inclusion of unnecessary parts of the program in the slice while the second option is accurate as it will only trigger the inclusion of the necessary parts of the program in the slice. In terms of cost, the second option seems to be more expensive (both in terms of time and space) than the first option as the algorithm has to record, process, and forget calling contexts. However, this conjecture may be false as the second option processes smaller parts of the program [Kri02].

This issue pertaining to calling context sensitive inter-procedural slicing was first identified and addressed via a graph based solution by Reps [RR95]. The issue was later revisited by Krinke [Kri02]. However, neither of these efforts addressed calling context sensitivity in the presence of aliasing.

Aliasing

Given the slice criteria {line 51}, a calling context sensitive slicing algorithm will descend into the method `Acct.deposit(int)`, include the entire body of the method, and ascend into the invocation site line 51. As line 12 in `Acct.deposit(int)` is included into the slice, to be correct the algorithm will need to include updates to `balance` field of the object referred to by `this`.

In a sequential context, as an effect of aliasing, the definition of `balance` at line 8 resulting from the invocation at line 47 is used at line 12. By merely relying on the intra-procedural identifier based data dependence, the slicing algorithm will fail to include line 8 in the slice. However, this can be rectified by leveraging aliasing based data dependence (ABDD) defined in Chapter 3.

Although the use of ABDD addresses the correctness issue, it introduces a crucial issue that affect the accuracy of slicing — *unavailability of valid/realizable calling context information at the destination site*. In the above example, there will be no associated calling context while processing line 8 as result of following ABDD from line 12.

Even if a calling context can be constructed, to be accurate there should be a valid inter-procedural control flow path from the destination site to the triggering site involving the associated calling contexts. In other words, there should be an execution path from the destination site to the triggering site that ascends (returns) along the destination calling context and eventually descends (calls) down the triggering calling context. This novel notion of relating calling contexts via connectivity based on valid inter-procedural control flow paths is referred to as *control-based calling context coupling*.

In the above example, the calling context $\sigma_1 = line\ 51 \rightarrow line\ 12$ can be coupled with the calling context $\sigma_2 = line\ 47 \rightarrow line\ 8$ but not with the calling context $\sigma_3 = line\ 52 \rightarrow line\ 8$ (as it is based on an invalid control flow path).

A combination of call graph reachability and control flow graph reachability (explained later in the chapter) can be used to detect such spurious calling context couplings. This technique can be applied while calculating ABDD to achieve the same effect. However, independent of its application, this technique cannot detect a form of spurious calling contexts coupling based on data (un)coupling.

Consider the calling contexts σ_4 and σ_1 starting from line 46 and line 51, respectively. These calling contexts can be declared as being coupled as there is a valid inter-procedural control flow path from line 8 to line 12 involving these calling contexts. However, the object referred by the *receiver* at line 46 (the primary at line 8) in context σ_4 is not the same object referred by the *receiver* at line 51 (the primary at line 12). Hence, σ_4 and σ_1 cannot be coupled.

Digressing, this detail is correctly *unobserved* during (calling context insensitive) aliasing based data dependence calculation for this example as `withdraw(int)` and `deposit(int)` are invoked on `savings` (a common receiver) in `Husband.run()`.

This form of calling context coupling that requires the calling contexts to agree on a given data coupling at each call point is referred to as *data-based calling context coupling*, and there are no know techniques to calculate this information and leverage it during slicing and other program analysis.

5.1.3 Concurrency

For concurrent programs, program slicing needs to consider interference dependence and ready dependence to account for inter-thread dependences. However, issues similar to those in case of aliasing and inter-procedural slicing occur.

In particular, calling context coupling needs to be leverage to generate accurate slices. However, the calculation of coupling based on call graph and control flow graph reachability in concurrent programs proves to be harder due to interleaving of threads (dually, valid *interleaved* inter-procedural control flow paths may be marked as invalid). Although it seems that concurrency sensitive versions of call graphs can be leveraged to address the issue, there are no known accurate and efficient algorithms to calculate such call graphs.

Similar issues plague the calculation data sensitive calling context coupling as well.

As for calling context sensitive concurrent slicing algorithms, Krinke [Kri03c] and Nanda [Nan01] have independently proposed such algorithms to perform calling context sensitive slicing of concurrent programs. These algorithms are based on dependence graphs, and they have exponential time complexity.

5.1.4 Summary

Despite the various existing algorithms and techniques to perform inter-procedural sequential and/or concurrent program slicing, there is no scalable algorithm that can be tuned to address aliasing and concurrency with varying level of accuracy. The contents of this chapter try to address this void.

5.2 Inter-Procedural Slicing Algorithm

In simple words, the proposed slicing algorithm merely calculates the transitive closure of the union of a given set of dependence relations while handling vertical inter-procedural data flow (across invocation sites) differently. Although this seems similar to the algorithm proposed by Horwitz et al., the proposed algorithm is distinct as it encompasses every detail specific to slice construction as opposed to relying on special representation of dependence relations that encompass these details.

By its iterative nature and reliance on dependence relation, the algorithm is similar to Weiser's slicing algorithm.

The algorithm processes the input programs written in the CJava language introduced in the previous chapter. This restriction is merely to simplify the exposition and does not hinder the application of the algorithm to programs written in Java and other languages.

5.2.1 Premises

Various notations, representation of dependence information, domains, and functions relevant and used in the description of the algorithm are described in this section.

Notations

As mentioned earlier, program slicing is an analysis that identifies the program points of the input program that constitute a slice. In this spirit, the slice S is represented as a set of program points $e \in P$. Despite this representation, the syntactic structure of the slice can be reconstructed by leveraging the structure of the program and the position of the slice program points in the program.

The slice criteria C is a set of slice criterion $c = \langle e \rangle$ where e is a program point that occurs in the input program P.[4] $\langle \ldots \rangle \# i$ denotes the i-th element in the given tuple (with the index of the first element being 1).

Each program point in the program is unique.

Given a program point e that corresponds to a procedure invocation expression, $e{\uparrow}i$ denotes the program point of the i-th argument in the invocation expression.

As for dependences, $\langle \mu, \nu \rangle$ $(\mu \xrightarrow{d} \nu)$ shall denote a dependence in which μ is the source of the dependence and ν is the destination of the dependence. For example, in case of identifier based data dependence, the definition site shall be the dependee μ and the use site shall be the dependent ν. A dependence relation is denoted by $\xrightarrow{d}$ and a set of dependence relations is denoted by D.

Dependences

In the rest of this chapter, only the following five forms of dependences are considered.

- *Identifier Based Data Dependence (IBDD)* is the intra-procedural data dependence involving simple variable names.
- *Aliasing Based Data Dependence (ABDD)* is the inter-procedural data dependence involving array or field access expressions.
- *Interference Dependence (ID)* is the inter-procedural inter-thread data dependence involving array or field access expressions.
- *Control Dependence (CD)* is the intra-procedural control dependence.

[4] The use of a tuple as opposed to just the lone element will be evident during the discussion of calling context sensitivity in Section 5.3.

- *Ready Dependence (RD)* is the intra- and inter-procedural control dependence stemming from synchronization constructs.

I assume that at least IBDD, ABDD, and CD relations are used during slicing.

For simplicity, dependees and dependents are represented as program points. The program points usually represent expressions. This representation may seem to be insufficient in the context of some dependences. However, as explained below, this representation is indeed sufficient.

Program points represent expressions This representation suffices in case of *identifier based data dependences (IBDD), aliasing based data dependences (ABDD),* and *interference dependences (ID)* because due to the assignment constraints and array/field access constraints of CJava.

Specifically, the dependee (definition) program points correspond to defining assignment expression/statement and the dependent (use) program points correspond to the atomic expression that contains the use variable or procedure invocation expressions. For example, the program point corresponding to the assignment in line 3 in Figure 5.1 is a valid dependee program point while the program point corresponding to the expression `f` in line 5 of the same figure is a valid dependent program point. The invocation expression on line 5 is an invalid dependent program point. In case of array/field access expression such as `a.f`, the program point corresponding to `a.f` is a valid dependent program point.

The same representation suffices for *control dependences (CD)* as well except for the special handling of two corner cases: 1) the program point of an `if` statement is the program point of the controlling expression, and 2) the program point of a `synchronized` statement is the program point of the lock expression of the statement.

What about `synchronized` methods? `synchronized` methods in Java provide syntactic sugar to execute the method within a synchronized statement in which the `this` variable of the method (the `Class` object in case of static methods) is used as the lock expression.

Unlike the `synchronized` statement, there is no explicit executable program point that can be associated with such synchronization/locking operation. This can become an issue with `synchronized` methods and ready dependence (RD) based on rule 3 — program points m and n occur in different threads and m is the start of a synchronized statement on object o and n is the finish of a synchronized statement on object o. However, this is a non-issue in CJava programs due to the lack of support for `synchronized` methods (Constraint 8).

Domains and Functions

Various domains used in the description of the algorithm are introduced below.

$\mathcal{I}$ the domain of positive integers including zero.

$\mathcal{P}$ the domain of programs.

$\mathcal{E}$ the domain of program points in a given program.

$\mathcal{V}$ the domain of program variables in a given program.

$\mathcal{M}$ the domain of procedures in a given program.

$\mathcal{D}$ the domain of dependence relations.

Various functions used in the description of the algorithm are introduced below.

ArgIndex: $\mathcal{E} \times P \rightarrow \mathcal{I}_\perp$ returns the position of the given program point in the argument list of the enclosing invocation expression in the given program. If the program point does not occur in an invocation expression, then the function returns $\perp$.

CallSites: $\mathcal{M} \times \mathcal{P} \rightarrow \wp(\mathcal{E})_\perp$ returns the call sites program point at which the given procedure is invoked in the given program. If the given procedure does not occur in the given program, then the function returns $\perp$.

Callees: $\mathcal{E} \times \mathcal{P} \rightarrow \wp(\mathcal{M})$ returns the set of procedures that may be executed at the call site in the given program point in the given program. If the given program point does not occur in the given program, then the function returns $\perp$.

ContainsCallSite: $\mathcal{E} \times P \rightarrow \{true, false\}$ returns *true* if the given program point in the given program contains a call site; *false*, otherwise. This function is irreflexive, i.e. a call site program point is not considered to contain a call site.

ControlBasedDependenceRelation: $\mathcal{D} \rightarrow \{true, false\}$ returns *true* if the given dependence relation is a control based dependence (if it is either control or ready dependence); *false*, otherwise.

ControlIndependents: $\mathcal{M} \times \mathcal{P} \rightarrow \wp\mathcal{E}$ returns the program points in the given procedure in the given program that are control independent, i.e. do not participate in control dependence relation.

EnclosingCallSite: $\mathcal{E} \times P \rightarrow \mathcal{E}_\perp$ returns the program point corresponding to the immediately enclosing invocation expression that contains the given program point in the given program. If the given program point does not occur in the given program, then the function returns $\perp$.

ExitPoints: $\mathcal{M} \times \mathcal{P} \rightarrow \wp(\mathcal{E})_\perp$ returns the set of exit program points (expressions in `return` statements) in the given procedure in the given program. If the procedure has no exit points, then the function returns the set of program points at which a search algorithm on the control flow graph of the procedure backtracks. These program points are referred to as *pseudo exit points*. If the given procedure does not occur in the given program, then the function returns $\perp$.

GetLValue: $\mathcal{E} \times \mathcal{P} \rightarrow \mathcal{E}_\perp$ returns the program point corresponding to the expression occurring in the l-position of the assignment statement in which the given program point corresponds to the expression in r-position in the given program. If OccursAsRValue returns *false* for the given program point and program, then this function will return $\perp$.

IsACallSite: $\mathcal{E} \times \mathcal{P} \rightarrow \{true, false\}$ return *true* if the given program point corresponds to an invocation expression in the given program; *false*, otherwise.

IsAnArgument: $\mathcal{E} \times \mathcal{P} \rightarrow \{true, false\}$ returns *true* if the given program point corresponds to an argument expression in an invocation expression in the given program; *false*, otherwise.

IsAnExitPoint: $\mathcal{E} \times \mathcal{P} \rightarrow \{true, false\}$ returns *true* if the given program point occurs in `return` statements; *false*, otherwise.

```
SLICE(P, C, D, SEEDCRITGENERATOR, DEPHANDLER, PROCASCHANDLER, PROCDSCHANDLER)
 1  workset : a set of program point and method pairs.
 2  S : the set of program points in the slice.
 3
 4  S ← ∅
 5  processed ← ∅
 6  workset ← SEEDCRITGENERATOR(C)
 7  while workset ≠ ∅
 8  do c ← remove(workset)
 9     S ← S ∪ {c#1}
10     processed ← processed ∪ {c}
11     for each --d--> ∈ D
12     do workset ← workset ∪ DEPHANDLER(c, --d-->)\processed
13     for each c_n ∈ PROCASCHANDLER(c, P) ∪ PROCDSCHANDLER(c, P)
14     do if c_n ∉ processed
15           then workset ← workset ∪ {c_n}
16  return S
```

Figure 5.4: A parametric inter-procedural slicing algorithm. *remove* function removes and returns an element from the given workset.

OCCURRINGPROCEDURE: $\mathcal{E} \times \mathcal{P} \rightarrow \mathcal{M}$ returns the procedure in which the given program point occurs in the program.

OCCURSASRVALUE: $\mathcal{E} \times \mathcal{P} \rightarrow \{true, false\}$ returns *true* if the given program point in the given program occurs in the r-position in an assignment expression.

USESITES: $\mathcal{I} \times \mathcal{M} \rightarrow \mathcal{E}_{\perp}$ returns the program points at which the parameter at the given position (specified by an integer) in the given procedure are used. If the parameter position is invalid, then the function returns $\perp$. The zero-th parameter is the receiver (`this` variable).

USEDPARAMETERS: $\mathcal{E} \times \mathcal{P} \rightarrow \mathcal{I}$ returns the index of the parameters used in the given program point in the given program.

5.2.2 Parametric Slicing Algorithm (PSA)

The algorithm is presented in Figure 5.4. It accepts a program along with the slice criteria and the set of dependence relations to be consider during the slice construction. This initial set of slice criteria is referred to as *seed slice criteria*. In other words, the algorithm calculates the slice as fixed point of the function involving the slice criteria and the slice and starting from the seed slice criteria and an empty slice.

The slice generated by the algorithm is controlled by four parameters:

SEEDCRITGENERATOR: $\mathcal{E} \times D \times \mathcal{P} \rightarrow \wp(\mathcal{E})$ parameter function accepts a slice criteria and returns a slice criteria.[5] It functions as a transformer that can add, delete, and/or modify the input slice criteria. This parameter is only used in conjunction with the seed slice criteria.

[5]Recap: A slice criteria is a set of slice criterion.

depHandler: $\mathcal{E} \times D \rightarrow \wp(\mathcal{E})$ parameter function processes the given criterion against the given set of dependence relations and provides a new slice criteria to consider. The algorithm includes the program points in the returned criteria into the slice, if they previously did not occur in the slice.

procAscHandler: $\mathcal{E} \times \mathcal{P} \rightarrow \wp(\mathcal{E})$ parameter function identifies vertical inter-procedural dependences. More specifically, it provides the program points in the caller (calling) procedure that should be processed to capture the influence/dependence on the given program point. The program points are provided as criteria and they are included in the slice by the algorithm.

procDscHandler: $\mathcal{E} \times \mathcal{P} \rightarrow \wp(\mathcal{E})$ parameter function identifies vertical inter-procedural influences. More specifically, provides the program points in the callee (called) procedure that should be processed to capture the influence/dependence on the given program point. The program points are provided as criteria and they are included in the slice by the algorithm.

Three instances of the algorithm parametrized by various functions are provided in the following sub-sections.

Correctness Argument

Independent of the parameter functions, it is trivial to see that the algorithm will terminate on programs composed of finite number of program points.

As for the correctness of the slice, provided the last three parameter functions correctly identify the relevant dependences involving the given program point (criterion) and the dependence information is correct (sound), it is trivial to see that the algorithm identifies a correct (sound) slice for the given seed criteria and program.

Complexity Analysis

The algorithm processes each program point utmost once. During each such processing, depHandler parameter function is executed for each dependence relation and the procAscHandler and procDscHandler functions are executed once. Also, seedCritGenerator function is executed once at the beginning of the algorithm. Hence, the worst-case time complexity of the algorithm is

$$O_t(PSA) = O(SCG + |\mathcal{E}| \times ((|D| \times DH) + PAH + PDH))$$

where $|\mathcal{E}|$ is the number of program points in the input program, $|D|$ is the number of dependence relations, $SCG$is the complexity of seedCritGenerator function, $DH$is the complexity of depHandler function, $PAH$is the complexity of the procAscHandler function, and $PDH$is the complexity of the procDscHandler function.

The number of dependence relations used is usually a small constant c_1. Hence, the worst-case time complexity can be rewritten as

$$O_t(PSA) = O(SCG + |\mathcal{E}| \times ((c_1 \times DH) + PAH + PDH))$$

As for the worst-case space complexity, the algorithm stores utmost each program point in the

program. Hence, the worst case space complexity will be

$$O_s(PSA) = O(|\mathcal{E}|)$$

5.2.3 Backward Slicing Algorithm (BSA)

The parametric slicing algorithm can be customized generated backward slice by providing appropriate parameter functions. These functions are presented in Figure 5.5. The functions are explained below in isolation followed by a comprehensive explanation of the entire algorithm.

BACKWARD-SEEDCRITGENERATOR returns the seed slice criteria without any modification. Hence, this function has no influence on the correctness or complexity of the slicing algorithm.

BACKWARD-DEPHANDLER identifies the dependee program points that influence the criteria program points according to the given dependence relation and returns the dependee program points as new slice criteria.

BACKWARD-PROCASCHANDLER identifies the caller side program points that transfer the control to and influence the data at the program points in the given criteria and returns these caller side program points as new criteria to be processed. Given a program point in a procedure, the execution of the program point depends on the invocation of the procedure.[6] Similarly, if the program point is associated with a parameter variable, then the value of the parameter variable depends on the argument provided to the invocation of the procedure. These dependences are exposed by this function.

BACKWARD-PROCDSCHANDLER identifies the callee side program points that transfer the control out of the callee procedures. Given an invocation program point, the completion of execution of this program point depends on the completion of the exit program points in the called procedure. These dependences is exposed by this function.

Given a set of seed criteria in a program, the backward slicing algorithm transitively identifies the dependee program points that influence the program points in the seed criteria via the BACKWARD-DEPHANDLER function. However, the dependence of a procedure on the call site and the dependence of the parameters on the arguments at the call site are not captured by any dependence relation; hence, these dependences are explicitly identified for processing by the BACKWARD-PROCASCHANDLER.

Similarly, a variable can depend on the return value at a call site (e.g. `a` depends on the value returned by `b.foo()` in `a = b.foo()`). This dependence can be broken down into two dependences: 1) the exit points in the invoked procedure, and 2) the dependence on the the invocation of the procedure to facilitate the execution of the relevant exit points. BACKWARD-PROCDSCHANDLER identifies the first dependence for processing. The second dependence is not identified as it will be done as a result of subsequent processing of the identified exit point by BACKWARD-PROCASCHANDLER.

Correctness Argument

For a given set of dependence relations, the BACKWARD-DEPHANDLER correctly identifies all relevant program points that need to be included in the slice with respect to the input criteria and any

[6] To elaborate, if the call site was control dependent on program point e, then the the control independent program points in the callee will be control dependent on e when the callee is inlined at the call site.

```
BACKWARD-SEEDCRITGENERATOR(C,D)
1  return C

BACKWARD-DEPHANDLER(c,→d)
1  result ← ∅
2  for each ⟨μ, c⟩ ∈ →d
3  do result ← result ∪ {⟨μ⟩}
4  return result

BACKWARD-PROCASCHANDLER(c,P)
1  result ← ∅
2  e ← c#1
3  m ← OCCURRINGPROCEDURE(e, P)
4  for each v ∈ CALLSITES(m, P)
5  do for each i ∈ USEDPARAMETERS(e, P)
6       do result ← result ∪ {⟨v↑i⟩}
7       result ← result ∪ {⟨v⟩}
8       result ← result ∪ {⟨v↑0⟩} //In case of virtual method dispatch
9  return result

BACKWARD-PROCDESCHANDLER(c,P)
1  result ← ∅
2  v ← c#1
3  if CONTAINSCALLSITE(v, P)
4    then for each m ∈ CALLEES(v, P)
5           do for each r ∈ EXITPOINTS(m, P)
6                do result ← result ∪ {⟨r⟩}
7  return result
```

Figure 5.5: Backward slice generating parameters for the inter-procedural slicing algorithm in Figure 5.4.

program point included in the slice.

For the given criteria, BACKWARD-PROCASCHANDLER correctly identifies the vertical inter-procedural data dependence when parameter variables are involved in the criteria and inter-procedural control dependence for the program points occurring in the criteria. Hence, the inter-procedural dependence stemming from control and data flow into procedures included in the slice is correctly captured.

Similarly, the inter-procedural dependence of a variable on the return value of an invoked procedure is identified by BACKWARD-PROCDSCHANDLER. Hence, the inter-procedural dependence stemming from the data flow out of procedures (via `return` statements) is also correctly captured.

As for the correctness of inclusion of call sites and exit points, observe that all call sites of a procedure are considered when a program point from that procedure is processed. This observation implies that, once any program point in a procedure m is processed, every call site of m will be included in the slice. Hence, the algorithm will correctly and safely not process any call sites of m when subsequently processing a program point in m.

In short, the given algorithm correctly calculates the backward transitive closure of the given dependence relations along with inter-procedural dependences and, hence, correctly calculates the backward slice.

Accuracy The accuracy of the backward slice depends on the accuracy of the call graph information and the dependence relations used by the algorithm. Nevertheless, the algorithm will include all call sites of a procedure (in BACKWARD-PROCASCHANDLER) even when the algorithm descends into a procedure (in BACKWARD-PROCDSCHANDLER) from a particular call site. Hence, the algorithm is *calling context insensitive*.

Complexity Analysis

The worst-case time complexity can be constructed based on the time complexity of the the parameter functions. In the worst case,

BACKWARD-DEPHANDLER considers every dependence for the given program point in the program and as a program point can be dependent on every other program point in the program, $O_t(DH) = O(|\mathcal{E}|)$.

BACKWARD-PROCASCHANDLER considers every call site of the procedure m in which the given program point occurs the program. As every program point in the program can be a call site invoking m, there can be $|\mathcal{E}|$ call sites. However, as there can be only one procedure associated as the occurring procedure for a program point, $O_t(PAH) = O(c_a \times |\mathcal{E}|)$ when the maximum number of arguments to procedure is limited by a small constant c_a.

BACKWARD-PROCDSCHANDLER considers every procedure invoked at a call site and every exit point in each procedure. Assuming every call site can invoke every procedure, $O_t(PDH) = O(c_e \times |\mathcal{M}|)$ where the maximum number of exit points in any procedure is limited to a small constant c_e.

Plugging in the above complexities into the complexity formula for parametric slicing algorithm, the worst-case time complexity will be

```
BACKWARD-CONTROL-SEEDCRITGENERATOR(C,D)
 1  result ← ∅
 2  for each c ∈ C
 3  do e ← c#1
 4     for each →^d ∈ D
 5     do if CONTROLBASEDDEPENDENCERELATION(→^d)
 6           then for each ⟨μ, e⟩ ∈ →^d
 7                do result ← result ∪ {⟨e⟩}
 8     m ← OCCURRINGPROCEDURE(e, P)
 9     for each v ∈ CALLSITES(m, P)
10     do result ← result ∪ {⟨v⟩}
11  return result
```

Figure 5.6: Backward control slice generating parameters for the inter-procedural slicing algorithm in Figure 5.4.

$$
\begin{aligned}
O_t(BSA) &= O(SCG + |\mathcal{E}| \times (|D| \times DH + PAH + PDH)) \\
&= O(SCG + |\mathcal{E}| \times (c_1 \times |\mathcal{E}| + c_a \times |\mathcal{E}| + c_e \times |\mathcal{M}|)) \\
&= O(SCG + |\mathcal{E}|^2 \times (c_1 + c_a) + |\mathcal{E}| \times c_e \times |\mathcal{M}|) \\
&= O(SCG + |\mathcal{E}|^2 + |\mathcal{E}| \times c_e \times |\mathcal{M}|) \\
&= O(|\mathcal{E}|^2 + |\mathcal{E}| \times c_e \times |\mathcal{M}|) \\
&= O(|\mathcal{E}|^2)
\end{aligned}
$$

when $O_t(SCG) \leq O_t(|\mathcal{E}|^2)$ and $c_e \times |\mathcal{M}| \leq |\mathcal{E}|$.

The worst-case space complexity of the algorithm is identical to that of the parametric slicing algorithm (Section 5.2.2).

5.2.4 Backward Control Slicing Algorithm (BCSA)

A *backward control slice* is the set of program points that merely influence the control flow to the seed slice criteria. Control slices do not capture the program points that influence the value of the variables used at seed slice criteria. In terms of data slices (discussed until now), a backward control slice is the backward data slice with respect to the program points that control influence the program points in the seed slice criteria.

Given the relation between control slices and data slices, the parametric slicing algorithm can be used to calculate backward control slices by using BACKWARD-CONTROL-SEEDCRITGENERATOR function (given in Figure 5.6) and the other parameter functions from backward slicing algorithm.

For each of the seed criterion program point[7], BACKWARD-CONTROL-SEEDCRITGENERATOR determines the dependees program points based on control-based dependences (control dependence, ready dependence, and synchronization dependence) and provides them as the resulting criteria. As the control can only reach the seed criterion program points via the invocation of the enclosing

[7] I shall use the term *criterion program point* to denote the program point of a criterion.

procedure, the call sites of the enclosing procedure needs to be included as well. This is done in the second half of the generator.

Correctness Argument

Based on the relation between control slices and data slices, the correctness of the algorithm trivially follows.

Accuracy As the parameter functions from backward slicing algorithm is used as is and the generated criteria are not extended to capture any sort of context information, the algorithm is *calling context insensitive*.

Complexity Analysis

Due to the reuse of parameter functions from backward slicing algorithm, the worst-case time complexity will be

$$O_t(BCSA) = O(SCG + |\mathcal{E}|^2 \times (c_1 + c_a) + |\mathcal{E}| \times c_e \times |\mathcal{M}|)$$

In the worst case, for each seed criterion, the generator function may explore every dependence in every dependence relation. There may be $|\mathcal{E}|$ dependences for each seed criterion program point. It may also explore every program point in the program as a possible invocation site for the method containing each seed criterion program point. Hence,

$$O_t(SCG) = O(c_2 \times (c_1 \times |\mathcal{E}| + |\mathcal{E}|)) = O(|\mathcal{E}| \times (c_2 c_1 + c_2))$$

where $c_1 = |D|$ and c_2 is the size of the seed criteria.

Substituting $O_t(SCG)$ in the above formula, the worst-case time complexity will be

$$\begin{aligned} O_t(BCSA) &= O(SCG + |\mathcal{E}|^2 \times (c_1 + c_a) + |\mathcal{E}| \times c_e \times |\mathcal{M}|) \\ &= O(|\mathcal{E}| \times (c_2 c_1 + c_2) + |\mathcal{E}|^2 \times (c_1 + c_a) + |\mathcal{E}| \times c_e \times |\mathcal{M}|) \\ &= O(|\mathcal{E}|^2 \times (c_1 + c_a) + |\mathcal{E}| \times (c_2 c_1 + c_2 + c_e \times |\mathcal{M}|)) \\ &= O(|\mathcal{E}|^2) \end{aligned}$$

The worst-case space complexity of the algorithm is identical to that of the parametric slicing algorithm (Section 5.2.2).

5.2.5 Forward Slicing Algorithm (FSA)

Forward slices can be calculated as the forward transitive closure of a set of dependence relation starting from a given set of program points (the slice criteria). Hence, the parametric slicing algorithm can be customized to generate forward slices by using appropriate parameter functions.

```
FORWARD-SEEDCRITGENERATOR(C,D)
1  return C

FORWARD-DEPHANDLER(c,→^d)
1  result ← ∅
2  for each ⟨c, ν⟩ ∈ →^d
3  do result ← result ∪ {ν}
4  return result

FORWARD-PROCASCHANDLER(c,P)
1  result ← ∅
2  e ← c#1
3  if ISANEXITPOINT(e, P)
4    then m ← OCCURRINGPROCEDURE(e, P)
5         for each v ∈ CALLSITES(m, P)
6         do if OCCURSASRVALUE(v, P)
7               then result ← result ∪ {⟨GETLVALUE(v, P)⟩}return result

FORWARD-PROCDSCHANDLER(c,P)
 1  result ← ∅
 2  e ← c#i
 3  if ISANARGUMENT(e, P)
 4    then i ← ARGINDEX(e, P)
 5         v ← ENCLOSINGCALLSITE(e, P)
 6         for each m ∈ CALLEES(v, P)
 7         do for each e_u ∈ USESITES(i, m)
 8              do result ← result ∪ {⟨e_u⟩}
 9  if ISACALLSITE(e, P)
10    then for each m ∈ CALLEES(e, P)
11         do for each e_d ∈ CONTROLINDEPENDENTS(m)
12              do result ← result ∪ {⟨e_d⟩}
13  return result
```

Figure 5.7: Forward slice generating parameters for the inter-procedural slicing algorithm in Figure 5.4.

FORWARD-SEEDCRITGENERATOR is identical to BACKWARD-SEEDCRITGENERATOR. It returns seed slice criteria without any modification.

FORWARD-DEPHANDLER is similar to BACKWARD-DEPHANDLER as it identifies the dependent (as opposed to dependee) program points that depend on the input criteria program points according to the given dependence relation and returns them as the new slice criteria.

FORWARD-PROCASCHANDLER identifies the program point on the caller side of an invocation that will be affected by the data flowing out of the procedure enclosing the criterion program point. The program points are returned as the new slice criteria. In case of data emitting `return` statements, this function identifies the program point on the caller end that corresponds to the variable that receives the returned data.

FORWARD-PROCDSCHANDLER identifies the callee side program points that are affected by the flow of data into the callee procedure at the given call sites. Specifically, the program points that use the values bound to parameters of the callee procedure. It also identifies the callee side program points that are influenced by the flow of control into the callee. However, this only happens when the call site is included in the slice due to control dependence in the caller procedure.[8][9] The identified program points are returned as the new slice criteria.

The algorithm is similar to backward slicing algorithm with FORWARD-PROCASCHANDLER and FORWARD-PROCDSCHANDLER performing operations similar to BACKWARD-PROCDSCHANDLER and BACKWARD-PROCASCHANDLER.

While calculating the forward slice, an argument at an invocation site may be identified as a dependent. As the argument may influence computation within the invoked procedure, the influenced (dependent) computation should be included in the slice. This captures the data influence of the slice on parts not yet included in the slice. As none of the dependence relations capture this form of inter-procedural dependence, this dependence is identified and captured in FORWARD-PROCDSCHANDLER.

Similarly, a computation in a procedure may be included in the slice and it may influence the return value of the procedure. If this return value is assigned to a variable at the call sites, then the assigned variable should be included in the slice along with the program point influenced by the assignment to the variable. This dependence (not captured by dependence relations) is identified and captured in FORWARD-PROCASCHANDLER.

Correctness Argument

FORWARD-DEPHANDLER correctly identified the dependents for each program point included in the slice. Such dependents are subsequently included in the slice if they are not already included in the slice. Hence, the PSA algorithm with this parameter function correctly calculates forward transitive closure of given dependence relations.

The inter-procedural forward data influence due to arguments passed to procedures and included in the slice is captured by FORWARD-PROCDSCHANDLER. Similarly, the inter-procedural forward

[8] Suppose the call site is control dependent on program point e. If the callee procedure is inlined at the call site, then the control independent program points in the callee will be control dependent on e. Hence, if e is included in a forward slice, then all control independent statements in procedures invoked at call sites that are control dependent on e should also be included.

[9] Observe that the forward transitive closure of control influences rooted at control independent program points in a procedure will include every program point in the procedure. Hence, the algorithm handles inter-procedural (vertical) data influences and control influences separately.

data influence due to values returned from procedures and included in the slice is captured by FORWARD-PROCASCHANDLER.

Hence, the algorithm correctly calculates the forward transitive closure with respect to to the input criteria by combining the dependence information provided from the relations and the inter-procedural dependences.

Accuracy As in case of BSA, none of parameter functions either record or process calling context information. Hence, the algorithm and the generated slice is *calling context insensitive*.

Complexity Analysis

The worst-case time complexity can be constructed based on the time complexity of the the parameter functions. In the worst case,

FORWARD-DEPHANDLER is identical to BACKWARD-DEPHANDLER, hence, $O_t(DH) = O(|\mathcal{E}|)$.

FORWARD-PROCASCHANDLER considers every call site of the procedure m in which the given program point occurs the program. As every program point in the program can be a call site invoking m, there can be $|\mathcal{E}|$ call sites. However, as there can be only one procedure associated as the occurring procedure for a program point, $O_t(PAH) = O(|\mathcal{E}|)$.

FORWARD-PROCDSCHANDLER has two parts. The first part considers every procedure invoked at a call site and every program point at which an parameter of a procedure is used. As every program point in the program can be a call site invoking m, there can be $|\mathcal{E}|$ call sites. Assuming every call site can invoke every procedure, the complexity of the first part will be $O(|\mathcal{E}| \times |\mathcal{M}|)$. The second part again considers every procedure invoked at a call site and also considers the every control independent program point in the invoked procedures. In the worst case, the complexity of this part will be the same as the first part. Hence, $O_t(PDH) = O(2 \times |\mathcal{E}| \times |\mathcal{M}|)$.

Plugging in the above complexities into the complexity formula for parametric slicing algorithm, the worst-case time complexity will be

$$\begin{aligned} O_t(FSA) &= O(SCG + |\mathcal{E}| \times (|D| \times DH + PAH + PDH)) \\ &= O(SCG + |\mathcal{E}| \times (c_1 \times |\mathcal{E}| + |\mathcal{E}| + 2 \times |\mathcal{E}| \times |\mathcal{M}|)) \\ &= O(SCG + |\mathcal{E}|^2 \times (c_1 + 1 + 2 \times |\mathcal{M}|)) \\ &= O(|\mathcal{E}|^2) \end{aligned}$$

when $O_t(SCG) \leq O_t(|\mathcal{E}|^2)$ and $(c_1 + 1 + 2 \times |\mathcal{M}|) \leq |\mathcal{E}|$.

The worst-case space complexity of the algorithm is identical to that of the parametric slicing algorithm (Section 5.2.2).

5.3 Calling Context Sensitivity

Given a call site v that should be included in the slice along with the invoked procedures, all of the above algorithms will include the call site and the required parts of the invoked procedures. However, when the algorithms ascend from each of invoked procedure, they ascend into every call site that can invoke the invoked procedure instead of ascending only into the call site v that triggered the descent into the invoked procedure. Hence, the resulting slice will be inaccurate.

A simple solution to address this problem is to record the call sites that trigger the descent into procedures and then use this information to ascend into the descend triggering call sites. This solution was first proposed by Reps et al. [RR95] and was based on the solution to the problem of matching/balanced parentheses.

In the rest of this section, I shall extend the backward slicing algorithm to illustrate how the above algorithms can be extended to be calling context sensitive. I shall also provide the correctness argument and the complexity analysis for the calling context sensitive BSA.

5.3.1 Premises

To facilitate calling context sensitivity in the above algorithms, the slice criterion tuple will be extended with a new *call stack* element σ to record the descend triggering call sites.

Definition 26 A call context(stack) σ is a stack that stores either call site program points or program points containing call sites. An empty call stack is denoted by $\perp$. □

The usual *pop* and *push* operations are supported on stacks. In case of an empty calling context, *pop* operations returns $\perp$. Two additional operations *copyOf* that creates a copy of the given stack and *contains* that checks if the given element exists in the given stack are also supported on stacks.

In addition to the functions listed in Section 5.2.1, a new function is defined for the purpose of this algorithm.

IsIntraProcDepRelation: $\mathcal{D} \rightarrow \{true, false\}$ returns *true* if the given dependence relation is captures intra-procedural dependences; *false*, otherwise. This function will return *true* only for IBDD and CD relations.

5.3.2 Calling Context Sensitive Backward Slicing Algorithm (CCSBSA)

The algorithm is presented in Figure 5.8. Basically, it is an extension of the backward slicing algorithm presented in Section 5.2.3. The extensions occur in three parameter functions as described below.

CCSBSA-DepHanlder extends Backward-DepHandler by propagating the calling context (call stack) of the depedent program point to the dependee program point when the dependence is due to intra-procedural dependence relations, i.e. CD or IBDD. In other cases (e.g. ABDD), the calling context of the dependee program point with respect to the dependence being con-

sidered cannot be determined; hence, any empty calling context (call stack) is associated with the dependent program point when creating new slice criterion.

CCSBSA-ProcAscHandler extends Backward-ProcAscHandler to ascend only into the descend triggering call site available at the top of the call stack associated with the criterion being processed. The extension also modifies a copy of the call stack and injects it into the new slice criteria corresponding to the program points at the call site.

CCSBSA-ProcDscHandler extends Backward-ProcDscHandler to record calling context information by pushing the descend triggering call site onto the call stack associated with the new slice criterion corresponding to the callee program points. The extension avoids descending into a procedure due to recursion because multiple processing of the procedure along a calling context does not increase the accuracy of the slice. (The proposed approach of descending into procedure was chosen to simplify exposition. However, it should be possible to adapt a relatively optimal approach of ascending into procedures (Section 5.4.2).)

In operation, when a call site is encountered, the algorithm records the call site in the call stack associated with the new callee side slice criteria. This information is extended upon further descents in the callee procedure. Upon processing any intra-procedural dependence, the algorithm propagates the call stack of the dependent to the dependee as they both will execute in the same calling context. Upon ascending from a procedure, the algorithm forgets the call site at the top of the call stack to make the caller procedure the current procedure. In other words, the algorithm achieves calling context sensitivity by simulating the method execution semantics of the language.

5.3.3 Correctness Argument

Ignoring the parts of the algorithm pertaining to calling context sensitivity, the correctness argument from Section 5.2.3 holds.

As for the correctness of calling context sensitivity, it is trivial to see that the algorithm correctly records, uses, and forgets calling context information when it descends into and ascends out of procedures. Further, the algorithm also propagates the calling context information across intra-procedural dependences and correctly captures the approximation that program points related by intra-procedural dependence will be executed in the same calling context.[10]

The non-trivial parts of the correctness argument are the following:

Termination Unlike in BSA, the criterion is composed of a program point and a call stack. Although the number of program points in a program is finite, the number of call stacks in a program can be infinite (due to recursion); hence, the algorithm may not terminate. However, this is not the case as the algorithm safely avoids the processing of recursive call stacks, i.e. a call site that already occurs on the call stack is not pushed on the call stack to generate a new call stack to be associated with the newly generated criteria. Hence, the number of considered call stacks is finite. Consequently, the number of program point and call stack pairs are limited. As a result, the algorithm will terminate as it will process each program point and call stack pair utmost once.

Recursion Being calling context sensitive, *is it correct to consider limited calling contexts?* Observe that intra-procedural dependence relations are used in a calling context insensitive manner,

[10] Observe that it is possible that certain intra-procedural dependences may not hold in certain calling contexts. Such dependence relations are calling context sensitive and are prohibitive to calculate.

```
CCSBSA-DepHandler(c,→^d)
1  result ← ∅
2  for each ⟨μ, c#1⟩ ∈ →^d
3  do if IsIntraProcDepRelation(→^d)
4        then σ ← CopyOf(c#2)
5        else σ ← ⊥
6     result ← result ∪ {⟨μ, σ⟩}
7  return result

CCSBSA-ProcAscHandler(c,P)
 1  result ← ∅
 2  e ← c#1
 3  m ← OccurringProcedure(e, P)
 4  σ ← CopyOf(c#2)
 5  if σ ≠ ⊥
 6     then callsites ← {pop(v)}
 7     else callsites ← CallSites(m, P)
 8  for each v ∈ callsites
 9  do for each i ∈ UsedParameters(e, P)
10        do result ← result ∪ {⟨v↑i, σ⟩}
11        result ← result ∪ {⟨v, σ⟩}
12        result ← result ∪ {⟨v↑0, σ⟩} //In case of virtual method dispatch
13        return result

CCSBSA-ProcDescHandler(c,P)
 1  result ← ∅
 2  v ← c#1
 3  if ContainsCallSite(v, P)
 4     then σ ← CopyOf(c#2)
 5          if ¬contains(σ, v)
 6             then push(σ, v)
 7                  for each m ∈ Callees(v, P)
 8                  do for each r ∈ ExitPoints(m, P)
 9                     do result ← result ∪ {⟨r, CopyOf(σ)⟩}
10  return result
```

Figure 5.8: Calling context sensitive backward slice generating parameters for the inter-procedural slicing algorithm in Figure 5.4. The extensions to the algorithm are presented in blue. CopyOf is a simple operation that creates a copy of a given object/entity.

i.e. the algorithm does not query for dependence in a particular calling context. This implies that the slice of a procedure is independent of the number of times the procedure is analyzed in a calling context. Hence, it is correct to consider maximal non-recursive finite prefixes of recursive calling contexts.

Closure Due to calling context insensitivity in BSA, when procedure m is processed at two call sites v_1 and v_2, the slice of m generated for v_1 will include the required parts of v_2 as well. It may seem that this will not happen in CCSBSA as the program points in m will be processed in the context of v_1 and then rejected as "processed" in the context of v_2. However, this is untrue as the calling contexts associated with the program points in the criterion will force the program points in m to be processed for each m invoking call sites in the slice. Hence, the algorithm ensures closure of dependence across inter-procedural boundaries.

Collectively, these arguments imply the correctness of CCSBSA.

5.3.4 Complexity Analysis

The algorithm does not add new loops to the BSA algorithm, but it does remove one loop in CCSBSA-ProcAscHandler and modify the shape of the slice criteria that indirectly governs the external loop in Slice. The latter modification yields the following complexity formula

$$O_t(CCSBSA) = O(SCG + |\mathcal{E}| \times |\hat{\Sigma}| \times ((c_1 \times DH) + PAH + PDH))$$

where $\hat{\Sigma}$ is the domain of non-recursive finite call stacks in a program.

The former modification yields $O_t(PAH) = c_a$. Plugging in this complexity and the complexity of PDH and SCG from BSA, the worst-case time complexity of CCSBSA will be

$$\begin{aligned} O_t(CCSBSA) &= O(SCG + |\mathcal{E}| \times |\hat{\Sigma}| \times ((c_1 \times DH) + PAH + PDH)) \\ &= O(SCG + |\mathcal{E}| \times |\hat{\Sigma}| \times (c_1 \times |\mathcal{E}| + c_a + c_e \times |\mathcal{M}|)) \\ &= O(SCG + (|\mathcal{E}|^2 \times |\hat{\Sigma}| \times c_1) + (|\mathcal{E}| \times |\hat{\Sigma}| \times (c_a + c_e \times |\mathcal{M}|))) \\ &= O(|\mathcal{E}|^2 \times |\hat{\Sigma}|) \end{aligned}$$

when $O_t(SCG) \leq |\mathcal{E}|^2 \times |\hat{\Sigma}|$ and $c_a + c_e \times |\mathcal{M}| \leq |\mathcal{E}|$.

The worst-case space complexity of the algorithm also changes due to the injection of the call stack into the criterion. Each program point can be paired and with every finite call stack in the program. Hence, the worst-case complexity of the algorithm will be

$$O_s(CCSBSA) = O(|\mathcal{E}| \times |\hat{\Sigma}| \times c_l)$$

where c_l is the length of the longest maximal non-recursive finite call stack in the given program.

5.3.5 Optimization (CCSBSA+)

As most often the number of call stacks in a program can be very large, the worst-case time complexity of CCSBSA can be prohibitive. The main reasons for the prohibitive cost is the *repetitive processing of program points in root-identical calling contexts* and the *redundant processing for subsuming calling contexts* independent of the accuracy of the proposed slicing algorithm.

Redundant Processing of Program Points in TOP-identical Calling Contexts

Consider a procedure m invoked at call sites v_1 and v_2 in two calling contexts σ_1 and σ_2, respectively. Upon descending into m, the algorithm will include every exit point in m independent of the current calling context. Hence, independent of σ_1 and σ_2, the slice of m with respect to to the exit points as the slice criteria will be the identical.[11] Such calling contexts with identical top of the stack elements are referred to as *TOP-identical*.

Given a pair composed of a program point and a calling context rooted in m, it is sufficient

- to generate the slice of m based on the program point and the calling context if the program point is not already included in the slice, and
- to generate slices of m based on its parameters and the given calling context if the program point was already included in the slice.

This observation can be leveraged in the CCSBSA algorithm as described below.

1. For each procedure into which the algorithm descends, the descend triggering call sites are recorded against the procedure.
2. For each procedure, the position of the parameters of the procedure that are included in the slice are recorded against the procedure.
3. While descending into a procedure at a call site, if the exit points of the procedure is already included in the slice, then new slice criteria based on the the arguments at the call site that correspond to the recorded parameter positions of the procedure are generated.
4. When a parameter of a procedure is processed, new slice criteria based on the corresponding argument at each recorded call sites for the procedure is generated.

Worst-case Time Complexity Step 3 will increase the time complexity of CCSBSA-ProcDscHandler by a factor of a small constant c_a, the largest number of parameters accepted by any procedure in the program. Similarly, step 4 will increase the complexity of CCSBSA-ProcAscHandler by a factor of $|\mathcal{E}|$ as a procedure may be called by all program points in the program. Hence,

$$\begin{aligned} O_t(CCSBSA+) &= O(SCG + |\mathcal{E}| \times |\hat{\Sigma}| \times (c_1 \times |\mathcal{E}| + c_a \times |\mathcal{E}| + c_e \times |\mathcal{M}| \times c_a)) \\ &= O(|\mathcal{E}|^2 \times |\hat{\Sigma}|) \end{aligned}$$

[11] This is untrue if the slice information can be maintained in a calling context sensitive manner. Although this option seems lucrative in terms of accuracy, I think it is highly unlikely that this option can be leveraged until the fundamental scalability and usability issues are addressed.

This asymptotic time complexity is identical to that of CCSBSA. However, in practice, avoiding redundant processing of root-identical calling context and program point pairs will lead to a reduction in execution time.

Worst-case Space Complexity Step 1 will additively contribute $|\mathcal{M}| \times |\mathcal{E}|$ to the space complexity as each procedure may be called at every program point. Similarly, step 2 will additively contribute $|\mathcal{M}| \times c_a$. Hence, as the new terms will be smaller than the other terms, the overall asymptotic worst-case space complexity will be comparable to $O_s(CCSBSA)$.

Redundant Processing of Subsuming Calling Contexts

Assume that σ_1 is a *top prefix* of σ_2, i.e. $0 < i \leq |\sigma_1|.pop(\sigma_1) = pop(\sigma_2)$. Hence, upon ascending out of a procedure m_a into the call site v_b in procedure m_b and at the bottom of σ_1, the algorithm will consider every possible call site that invokes m_b and, consequently, every possible call site leading to the invocation of m_b. In other words, the algorithm will consider the relevant program points along every possible calling context leading to the invocation of m_b. If σ_3 is the resulting call stack by popping $|\sigma_1|$ number of elements from σ_2, then σ_3 will be considered by the algorithm as a result of processing σ_2. If a program point was processed in conjunction with σ_1 before being processed in conjunction with σ_2, then performance can be improved by not processing program points in m in the context of σ_2.

σ_1 is said to *subsume* $\sigma_2(\sigma_1 \prec \sigma_2)$ when σ_1 is a top prefix of σ_2. $\perp$ subsumes every non-empty call stack.

This observation can be leveraged in the CCSBSA algorithm as described below.

1. The call stack (calling context) in each processed criterion is recorded against the procedure containing the criterion program point. When a call stack σ_n is recorded against a procedure, then all the call stacks recorded against the procedure that are subsumed by σ_n are forgotten.
2. While descending into a procedure at a call site, if the descending call stack is subsumed by a call stack recorded against the procedure, the descent is ignored.

Worst-case Time Complexity Step 1 will increase the complexity of CCSBSA-ProcDscHandler by a factor of $\lg(|\hat{\Sigma}|)$ when the call stacks are maintained against the procedures as a inverted rooted tree. Hence,

$$\begin{aligned} O_t(CCSBSA+) &= O(SCG + |\mathcal{E}| \times |\hat{\Sigma}| \times (c_1 \times |\mathcal{E}| + c_a \times \lg(|\hat{\Sigma}|) + c_e \times |\mathcal{M}|)) \\ &= O(|\mathcal{E}|^2 \times |\hat{\Sigma}|) \end{aligned}$$

Worst-case Space Complexity Due to step 1, there will be an increase by an additive term of $|\mathcal{E}| \times |\mathcal{M}|$; however, the overall asymptotic worst-case space complexity will be comparable to $O_s(CCSBSA)$.

This optimization was also proposed by Krinke [Kri03c].

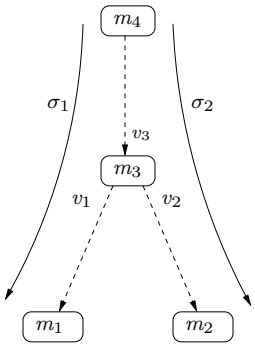

Figure 5.9: An illustration of the setting that warrants property sensitivity. The solid lines denote calling contexts (call stacks) while the dashed lines denote a call path from the methods denoted by the nodes. The call sites are positioned at suggestive locations in the call path and the calling contexts.

5.4 Property Sensitivity

The terms *control-based calling context coupling* and *data-based calling context coupling* that were mentioned in Section 5.1 will be explained in this section along with the underlying concept of *property sensitivity* and its application to improve the accuracy of program slicing.

5.4.1 Motivation

Issue

Most program analyses are realized by iteratively calculating the fixed point of a collection of constraints. This is true of most lattice-based analysis independent of the framework (i.e. constraint based, abstract interpretation, type-effect, etc) used to present the program analysis.

In such realizations, new information is calculated in each iteration based on the information calculated in the previous iterations. For example, in a points-to analysis, if the information that $x \mapsto o$ (read as "x points to o") is generated in the i-th iteration, then the information $y \mapsto o$ is generated in j-th ($j > i$) iteration upon processing the assignment of x to y. To improve accuracy, the information is maintained in a context sensitive manner. Continuing the previous example, a calling context sensitive version of the points-to analysis can maintain the information as $\sigma_1 \models x \mapsto o$ (read as "x points to o in the calling context σ_1") and, hence, calculate relatively accurate information $\sigma_2 \models y \mapsto o$ where $\sigma_1 \prec \sigma_2$ (read as σ_1 can be extended into σ_2) or $\sigma_2 \prec \sigma_1$.[12]

It is obvious that the above approach works in an intra-procedural setting. It can also be easily adapted for inter-procedural settings provided the context information can be altered (extended or protracted) and related based on the results of such alterations (as done in the previous example). However, this adaptation does not suffice for some analyses.

For example, consider the call graph in Figure 5.9. The nodes represent procedures, edges represent call paths along the direction of the arrow, and the order of procedure invocations in a

[12] Alternatively, $\sigma_1 \prec \sigma_2$ can be viewed as σ_1 being the bottom suffix of σ_2.

procedure is given by the order of the outgoing arrows from left to right. In the figure, procedure m_3 has two call sites v_1 and v_2 that lead to the invocation of m_1 and m_2 (in that order) along disjoint call paths. Further, x is assigned object o_1 in m_1 and y is assigned x in m_2. Procedure m_4 has a call site that leads to the invocation of m_3 at call site v_3. Suppose $\sigma_1 \models x \mapsto o$ and $\sigma_2 \models y \mapsto o$ when v_3 and v_1 occur in σ_1 and v_3 and v_2 occur in σ_2.

Given the above setting, an alias analysis can leverage points-to information and infer that x and y are aliases as they may refer to a common object o. However, applying the same reasoning in a slightly altered setting — *the order of procedure invocation is given by the order of the outgoing arrows from right to left* — the alias analysis will still infer that x and y are aliases as they may refer to a common object o. However, this information is inaccurate because o is assigned to x in m_1 after x is assigned to y in m_2 and $\sigma_2 \not\models y \mapsto o$ (assuming that there is no control flow path from v_1 back to v_2 in m_3).

The key observation in the example being that *points-to information is path insensitive*, i.e. the points-to analysis does not consider the effect of the control flow path on the calculation of points-to information. Although this issue can addressed by tuning the points-to analysis to be path sensitive, the cost of the resulting analysis is likely to be more expensive[13] than its calling context counterpart.[14]

Even if such a path sensitive analysis is tractable in the context of sequential programs, it would be intractable in the context of many interesting concurrent programs as the analysis will need to explore exponential number of interleaved control flow paths. Comparatively, the complexity of a path sensitive analysis would be close to that of exploring the entire state space of a program.

Solution

A cheap solution to this issue can be devised based on the following observations:

- *The calling contexts associated with the concerned alias sites have a common bottom suffix*: In the above example, the call path from m_4 to m_3 is common to both σ_1 and σ_2. This implies that it is possible (independent of intra-procedural control flow paths) for the control to reach at least one common procedure from which the aliasing occurring procedures can be reached.
- *There is an intra-procedural control flow path between the first uncommon call sites of the concerned calling contexts in the last common procedure*: The intra-procedural control flow paths between call sites v_1 and v_2 in σ_1 and σ_2 form such a path. This implies that there is a possible control flow path from m_1 to m_2 in the former setting and not in the latter setting.

Interestingly, these two observations together imply that *there is a valid/realizable inter-procedural control flow path between the concerned alias sites* — the primary property required to establish a valid aliasing relation. The first observation contributes the inter-procedural control flow evidence required to establish the relation while the second observation contributes the essential intra-procedural control flow evidence.[15]

[13] Although the cost of certain program analysis can be inversely proportional to accuracy in certain cases [Kri02], there is no general mechanism or empirical evidence to prove that path sensitive points-to analysis will be cheaper than the calling context sensitive points-to analysis.

[14] The latest and most accurate points-to analyses [WL04, Lho06] are only calling context sensitive.

[15] Interestingly, these two observation prove to be useless if considered independently. Based on the first observation alone, it is not possible to correctly infer that x and y are not aliases when procedure invocations are ordered from right to left. As for the second observation, the existence of a control flow path from m_1 to m_2 merely suggests that

Hence, instead of employing a relatively expensive path sensitive points-to analysis, a cheap solution would be to *combine* a property captured in existing calling context sensitive points-to analyses along with a property that can be trivially calculated from call graphs and control flow graphs to inexpensively infer a stronger property to aid the calculation of aliasing information that is comparable to path sensitive aliasing analysis in terms of accuracy.

The proposed property sensitivity based approach can be perceived as the generalization of *restriction/elimination based approach*[16] that can be applied to any piece of data (context, variable/data type information, etc.) used, recorded, and calculated by the analysis.

Application This optimization can be trivially applied to the calculation of ABDD, and it has been successfully applied in the calculation of ABDD in Indus project. The data pertaining to ABDD presented in Chapter 3 is indeed based on property sensitivity optimization.

In the rest of this section, I will describe control-flow based and data based instances of property sensitivity in the context of calling context sensitive backward program slicing.

5.4.2 Parametric Property Aware Calling Context Sensitive Backward Slicing Algorithm (PS-CCSBSA)

Building on the previous example, suppose the program point e_1 (source) in procedure m_1 is the slice criteria in the i-th step in CCSBSA, e_1 depends on program point e_2 (destination) in procedure m_2, and e_2 is not included in the slice.

In this situation, an accurate variant of CCSBSA needs to construct calling contexts that lead to e_2 to be paired with e_2 to generate new slicing criteria. The algorithm can pessimistically preserve all possible (including some unrealizable) calling contexts leading to e_2 by constructing slice criteria composed of e_2 and all possible calling contexts leading to m_2. This effect is achieved in CCSBSA by composing e_2 with the empty calling context $\perp$.

Instead, the algorithm can be property sensitive and construct only calling contexts that are most likely to honor the properties enabling the dependence relation between e_1 and e_2 considered by the algorithm. Such construction can be realized by incrementally constructing the calling contexts while preserving the relevant properties in each increment.

This approach can be realized by extending the CCSBSA algorithm (Figure 5.8) with PROPERTYAWARECONTEXTCONSTRUCTOR algorithm that will be used to construct property sensitive calling contexts while processing inter-procedural dependences in PS-CCSBSA-DEPHANDLER. [17]

The new auxiliary function used in the parametric extensions to CCSBSA (given in figures 5.10 and 5.11) are given below.

CALLERS: $\mathcal{M} \rightarrow \wp(\mathcal{E})$ returns the call sites at which the given procedure may be invoked. An empty set is returned if the given procedure is not invoked in the system.

x and y may be aliases provided they refer to some common object; however, it does not assert the essential validity of the control flow path in the current behavioral abstraction of the program.

[16] A good example is the use of variable type information to hone points-to information.

[17] PS-CCSBSA-DEPHANDLER algorithm requires the input program as a parameter; hence, SLICE algorithm (Figure 5.4) needs to be trivially modified to providing the input program as argument in the call to DEPHANDLER.

PS-CCSBSA-DepHandler($c, \xrightarrow{d}, P$)
1 $result \leftarrow \emptyset$
2 **for each** $\langle \mu, c\#1 \rangle \in \xrightarrow{d}$
3 **do if** IsIntraProcDepRelation($\xrightarrow{d}$)
4 **then** $\Sigma' \leftarrow \{\text{CopyOf}(c\#2)\}$
5 **else** $\Sigma' \leftarrow$ PropertyAwareCallingContextConstructor($c, \mu, \xrightarrow{d}, P$)
6 **for each** $\sigma \in \Sigma'$
7 **do** $result \leftarrow result \cup \{\langle \mu, \sigma \rangle\}$
8 **return** *result*

Figure 5.10: Property Aware extension to CCSBSA (presented in Figure 5.8). The extension to the CCSBSA algorithm is presented in blue.

Two additional operations *top* and *reverse* are supported on stacks. *top* returns the top element of the stack. If the stack is empty, it returns $\perp$. *reverse* flips the ordering of the elements in the stack.

The parametric PropertyAwareCallingContextConstructor algorithm incrementally constructs calling contexts by exploring the call graph, in reverse, starting at the destination procedure. For each explored path (calling context), the parameter AcceptContext is used to determine if the explored calling context should be accepted as a context to be paired with the destination program point. If so, the calling context is recorded and excluded from further processing. If not, the parameter ExtendContext is used to determine if the explored calling context should be further extended.

The outermost check for accepting an empty calling context handles the situation when the source procedure is reachable from the destination program point via a reachable call site in the destination procedure.

Recursion

Independent of AcceptContext parameter, the algorithm independently decides if an extensible recursive calling context should be accepted or processed.

In the call graph in Figure 5.12, infinite calling contexts can be constructed starting from any call site except v_4. However, processing these infinite calling contexts will not improve the accuracy of the slicing algorithm.

To understand, assume that AcceptContext will accept the reverse path from v_1 to v_2 but it will reject the reverse path from v_1 to v_3. Hence, the algorithm will only generate infinite number of calling contexts composed of v_1, v_2, and v_4. As multiple processing of the segment $[v_1, v_2]$ does not contribute to the slice due to calling context insensitive dependence information, it would suffice to accept the segment only once. In other words, it would suffice to accept only maximal non-recursive or utmost once-recursive paths.

Suppose the reverse path from v_1 to v_3 is accepted. The segment $[v_1, v_4]$ will be processed twice as it is included in two maximal non-recursive reverse paths starting from v_2 and v_3. Once again, such multiple processing can be avoided without loss of accuracy.

```
PropertyAwareCallingContextConstructor(c,μ,→d,P)
 1  result ← ∅
 2  if AcceptContext(⊥, c, μ, →d)
 3    then result ← {⊥}
 4    else workset ← ∅
 5         for each v ∈ Callers(OccurringProcedure(c#1, P))
 6         do σ ← ⊥
 7            push(σ, v)
 8            workset ← workset ∪ {σ}
 9
10         while workset ≠ ∅
11         do σ ← remove(workset)
12            if AcceptContext(σ, c, μ, →d)
13              then result ← result ∪ {σ}
14              else if ExtendContext(σ, c, μ, →d)
15                     then for each v′ ∈ Callers(OccurringProcedure(top(σ), P))
16                            do σ′ ← CopyOf(σ)
17                               if v′ ∈ σ
18                                 then push(σ′, ⊤)
19                                      reverse(σ′)
20                                      result ← result ∪ {σ′}
21                                 else push(σ′, v′)
22                                      workset ← workset ∪ {σ′}
23  return result
```

Figure 5.11: The parametric PropertyAwareContextConstructor algorithm that can be parametrized by AcceptContext and ExtendContext algorithms.

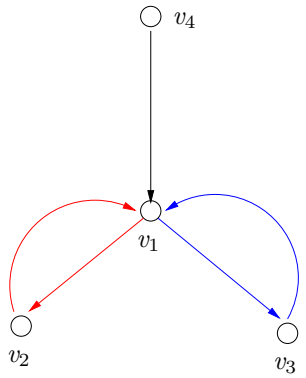

Figure 5.12: A call graph containing recursive call paths. The nodes represent call sites while the edges represent invocations leading from the source call site to the destination call site.

These optimizations can be trivially realized by processing maximal non-repetitive calling contexts. However, this realization can lead to suboptimal slicing

Suppose there was another call path from v_5 to v_1 that will be rejected by AcceptContext. Upon ascending through $[v_2, v_1]$, all call paths leading to v_1 will be considered to be included in the slice; hence, leading to suboptimal slicing. This can be fixed by identifying non-repetitive maximal extensible calling context segments of recursive calling contexts and inhibiting the further processing of such calling contexts after their bottom element has been processed. This fix is realized in the algorithm by terminating such calling contexts by a special $\top$ symbol and extending the CCSBSA-ProcAscHandler algorithm to handle this symbol appropriately.I shall refer to calling contexts with a terminal element as *terminal calling contexts*.

Note that this optimization is applicable only when AcceptContext is insensitive to the current contents of the calling context.

Correctness Argument

Like a graph search algorithm, the algorithm traces reverse paths over the call graph by exploring incoming edges on the nodes of the graph. Hence, the algorithm correctly identifies calling contexts. As the algorithm only traces finite segments of recursive paths, the algorithm will terminate; hence, the algorithm will identify finite calling contexts. The correctness of the finite calling context depends on the input parameters.

Complexity Analysis

In the worst case, the algorithm will traverse each finite call stack (context) in the call graph at least once. Hence, the worst case complexity of the algorithm will be

$$\begin{aligned} O_t(PA\text{-}CCSBSA) &= O(SCG + |\mathcal{E}| \times |\hat{\Sigma}| \times ((c_1 \times DH) + PAH + PDH)) \\ &= O(SCG + |\mathcal{E}| \times |\hat{\Sigma}| \times (c_1 \times |\mathcal{E}| \times |\hat{\Sigma}| + c_a + c_e \times |\mathcal{M}|)) \\ &= O(SCG + (|\mathcal{E}|^2 \times |\hat{\Sigma}|^2 \times c_1) + (|\mathcal{E}| \times |\hat{\Sigma}| \times (c_a + c_e \times |\mathcal{M}|))) \\ &= O(|\mathcal{E}|^2 \times |\hat{\Sigma}|^2) \end{aligned}$$

when $O_t(SCG) \leq O(|\mathcal{E}|^2 \times |\hat{\Sigma}|^2)$.

The algorithm may generate every possible finite calling contexts; hence, each program point may be paired with every possible finite calling context. As a result, the worst case space complexity would be identical to that of CCSBSA.

5.4.3 Control Flow-based Property Aware Calling Context Sensitive Backward Slicing Algorithm (C-PS-CCSBSA)

As described in the example in Section 5.4.1, inter-procedural and intra-procedural control flow reachability between two program points can be used to guide the construction of calling contexts for the purpose of slicing. The processing is sensitive to and honors a control flow property, hence,

the term *control flow-based property sensitivity*

In terms of parameterizing PS-CCSBSA to realize a calling context sensitive backward slicing algorithm, the existence/absence of the control flow reachability property can be encoded as the verdict of ExtendContext and AcceptContext parameters.

The control flow reachability property required to validate inter-thread dependences differs from that required to validate intra-thread dependences. Hence, the parameters are presented as two separate compositional instances specific to the mode of dependence relation.

Intra-thread Dependence Relation

C-AcceptContext$(\sigma, c, \mu, \xrightarrow{d})$ returns *true* if

- there exists a call path from a call site v in procedure m_t containing $v_t = top(\sigma)$ that leads to the procedure m_ν containing $\nu = c\#1$ and
- there exists an intra-procedural path from the call site v_t to v in m_t.

C-ExtendContext$(\sigma, c, \mu, \xrightarrow{d})$ returns *true* if

- v_t is not a thread creation site.

Inter-thread Dependence Relation

C-AcceptContext$(\sigma, c, \mu, \xrightarrow{d})$ returns *true* if

- the call site $v_t = top(\sigma)$ is a thread creation site and

C-ExtendContext$(\sigma, c, \mu, \xrightarrow{d})$ returns *true* if

- there are no intra-procedural paths from the call site v_t to a call site v_ν in m_t when there is a call path from v_ν to procedure m_ν.

When dealing with intra-thread dependences, the control flow path from the destination program point to the source program point needs to be confined to the same thread. In the case of inter-thread dependences, the control flow path should contain a thread creation site. This property about the control flow path is encoded in C-ExtendContext and C-AcceptContext in the context of intra- and inter-thread dependences, respectively. Observe that in the context of inter-thread dependences, this encoding proves to be suboptimal as every path from every thread creation site to the ν will be captured.

In the context of intra-thread dependences, upon processing a calling context up to the *connective* procedure that contains a call site that leads to the procedure containing the criterion program point, the slicing algorithm will consider all possible extensions of the calling context beyond the connective procedure. Hence, this property is encoded in C-AcceptContext in the context of intra-thread dependences. A calling context satisfying this property does not facilitate inter-thread control flow path. Hence, the inverse of this property is encoded in C-ExtendContext in the context of inter-thread dependences.

Static Initializers In Java, the static initializers of the classes used in the program are invoked implicitly by the virtual machine and not explicitly in the program. This disconnect between various methods can be handled either by accepting calling context when the top element occurs in a static initializer or by treating the program as a concurrent program in which each static initializer executes in a different thread.

Optimization If the calling context σ_c is associated with c, then identifying the reachability from v_t to $\nu(c)$ via a call path not terminating (subsuming) in σ_c as witness for the property implies that current dependence relation holds along an unrealizable path; hence, as an optimization, it will suffice to identify the reachability from v_t to $top(\sigma_c)\#1$.

However, as a corner case and to be correct, calling contexts in which v_t occurs in σ_c should be accepted. This captures that situation that in a procedure m along the call path to c, μ would be executed due to a call from m leading to m_μ. To achieve this accuracy, the calling contexts can be bottom extended with bottom suffix of σ_c terminating at v_t.

Correctness Argument The correctness of the C-PS-CCSBSA algorithm with the above parameter definitions follows trivially.

Complexity Analysis The parameters rely on the connectivity information of the call graph and a trivial check for the existence of thread creation site. Although a naive algorithm to calculate connectivity information is $O(\mathcal{E}^3)$, the information is calculated once and used multiple times. Hence, there will be an additive increase of $O(\mathcal{E}^3)$ in the cost of the analysis. The worst case time complexity will be $O_t(\textit{C-PS-CCSBSA}) = O(|\mathcal{E}|^2 \times |\hat{\Sigma}|^2 + \mathcal{E}^3)) = O(|\mathcal{E}|^2 \times |\hat{\Sigma}|^2)$ as most often $O(\mathcal{E}^2 \times |\hat{\Sigma}|^2) > O(\mathcal{E}^3)$. Similarly, the maintenance of the connectivity information (the only extra information needed by the algorithm) will contributed an additive cost of $O(\mathcal{E}^2)$. Hence, the worst case space complexity of C-PS-CCSBSA will be the same as that of PS-CCSBS.

Forward Slicing C-PS-CCSBSA can be converted into its forward counterpart by considering the reachability from the source (criterion/dependee) program point to the destination (dependent) program point.

Control flow-based Calling Context Coupling This notion of coupling an existing calling context (at the dependent) with another calling context paired (at the dependee) by leveraging the control flow relation between the call sites in the calling context is referred to as *control flow-based calling context coupling.* Property sensitivity is an approach to achieve such coupling.

5.4.4 Data-based Property Sensitive Calling Context Sensitive Slicing Algorithm (D-PS-CCSSA)

Issue

In Bandera [CDH+00, DHH+06], program slicing is used as a model reduction technique en route to software verification of Java programs via model checking. By backward slicing the Java program with respect to the property being verified before extracting the its model, parts of the program not relevant to the property are eliminated leading to a smaller sized model; hence, reducing the cost of model checking.

To detect the deadlocking property in programs in Bandera, the synchronization and wait/notify program points in the programs are selected as seed slice criteria. Proceeding naively, every path leading to the seed criteria will be included in the slice as relevant calling contexts are not associated with the seed criteria.

```
1  public class Top extends Thread {
2    private Object f = new Object();
3
4    public static void main(String[] s) {
5      new Top().start();    //v1
6    }
7
8    public void run() {
9      Proc p = new Proc();
10     p.bar(f);             //v2
11   }
12 }
13
14 class Proc {
15   void bar(Object n) {
16     foo(n);               //v3
17     foo(new Object());    //v4
18   }
19
20   void foo(Object o) {
21     sync(o) {             //e
22       ...
23     }
24   }
25 }
```

Figure 5.13: An example program to illustrate the benefits of D-PS-CCSBSA over C-PS-CCSBSA.

The issue can be tackled by using an approach used in C-PS-CCSBSA. However, due to the lack of a source program point, call paths from thread creation call sites to the seed criterion need to be considered as calling contexts. Applying this approach to the program in Figure 5.13, the call paths $[v_1, v_2, v_3]$ and $[v_1, v_2, v_4]$ would be considered.

Observe that the argument at callsite v_4 is non-escaping, i.e. accessed only in a single thread; hence, it cannot contribute to deadlocks involving program point e. The argument at callsite v_3 is escaping and it can possible contribute to a deadlock involving program point e. Hence, e needs to be considered as a seed criteria[18] along with one valid and another invalid calling context.

Solution

Suppose an analysis could provide escape information for the variable `o` in `foo(Object)` at each call sites leading to `foo(Object)`. Then using this information, the latter calling context can be inferred to be invalid or only the valid former calling context could be constructed. The calling contexts constructed via this approach is referred to as *escaping calling contexts* and this form of seed criteria generation is referred to as *calling context enriched seed criteria generation*.

[18]The selection of synchronization and wait/notify statements involving escaping entities is a trivial optimization in seed criteria generation for preserving deadlocking properties.

Data-based Property Sensitivity

The above approach is an example of data-based property sensitive calling context construction. As in control flow-based property sensitivity, a data-based property drives the calling context construction in *data-based property sensitive approach*.

The key component to realize data-based property sensitivity is to identify data-based property that is *trace specific*, i.e. specific to a control flow path and the sequence of value bindings of a set of variables along the control flow path. [19]

Although such properties are ideal for partitioning call paths into valid (property satisfying) and invalid (property violating) sets, they are most likely to be computationally expensive. A cheaper alternative are the calling context sensitive specific variants of such properties. For example, in the previous example, the information that `o` escapes in the calling context $[v_1, v_2, v_3]$ but not in the calling context $[v_1, v_2, v_4]$ is sufficient to facilitate the call path partitioning at the reasonable level of accuracy. An equally powerful and cheaper variant of the escaping property is its method context specific variant — the value bound to `o` in `foo(Object)` escapes at call sites v_3, v_2, and v_1 but not at v_4.

Dimensions In the above example, given a program point e, the calling context was constructed merely based on the property involving one program point, e. This approach to property sensitivity is referred to as *one-dimensional property sensitivity (1D-PS)*. If the calling context construction relied on the property involving two program points, then such property sensitivity is referred to as *two-dimensional property sensitivity (2D-PS)*. An instance of each form of data-based property sensitivity in the context of backward slicing algorithm follows.

Destination Specific Calling Context Construction (1D-PS-CCSSA)

Suppose the source program point $e_1(c)$ in procedure m_1 is the slice criteria in the i-th step in CCSBSA, e_1 depends on destination program point $e_2(\mu)$ in procedure m_2, and e_2 is not included in the slice.

Based on the nature of the dependence relation $\xrightarrow{d}$ (intra-thread vs inter-thread), either *aliasing* or *escaping* can be used the property for data-based property sensitivity. The instances of AcceptContext and ExtendContext can be defined as follows when v is the variable that causes the dependence from c to μ.

Intra-thread Dependence Relation

1D-AcceptContext$(\sigma, c, \mu, \xrightarrow{d})$ returns *true* if the $top(\sigma)$-image of v is not alias of v at μ.

1D-ExtendContext$(\sigma, c, \mu, \xrightarrow{d})$ returns *true* if the $top(\sigma)$-image of v is an alias to v at μ.

Inter-thread Dependence Relation

1D-AcceptContext$(\sigma, c, \mu, \xrightarrow{d})$ returns *true* if $top(\sigma)$ is a thread creation site.

1D-ExtendContext$(\sigma, c, \mu, \xrightarrow{d})$ returns *true* if the $top(\sigma)$-image of v is escaping (i.e. refers to escaping data).

[19] More details about trace specificity and sensitivity is given in Section 5.4.5.

Given a variable v in a procedure m_v is bound to a value l, it is possible that l is reachable in the heap via reference chains rooted at objects accessible in another procedure m_w that (directly or indirectly) invokes procedure m_v. In such cases, the reference chain leading to l in m_w is referred to as the m_w-*image* of v relative to m_v. In the above exposition, the relativeness is omitted as the image is relative to procedure m_μ.

In case of inter-thread dependences, 1D-ExtendContext is defined to include only calling contexts along which the value of interest is escaping and 1D-AcceptContext is defined to accept a calling context only upon reaching a thread creation site. Collectively, these parameters reject intra-thread call paths that lead to the destination program point.

In the intra-thread case, 1D-ExtendContext is defined to preserve alias preserving calling context and 1D-AcceptContext accepts any path that cannot be further extended to preserve aliasing. The definition of 1D-AcceptContext is weak to (correctly) allow the inclusion of control flow paths that lead to the aliasing triggering call sites. Depending on the accuracy of the analysis used to check if the property holds in a specific method, it is possible that all call paths to dependees involved in intra-thread dependence will be included in the slice. This situation can be remedied by the below proposed optimization.

Static Fields Unlike in control flow-based property sensitivity, as visible static fields (data) are accessible in every procedure, the above parameter definitions may pessimistically include all call paths leading to the dependee involving a static field. This pessimism can be partially remedied by the following optimization.

Optimization As the definitions of 1D-AcceptContext and 1D-ExtendContext are orthogonal to the definitions of 1C-AcceptContext and C-ExtendContext, these definitions can be safely combined to further improve accuracy.

Implementation In the intra-thread case, aliasing information calculated by points-to analysis [Ran02, Lho02] or by equivalence class analysis (as described in Chapter 3). In terms of realization, the property verification translates into checking if the points-to sets of the images of the variable at μ on both sides of the call site (caller and callee) have a common element. In the inter-thread case, the equivalence class based escape analysis from Chapter 3 can be leveraged to verify the property at a call site by checking if the images of the variable at μ on both sides of the call site is escaping.

If the equivalence class from Chapter 3 is used, then the property verification can be realized as a check for the existence of common entity associated with the variable images on either side of the call site. Depending on the dependence relation, either ready entities or read-write entities can be used. In other words, the property being checked would be that the references are shared according to an sharing aspect (e.g. data, lock, wait-notify).

Correctness Argument The correctness of the 1D-PS-CCSBSA algorithm with the above parameter definition follows trivially.

Complexity Analysis The time complexity of 1D-PS-CCSBSA will be the sum of the time complexity of PS-CCSBSA and the analysis used by the parameters to verify the properties. The same is true for space complexity.

```
1   class Container {
2     Object[] elements = new Object[10];
3     int count = 0;
4
5     void add(Object i) {
6       elements[count++] = i; //e_a
7     }
8
9     Object get(int i) {
10      return elements[i];   //e_g
11    }
12  }
13
14  public class Main {
15    public static void main(String[] s) {
16      english();                          //v_e
17      hindi();                            //v_h
18    }
19
20    void english() {
21    Container english = new Container(); //e_ec
22      english.add(``Hello'');             //v_ea
23      Object o = english.elements[0];     //e_ea
24    }
25
26    void hindi() {
27    Container hindi = new Container(); //e_hc
28      hindi.add(``Hello'');               //v_ha
29      Object o = hindi.elements[0];       //e_ha
30    }
31  }
```

Figure 5.14: A program to illustrate situations not handled by 1D-PS-CCSBSA.

Forward Slicing 1D-PS-CCSBSA can be converted into its forward counterpart by considering the reachability from the source (criterion/dependee) program point to the destination (dependent) program point.

Data-based Calling Context Coupling This notion of coupling an existing calling context (at the dependent) with another calling context paired (at the dependee) by leveraging the data-based relation between the call sites in the calling context is referred to as *data-based calling context coupling*. Property sensitivity is an approach to achieve such coupling.

Source and Destination Specific Calling Context Construction (2D-PS-CCSSA)

Consider the program in Figure 5.14. If line 29is the slice criterion, then the entire program except the method `Container.get(int)` and line 23 will be included in a slice generate by 1D-PS-CCSBSA. Although the invocation of `Main.english()` in `Main.main(String[])` and the method `Main.english()` do not contribute to the variable-value binding or the control flow to *line* 29, these program parts are extraneously included in the slice.

To understand the cause of inaccuracy, let us trace the algorithm. After including the seed criterion into the slice, the algorithm detects that line 6 needs to be included in the slice due to ABDD. Upon including line 6, leveraging aliasing information, the algorithm property sensitively constructs calling contexts $[v_{ea}, v_e]$ and $[v_{ha}, v_h]$ to ascend into from line 6.

Given that the algorithm selects all paths along which variable `elements` is aliased, both paths along which `elements` variable is aliased are constructed. However, the aliasing along the former calling context is invalid with respect to the aliasing at the criterion program point. In other words, the algorithm does not consider the aliasing between the variable at the source (dependent) program point and the image of the variable at the destination (dependee) program point at various points in the call path.

This situation can be remedied by checking for aliasing between the variable v_ν at the source program point, variable v_μ at the destination program point, and the m-image of v_μ in the top procedure in the calling context being constructed, i.e. check if the same value may be accessed at the three locations. In the example, this approach will detect the `english`-image of `elements` at call site v_{ea} is an an alias of `elements` at line 6 but it is not an alias of `elements` at line 29; hence, the former calling context will be discarded by the slicing algorithm and, consequently, exclude the method `english()` and its invocation in `main(String[])`.

This property sensitivity approach that leverages properties involving two program points, source and destination, is referred to as *source and destination specific property sensitivity* and it is an instance of two-dimensional property sensitivity.

Parameters The parameters to realize 2D-PS-CCSBSA are similar to those in case of 1D-PS-CCSBSA with the exception that the property check during intra-thread calling context extension will include the variable at the criterion program point. In the inter-thread case, the escaping information based calling context construction will suffice.

Intra-thread Dependence Relation

1D-AcceptContext$(\sigma, c, \mu, \xrightarrow{d})$ returns *true* if the $top(\sigma)$-image of v is not alias of v at μ.

1D-ExtendContext$(\sigma, c, \mu, \xrightarrow{d})$ returns *true* if the $top(\sigma)$-image of v is an alias to v_μ at μ and v_c at $c\#1$.

Inter-thread Dependence Relation

1D-AcceptContext$(\sigma, c, \mu, \xrightarrow{d})$ returns *true* if $top(\sigma)$ is a thread creation site.

1D-ExtendContext$(\sigma, c, \mu, \xrightarrow{d})$ returns *true* if the $top(\sigma)$-image of v has a common entity

If available, stronger sharing aspect based information can be leveraged. For example, given an interference dependence relation, the two variables and a variable's image should be shared for read-write purposes. As this sort of stronger information is provided by the equivalence class analysis described in Chapter 3, the analysis can be leveraged to realize the property as a check for a common entity (ready, lock, read-write) associated the variables and the image.

Correctness Argument The correctness argument for the algorithm follows trivially.

Complexity Analysis As in case of 1D-PS-CCSBSA, the time and space complexity of the analyses used to check the property will contributed an additive factor the complexity of the algorithm.

5.4.5 Trace Sensitivity

Suppose in the program in Figure 5.14, `Container.get(0)` was invoked at lines 29 and 6. Depending on the used alias analysis, 2D-PS-CCSBSA will most likely generate a slice that is identical to the one generated by 1D-PS-CCSBSA because `elements` at line 29 will alias along both calling contexts $[v_{ea}, v_e]$ and $[v_{ha}, v_h]$. However, the cause for aliasing differs in the calling contexts. Specifically, the aliasing along the former calling context is due to the `Container` object created at line 21 while the latter calling context is due to the `Container` object created at line 27.

In this example, if the slicing algorithm considered the image of the `element` at line 29 while establishing the property, then it will correctly identify a slice that does not include the method `english()` along with its invocation. This can be perceived as as *three dimensional data-based property sensitivity (3D-PS)*.

From the above example, property sensitivity can be perceived as sensitivity to the control flow paths and variable-to-value bindings for selected variables at various points in the control flow path.

If a *trace* is defined as *a control flow path along with the variable-to-value bindings at every program point in the control flow path*, then a *trace projection* can be defined as *a projection of a control flow path along with a consistent projection of variable-to-value bindings with respect to a set of variables at the program points in the path projection*. By considering a dependence relation as an abstraction of control flow path segments with consistent variable-to-value binding and the aliasing/points-to information as an abstraction of variable-to-value binding across a control flow path segment, the path constructed by property sensitivity can be coarsely perceived as a trace projection.[20] Hence, property sensitivity can be viewed as a general form of a new form of sensitivity — *trace sensitivity*.

While *path sensitivity* deals with a control flow path and possibly some intrinsic variable-to-value bindings required to realize the path, trace sensitivity deals with a trace/history (or its projection relative to a set of variables) along with the complete (partial, in case of a projection) variable-to-value bindings at all points in the trace. In other words, path sensitivity is a general (abstracted) form of trace sensitivity in which the data at various program points are abstracted away.

Observe that control-flow based property sensitivity based on control flow reachability is path sensitivity with respect to an abstraction of control flow paths. Such observations hint at equivalence of property sensitivity and path sensitivity; however, the existence or absence of such equivalence or the relation between property sensitivity and path sensitivity needs to be proven.

Based on the observations from property sensitive slicing, it is most likely that trace sensitive information can be further improve the accuracy of the proposed slicing algorithms. However, the two main challenges to be addressed to realize trace sensitive slicing is the complexity of bookkeeping to ensure correctness of slicing and the cost of calculating trace sensitive information required for slicing. The main challenges in reasoning about trace sensitivity is to define various properties/relations (e.g. dependence) of/in programs in terms of traces. The work presented in Chapter 2 is a stride in this direction.

[20] At a finer level, the constructed path is a splice of the trace segment projections.

5.5 Empirical Evaluation

In this section, I shall describe the experiments conducted to evaluate the resource requirement and relative accuracy of the proposed algorithms along with a discussion about the results.

5.5.1 Implementation

Each of the proposed algorithms have been implemented as part of the Indus slicing framework. The customizations have been realized as modules that can be plugged-in to realize the slicing algorithms in a manner similar to the description. The deviations and implementation details of Indus are given below.

- The program points that belong to the slice are identified by annotating the AST of the program as opposed to maintaining a slice set.
- The optimizations proposed in C-PS-CCSBSA and D-PS-CCSBSA have not been implemented.

5.5.2 Experimental Setup

As in the experiments in Section 3.7, Java Grande Benchmark programs were considered as input programs. For each program in the benchmark, 26 slices were generated using different algorithms and different configurations to preserve the deadlocking behavior of the program. These 26 slices comprised of three sets of slices.

The first set of 11 slices is used to evaluated various slicing algorithms in combination with varying levels of accuracy of interference and ready dependence information. The legends are given below.

BSA_t Backward slice with type-based interference and ready dependence information.

BSA_e Backward slice with escape-based interference and ready dependence information.

BSA Backward slice with entity-based interference and ready dependence information.

CCSBSA+ Optimized calling context sensitive backward slice with entity-based interference and ready dependence information.

CCSBSA+_t Optimized calling context sensitive backward slice with type-based interference and ready dependence information.

CCSBSA+_e Optimized calling context sensitive backward slice with escape-based interference and ready dependence information.

CCSBSA+_1D-PS Optimized calling context sensitive backward slice with entity-based interference and ready dependence information and calling context enriched seed criteria generation with the length of the seed calling context limited to 10.

2D-PSSA_4 Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and source and destination specific data-based property sensitivity with the length of property sensitive calling contexts limited to 4.

2D-PSSA_16 Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and source and destination specific data-based property sensitivity with the length of property sensitive calling contexts limited to 16.

2D-PSSA_256 Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and source and destination specific data-based property sensitivity with the length of property sensitive calling contexts limited to 256.

2D-PSSA_10000 Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and source and destination specific data-based property sensitivity with the length of property sensitive calling contexts limited to 10000.

The second set of 6 slices is used to evaluate the combination of accurate dependence calculation as described in the previous chapter along with property sensitivity in the context of calling context sensitive slicing.

1C-PSSA Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and control based property sensitivity with the length of the property specific calling context limited to 256.

1D-PSSA Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and destination specific data-based property sensitivity with the length of the property sensitive calling context limited to 256.

2D-PSSA Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and source and destination specific data-based property sensitivity with the length of the property sensitive calling context limited to 256.

t_1C-PSSA Property sensitive calling context sensitive backward slice with type-based interference and ready dependence information and control based property sensitivity with the length of the property specific calling context limited to 256.

t_1D-PSSA Property sensitive calling context sensitive backward slice with type-based interference and ready dependence information and destination specific data-based property sensitivity with the length of the property sensitive calling context limited to 256.

t_2D-PSSA Property sensitive calling context sensitive backward slice with type-based interference and ready dependence information and source and destination specific data-based property sensitivity with the length of the property sensitive calling contexts limited to 256.

The third set of 9 slices is used evaluate the combination of various optimizations to equivalence class analysis along with the proposed slicing algorithms.

BSA_sf Backward slice with entity-based interference and ready dependence and static filtering optimized equivalence class analysis.

BSA_tf Backward slice with entity-based interference and ready dependence and type filtering optimized equivalence class analysis.

BSA_both Backward slice with entity-based interference and ready dependence and both static and type filtering optimized equivalence class analysis.

CCSBSA+_sf Optimized calling context sensitive backward slice with entity-based interference and ready dependence and static filtering optimized equivalence class analysis.

CCSBSA+_tf Optimized calling context sensitive backward slice with entity-based interference and ready dependence and type filtering optimized equivalence class analysis.

CCSBSA+_both Optimized calling context sensitive backward slice with entity-based interference and ready dependence and both static and type filtering optimized equivalence class analysis.

2D-PSSA_sf Property sensitive context sensitive backward slice with entity-based interference and ready dependence, static filtering optimization for equivalence class analysis, and source and destination specific data-based property sensitivity with the length of the property sensitive calling contexts limited to 256.

2D-PSSA_tf Property sensitive context sensitive backward slice with entity-based interference and ready dependence, static filtering optimization for equivalence class analysis, and source and destination specific data-based property sensitivity with the length of the property sensitive calling contexts limited to 256.

2D-PSSA_both Property sensitive context sensitive backward slice with entity-based interference and ready dependence, static filtering optimization for equivalence class analysis, and source and destination specific data-based property sensitivity with the length of the property sensitive calling contexts limited to 256.

For each slice generation, the following data was collected.

Time Three time measures were taken in each experiment. The first was the measure of the time taken to merely identify the slice. The second was the measure of the time taken to identify the slice and inject executability into the slice. This subsumed the first measure. The third measure accounted for the time taken by other analysis other than residualization and serialization. These measures are presented as three slash separated values under the column *Time*.

Memory Two memory measures were taken in each experiment. The first was the measure of memory consumed during slice identification and the second was the measure of memory consumed during slicing and other analysis except residualization and serialization. These measures are presented as two slash separated values under the column *Memory*.

Classes The number of classes in the slice.

Methods The number of methods in the slice.

Fields The number of fields in the slice.

Stmts The number of Jimple statements in the slice.

Exprs The number of Jimple expressions in the slice.

Bytecodes The number of compressed bytecodes in the slice.

The experiments were executed on an assertion enabled JVM available as part of JDK 1.6.0_b104 with 512MB of maximum heap space on a 1.4GHz and 1GB Linux box.

The time and memory measurements were collected by instrumenting the code via AspectJ[21].

[21] http://www.eclipse.org/aspectj

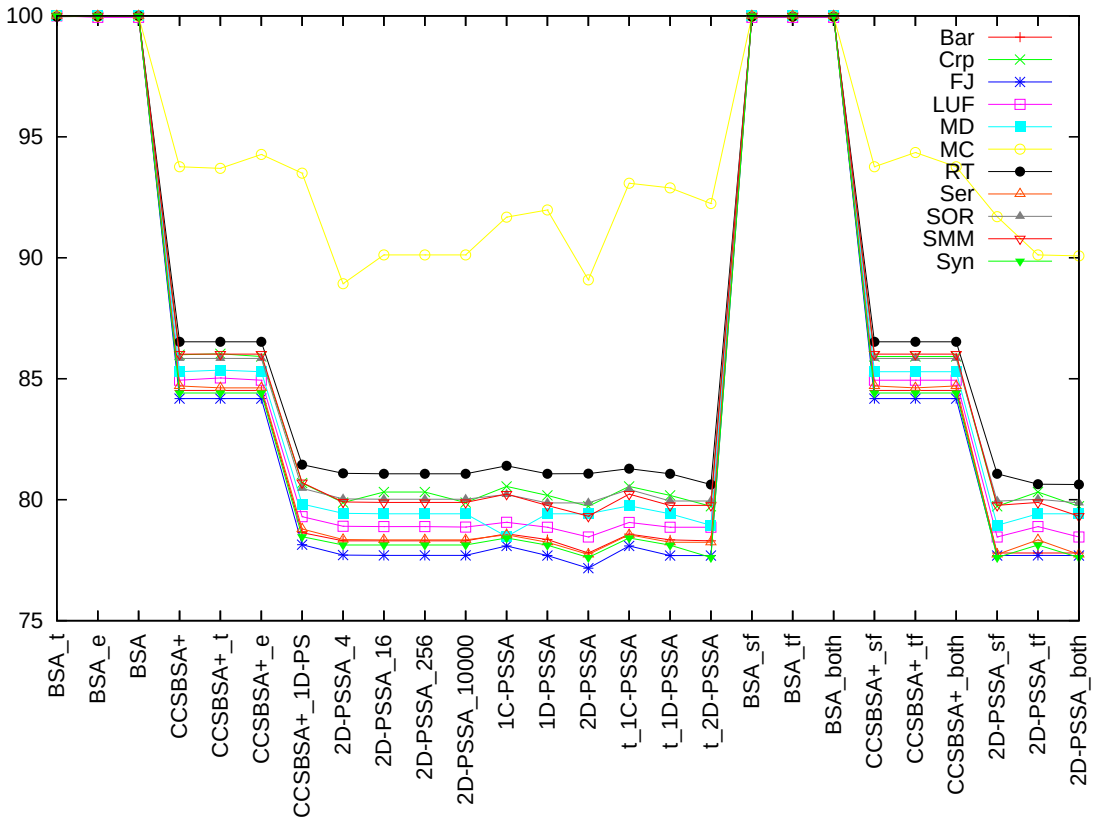

Figure 5.15: The graph of normalized slice sizes (in terms of compressed residualized bytecodes) Java Grande benchmark programs obtained by slicing via the proposed algorithms. For each program, the data is normalized w.r.t to the largest slice. Please refer to Appendix A for detailed data from the experiments.

5.5.3 Experimental Results

Each slice generation took approximate 2 minutes of wall clock time. This time included the cost parsing, object flow analysis, monitor analysis, equivalence class analysis, five dependence analysis, slice identification, injection of executability (refer to Section 5.6.3), residualization, and serialization of the slice.

The detail data for each slice generated for each benchmark program is provided in Appendix A.

The overall results from the experiments are summarized in Figure 5.15 and Table 5.1. The normalized data indicates that the algorithms behaved relatively similar in case of each program.

The following conclusions can be drawn from the data from the first set of slices.

- CCSBSA+ always performs better than BSA independent of the accuracy of the dependence relations. Hence, given a slicing algorithm, merely increasing the accuracy of the dependence information does not necessarily increase the accuracy of the generated slices. This result will most likely hold for CCSBSA.

	Bar	Crp	FJ	LUF	MD	MC	RT	Ser	SOR	SMM	Syn
BSA_t	100	100	100	100	100	100	99.97	100	100	100	100
BSA_e	100	99.98	100	99.94	100	99.98	99.98	100	100	100	100
BSA	100	99.98	100	99.94	100	99.98	100	100	100	100	100
CCSBSA+	84.52	85.99	84.18	84.94	85.29	93.76	86.53	84.7	85.84	86.02	84.41
CCSBSA+_t	84.52	86.03	84.18	85.03	85.36	93.7	86.53	84.62	85.84	86.02	84.41
CCSBSA+_e	84.52	85.92	84.18	84.94	85.29	94.27	86.53	84.62	85.84	86.02	84.41
CCSBSA+_1D-PS	78.64	80.69	78.14	79.3	79.82	93.5	81.45	78.79	80.47	80.71	78.47
2D-PSSA_4	78.3	79.88	77.71	78.9	79.44	88.93	81.09	78.35	80.03	79.91	78.13
2D-PSSA_16	78.3	80.32	77.7	78.89	79.42	90.12	81.07	78.34	80.02	79.89	78.13
2D-PSSA_256	78.3	80.32	77.7	78.89	79.42	90.12	81.07	78.34	80.02	79.89	78.13
2D-PSSA_10000	78.3	79.88	77.7	78.87	79.42	90.12	81.07	78.34	80.02	79.89	78.13
1C-PSSA	78.59	80.55	78.09	79.07	78.48	91.68	81.4	78.55	80.22	80.23	78.42
1D-PSSA	78.35	80.18	77.69	78.86	79.42	91.98	81.07	78.24	79.87	79.76	78.12
2D-PSSA	77.8	79.73	77.17	78.46	79.42	89.09	81.08	77.74	79.87	79.32	77.61
t_1C-PSSA	78.58	80.55	78.09	79.06	79.75	93.08	81.28	78.55	80.42	80.23	78.42
t_1D-PSSA	78.34	80.18	77.69	78.86	79.42	92.89	81.07	78.24	79.95	79.76	78.12
t_2D-PSSA	78.3	79.73	77.69	78.86	78.94	92.24	80.63	78.24	79.95	79.77	77.61
BSA_sf	100	99.98	100	99.94	100	99.98	99.97	100	100	100	100
BSA_tf	100	99.98	100	99.94	100	99.98	99.97	100	100	100	100
BSA_both	100	99.98	100	99.94	100	99.98	99.97	100	100	100	100
CCSBSA+_sf	84.52	85.92	84.18	84.94	85.29	93.76	86.53	84.7	85.84	86.02	84.41
CCSBSA+_tf	84.52	85.92	84.18	84.94	85.29	94.35	86.53	84.62	85.84	86.02	84.41
CCSBSA+_both	84.52	85.92	84.18	84.94	85.29	93.76	86.53	84.7	85.84	86.02	84.41
2D-PSSA_sf	77.8	79.73	77.69	78.46	78.94	91.7	81.07	77.74	79.94	79.77	77.61
2D-PSSA_tf	77.8	80.32	77.7	78.89	79.42	90.12	80.64	78.34	80.02	79.89	78.13
2D-PSSA_both	77.8	79.73	77.69	78.46	79.42	90.08	80.63	77.74	79.87	79.32	77.61

Table 5.1: The normalized slice sizes (in terms of compressed residualized bytecodes) of Java Grande benchmark programs obtained by slicing via the proposed algorithms. For each program, the data is normalized w.r.t to the largest slice. Please refer to Appendix A for detailed data from the experiments.

	BSA	CCSBSA+	PS-CCSBSA
Max Slicing Time	3.65	3.60	8.72
Min Slicing Time	1.18	1.17	1.31
Max Slicing Phase Time	20.0	19.42	24.16
Min Slicing Phase Time	16.45	16.21	14.28
Max Memory	16.95	17.24	20.97
Min Memory	10.71	9.52	9.09

Table 5.2: The maximum and minimum of normalized time and space data for various algorithms. The sets were normalized with respect to to the largest value in the set.

- PS-CCSBSA always performs better than CCSBSA+ independent of the accuracy of the dependence relations.
- CCSBSA+_1D-PS performs better than CCSBSA+. The improvement can be attributed to the injection of calling contexts into seed criterion; hence, calling contexts do matter and they can improve accuracy if introduced earlier in processing and maintained during slicing.

  However, there improvement is relatively low in case of MC. This may be due to the structure of the program.
- Interestingly, the limit on the length of the property sensitive calling context does not seem to affect the level of accuracy of PS-CCSBSA. This may be due to the shorter call graph depth in the benchmark programs. This aspect of the result requires further examination.
- In most cases, PS-CCSBSA yields up to 22% improvement over BSA and 10% improvement over CCSBSA+ in terms of the slice size.
- As the property sensitivity experiments did not leverage proposed optimizations to various forms of property sensitivity, by projection, it is most likely that optimized property sensitivity will result in increased accuracy.

The following conclusions can be drawn from the data from the second set of slices.

- Independent of the accuracy of interference and ready dependence, property sensitivity delivers the same level of accuracy. This implies that, in the context of slicing for residualization, the lack of accuracy in dependence information can be compensated via property sensitivity.

  However, in context of program understanding, it would be cost effective to have accurate dependence information instead of recovering the accuracy via post processing.
- As predicted, 2D-PS-CCSBSA performs better than 1D-PS-CCSBSA and 1D-PS-CCSBSA performs better than 1C-PS-CCSBSA.

The data from the third set of slices indicates that the ordering of slicing algorithms in terms of accuracy is independent of the sort of optimization applied to equivalence class analysis.

Beyond the quality-based results, the proposed slicing algorithms take 2-9% of the total execution time. Along with the post-processing required to inject executability into the slice, 14-24% of the total processing time and 9-20% of the total memory is used during execution.

The normalized maximum and minimum time and space numbers for BSA, CCSBSA+, and PS-CCSBSA variants (as shown in Table 5.2) indicate that the cost of slicing does increase as predicted

by the complexity analysis. However, even in the worst cases, the cost contributed by the algorithms is utmost one-fourth the total cost of calculating and generating a slice.

Based on the experimental result, there are few areas that need further empirical exploration with possibilities for new extensions and optimizations to the proposed algorithms.

5.6 Extensions

This section presents extensions to the proposed algorithms to improve scalability, accuracy, and enhance the generated slice to conform to certain properties such as executability. These extensions also illustrate the customizability of the proposed algorithm.

5.6.1 Scoping

In some applications, it is known that some parts of the program may not contribute interestingly to the slice, e.g. the classes corresponding to the AST nodes of a compiler infrastructure. In some situations, the user may want to perform incremental slices to expedite slicing of large programs by considering only parts relevant according to a particular static aspect of the program. In such cases, analyses such as slicing can be made more efficient by not considering such parts of the program. This notion of limiting the analysis to the specified scope (parts of the program) and analyzing only specific parts of the program that occur within a specified scope is referred to as *scoping*. The analysis that supports scoping is referred to as *scope sensitive analysis*.

Given a scope specification (e.g. regular expressions over on the names of various parts of the program), the proposed slicing algorithms can be adapted to perform scoped slicing by merely excluding slice criteria considered during slice construction. Specifically, by controlling the addition of the criteria to the workset at lines 6, 12, and 15 in PSA (Figure 5.4) when the corresponding program points are within the specified scope. This form of slicing is referred to as *scope sensitive slicing* or just *scoped slicing*.

This notion of restrictive slicing was introduced as *Barrier Slicing* by Krinke [Kri03b]. In contrast with my work, Krinke neither suggested nor explored the application of scoping to program analysis in general.

As for the applications of scoping, it is useful for removing parts of the runtime library that are used during application boot strapping and/or for user interface, hence, contribute unnecessarily to the time required to calculate slices and the size of the slices pertaining to core functionality of programs. Scoped slicing is also useful in checking for data confinement in the realm of security. For example, one could define a secure scope, calculate a forward slice with respect to this scope, and detect information flow based security breaches when the slice includes parts of the scope boundary.

Empirical Evaluation

To determine the effectiveness of scoped slicing, a simple experiment of slicing a non-trivial Java application was conducted. JReversePro 1.4.1[22], a Java Decompiler/Disassembler was chosen as the test input for this experiment for the following two reasons:

[22] http://jrevpro.sourceforge.net

- JReversePro consists of 90 classes, 745 methods, 786 fields, and 216KB of application class bytecodes.
- JReversePro can be used in two modes: command line mode and GUI mode, and the usage mode does not affect its core functionality.

Given the size of the application and the inherent complexity in identifying the separation of functionalities (interface v/s core) in the web of dependences encoded in the program, JReversePro was a good candidate for this experiment.

The experiment involved sequential slicing of JReversePro in four different settings: 1) *no scoping (none)*, 2) *manually scoping (manual)* by eliminating the part of the application via which the control reaches the GUI part of the application, 3) *automatic scoping during slicing (auto-slicing)* and 4) *automatic scoping during analysis and slicing (auto-analysis)*. In each of these settings, a statement that contributes to the core decompiler functionality was selected as the slice criterion.

The general scoping support provided by Indus was leveraged in this experiment. The scope was specified as a regular expression over the fully qualified names/signatures of classes, methods, and fields. The example regular expression $appl.*|Appl$ can be used to limit the analysis to consider only parts of the program belonging to the class `Appl` or the classes with fully qualified name beginning with `appl.`, i.e. belonging to package `appl`. In this experiment, a scope specification equivalent to $(sun.awt|java.awt|javax.swing|JAwtFrame|JMainFrame).*$ was used.

The data pertaining to various forms of scoping (as given in Table 5.3) indicates that, in terms of slice sizes and required time and memory to perform slicing,

- *scoped slicing performs better than no scoping.* Scoped slicing provides up to 50% reduction in terms of the slice size. This is also true in case of the required time and memory to perform slicing. However, the overall (inclusive of dependent analyses) required time and memory for slicing only reduces by utmost 10%. This can be attributed to the analysis of the parts of the application that lie outside the specified scope.
- *scoped analysis performs better than scoped slicing.* In contrast with the previous comparison, scoped analysis (and slicing) provides a lower reduction (close to 30%) over scoped slicing in terms of the slice size and the required time and memory to perform slicing. However, scoped analysis provides 70% reduction in terms of overall required time and memory to perform slicing as the dependent analyses (alike slicing) do not analyze parts of the application that lie outside the specified scope.
- *manual scoping performs slightly better than scoped analysis.* In contrast with the above comparisons, the reductions in this case is utmost 9%. Further, there is negligible or no reduction when required time and memory data is considered.

In terms of performance and relative merits, performing scoped analysis and slicing provides accuracy and scalability that is very close to the combination of performing manual scoping followed by unscoped slicing. The benefits of mere scoped slicing is strongly tied to the cost of dependent analyses. As the cost these analyses will most likely nullify the benefits of scoped slicing, the application of scoped slicing may be limited in the context of medium- and large-scale applications. Further, as scoped slicing does not depend on information pertaining to program points outside the given scope, based on the data it would be efficient to use scoped analysis and slicing together as opposed to only using scoped slicing.

Scoping	Class	Method	Fields	Size (KB)	Time (sec)	Mem (MB)
none	1198	6136	1973	972	117/539	64/607
auto-slicing	688	2881	879	485	67/462	31/568
auto-analysis	478	1902	597	334	33/142	21/164
manual	436	1856	590	318	30/129	20/150

Table 5.3: Data from generating sequential executable slices of JReversePro. The data was collected on a Linux box (2GHz/2GB) running Java 1.5.0 with maximum heap space of 1700MB. In the data of the form X/Y, X represents the data for slicing only and Y represents the (overall) data for slicing and the dependent analyses (not transformations). The classes, methods, fields, and bytecode count is inclusive of code pertaining to the application and the required libraries.

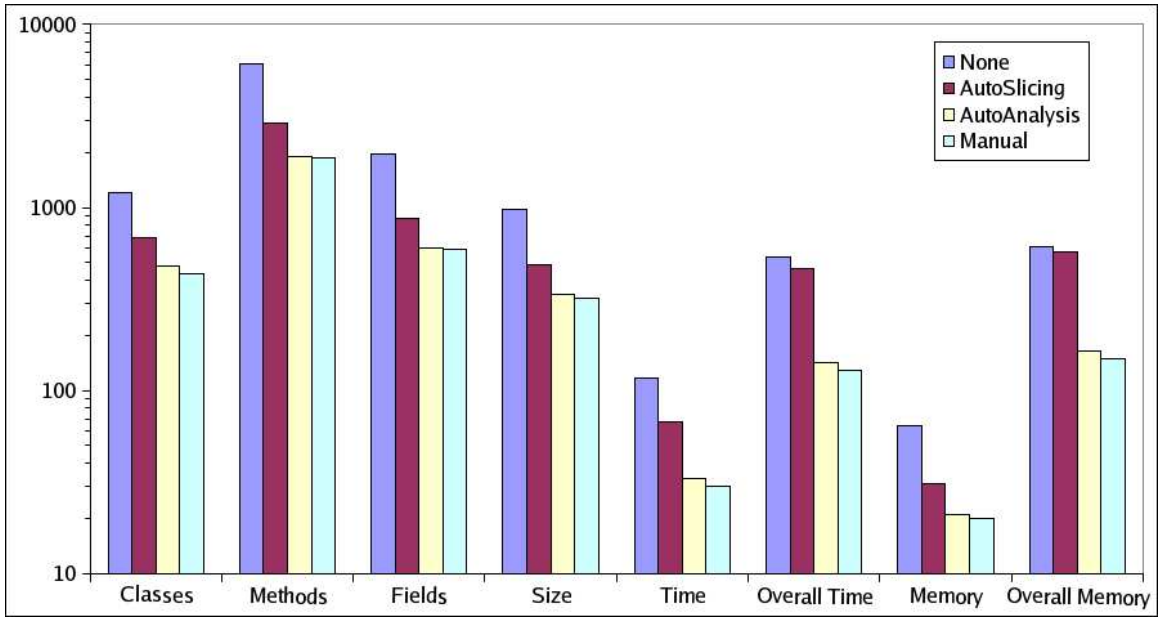

Figure 5.16: Graphical representation of the data in Table 5.3.

```
1  class CCSS {
2    public static void main(String[] s) {
3      int v = foo();
4      int u = bar();
5      int k = foo();
6    }
7
8    static int foo() {
9      return bar();
10   }
11
12   static int bar() {
13     Long l = new Long(10L);
14     return l.intValue();
15   }
16 }
```

Figure 5.17: Example to illustrate calling context restrictive slicing.

Correctness

Although scoped slicing is usually fast and cheap but it may be *unsound (incorrect)* – two program points within the scope may be related by a chain of dependences that involves program points outside the scope. However, such cases are trivially exposed by the inclusion of program points belonging to scope boundary. In such cases, assuming the accumulated cost of successively generating incrementally sound scoped slices/analysis is less than the generation of one sound unscoped slice/analysis, the user can amend the scope appropriately and incrementally obtain accurate and sound slices/analysis.

Orthogonally, tools to efficiently detect and present information about the inclusion of out-of-scope program parts in the slice/analysis may prove to be useful in the context of understanding the security aspects of a program and automatically correcting the scope to generate sound slicing/-analysis result.

5.6.2 Context Restriction

In the program in Figure 5.17, suppose a user is interested in determining the parts of the program that are affected by the invocation at line 13 as a result of its execution due to the invocation at line 3. In such cases, specifying line 14 as a criteria would result in an inaccurate slice containing lines 3, 4 and 5. The reason being that the slicing algorithm will consider every calling context leading out (into) from the method containing the slice criteria (line 13) and include every method occurring in these calling contexts in the slice.

This shortcoming can be addressed by enriching the seed slice criteria with a calling context (call stack) that can be specified by the user. For example, in the scenario described above, the call stack `[main():*, foo():3, bar():9]`[23] can be supplied as the calling context with the program point associated with the assignment expression at line 13. This restricts the slicing algorithm ascent into invocation sites (e.g. `bar():9`) to only those mentioned in the sequence from right to left. With this extension, the calling context sensitive instances of PSA can trivially leverage auxiliary contextual

[23] An element `x:y` in the calling context is to be read as line `y` in method `x`.

information to provide accurate call chain specific slices.

This form of slicing is referred to as *context restricted slicing* and it was introduced by Krinke [Kri04]. Unlike tailoring the slicing algorithm as in Krinke's approach, my approach leverages CCSBSA to realize context restriction.

This form of slicing can be useful for debugging an application based on an exception stack trace, i.e. a user would like to calculate the slice that affects only the parts of the program occurring on an exception stack trace. In security-related applications, this feature can be used to accurately identify the parts of the programs that affect the data/control flow path between two modules, hence, easily identify any insecure parts of the program.

5.6.3 Executability

In applications such program comprehension, it is not necessary for the slices to be syntactically correct with respect to the structural requirements imposed by the execution infrastructure. In other words, if class files were generated from a slice of a CJava program generated for program comprehension purpose, then it is not necessary for the class files to be verifiable according to the rules in section 4.8 and 4.0 in the JVM specification [LY99].

Such non-conformance to execution requirement will not suffice in applications such as model reduction via program slicing [CDH+00] in which the slice is executed, either directly or indirectly. This issue is also specific to backward slices.

The aspects of the slice that need to be considered to ensure executability are presented along with the aspect-specific solutions to render the slice executable.

Class Hierarchy

Although the proposed algorithms will include relevant classes to render the slice class hierarchy structurally valid, they will not explicitly include method declarations and implementations required to render the slice class hierarchy semantically valid.

To understand, suppose interface `A` contains methods `foo()` and `bar()`, abstract class `B` implements `A` and defines `foo()`, and concrete class `C` extends `B` and defines `bar()`. If `C.bar()` is included in the slice, then trivially `A`, `B`, and `A.bar()` should be included in the slice.[24] The resulting slice class hierarchy is valid in terms of its structure and semantics. However, if `A.foo()` is included in the slice due to class `D`, a sibling of class `B`, then the slice class hierarchy is semantically invalid as `C` is a concrete class that does not implement `foo()`.

The solution would be to inject a dummy implementation of `foo()` in either `B` or `C`. Optimally, such injection should occur as high as possible in the class hierarchy. As a fallout of such implementation injections, the number of methods in the slice of a program may be larger than the number of methods reachable in a program. This effect is prevalent in the empirical data presented in the previous section.

As for implementation, this transformation can be realized as an algorithm that accumulates implemented methods and injects methods required for semantic correctness by performing a bottom-up processing of the class hierarchy.

[24] Due to the semantics of invoking a method via a receiver variable of the interface type that declares the method.

```
1  while (i > 0) {
2    if (j < 5) {
3      break;
4    }
5    ...
6  }
7  System.out.println(j);
```

```
1  while (i > 0) {
2    if (j < 5) {
3
4    }
5    ...
6  }
7  System.out.println(j);
```

Figure 5.18: Example to illustrate the effect of non-inclusion of `break/continue` in backward slices.

Method Expressions

During slicing, it is possible for an invocation site to be included in the slice but some of the arguments at the site to be omitted from the slice. Before residualizing such a slice into an executable program, dummy values based on the type of the arguments at corresponding positions need to be injected into the invocation expression to render the residue program executable. Similar situation can occur in case of `return` statements with expressions and `throw` statements.

This transformation can be trivially realized by visiting the invocation expressions, *return* statements, and `throw` statements included in the slice and injecting the required dummy values.

Non-Sequential Control Flow

`break` and `continue` statements in CJava provide mechanisms to realize unstructured control flow paths. As they can be treated as restricted forms of *goto* statements, it is obvious that they do not affect any dependences. Hence, these statements will not be included in backward slices; however, such omission leads to control flow paths in the slice that did not exist in the original program and defeats the purpose of slicing.

To understand, consider the program in the left-hand column of Figure 5.18. According to the definitions in Chapter 2, Line 7 will be control dependent on lines 1 and 2. Hence, a backward slice based on the slice criteria {*line* 7} will be as shown in the right-hand column of Figure 5.18.

Interestingly, line 7 is *not* control dependent on line 2 in the slice as there is no alternative control flow path from line 2 to line 7. Hence, the slice does not preserve the execution semantics of the original program relevant to reproduce the behavior at line 7.

To avoid such unsoundness with respect to to executability, `break` and `continue` that occur along control flow paths between program points included in a slice should also be included in the slice. This transformation becomes essential while generating executable slices of programs written in languages, such as Java bytecodes, that support unstructured control flow via `goto` statements.

As for implementation, this transformation can be phrased and solved as an instance of the graph search problem.

Exit Points

While descending into a method at a call site during backward slicing, the algorithms include the exit points of the method into the slice. Hence, all paths in such methods will terminate with an exit

program point (normal or exceptional) or an infinite loop. However, in cases where the algorithms enter a method by following horizontal dependences such as ABDD, the exit points of the entered method are not included in the slice. Hence, there will be control flow paths in the slice of such methods that terminate neither with an exit program point nor an infinite loop. This violates the structural requirements of a Java class file.

This issue can be addressed by including, for each program point in the slice, an exit point that is reachable from the program point along with any unstructured *goto* statements required to facilitate such reachability. For methods that are non-terminal, the transformation should include the goto statements of the non-terminal loop that is reachable from the slice program points such that the non-terminal loop is preserved in the slice.

As for implementation, this transformation can also be phrased and solved as an instance of the graph search problem.

In terms of sequencing these transformations, as *exit points* aspect alters the slice, it would be best to perform *non-sequential control flow* specific transformation after the required exit points have been included into the slice.

5.7 Handling Exceptions

Alike Java, CJava supports exceptions. Rather than handling exceptions as an alternate control flow construct, existing literature handles exceptions as conditional constructs. With such a perception, the control flow due to exceptions and the resulting dependences can be captured as control dependences. Hence, the proposed algorithms can be used as is.

However, if extra information about the nature of the control flow is maintained along with the control dependences, then this information can be leveraged to improve the accuracy of interprocedural slicing. Specifically, if a dependent call site was reached due to a control dependence stemming from an normal (exceptional) control flow edge, then only normal (exceptional) exit points need to be considered in the invoked procedures.

5.8 Related Work

Although program slicing has been studied for over two decades, very few efforts have explored and tried to address the implications of aliasing and concurrency on efficient program slicing.

In 1998, Krinke [Kri98] first proposed the notion of interference dependence in the context of slicing concurrent programs using program dependence graphs. In the following year, Hatcliff et al. [HCD+99] identified that preservation of liveness due to synchronization was relevant in the context of concurrent programs and proposed ready dependence to capture dependences relevant to liveness preservation. Nanda et al. [NR00] identified implications of nested loops and parallelism on the accuracy of concurrent program slicing and proposed alternative algorithms. In succession, Chen et al. [CB01] proposed concurrent slicing algorithms for Java that were based on concurrent control flow graphs and concurrent program dependence graphs and sensitive to the effect of synchronization.

Subsequently, Krinke performed an empirical evaluation of calling context sensitive slicing [Kri02] and explored the implication of context sensitivity and calling contexts on the accuracy of slices [Kri04]. However, none of these efforts explored the combined effect of calling context sensitivity

and aliasing on program slicing.

The first calling context sensitive program slicing algorithm for concurrent programs was proposed by Nanda [Nan01] in 2001. This was followed by an alternative algorithm proposed by Krinke [Kri03c] in 2003. These algorithms were based on similar extensions to program dependence graphs. In contrast, the proposed algorithms are based on primitive dependence relations. Krinke's and Nanda's algorithms use an approach in which information about the last node visited in a thread is recorded and used later to determine if the next node visited in that thread should be added to the slice. As the order of node visitation is crucial, these backward slicing algorithms would approximate to the reverse simulation of the programs; hence, the algorithms would be accurate and have exponential worst-case time complexity. In comparison, as the proposed algorithms relax the order of program point visitation, they are more scalable with polynomial worst-case time complexities and they may also lead to relatively less accurate slices. However, given the absence of comparative data for these algorithms, the drop in accuracy should be considered as a mere speculation.

In terms of similarity, the optimizations proposed for CCSBSA in Section 5.3.5 can be mapped to similar optimizations (summary edges and subsumption) in Krinke's and Nanda's algorithms. Similar to our approach, in the context of slicing Java programs, Nanda explored the issues stemming from aliasing and ready dependences[25] in a concurrent setting while Hammer et al. [HS04] explored the effect of aliasing at procedural boundaries in a sequential setting.

Finally, to the best of my knowledge, the proposed algorithms are the first layered collection of algorithms that can be seamlessly extended to achieve various forms of slicing. Further, this is the first attempt to seamlessly leverage various properties (such as structure, domain/application-sensitive, etc) of the program in a compositional manner to cost efficiently improve accuracy of program slicing in both sequential and concurrent settings.

[25] Nanda refers to ready dependence as *synchronization dependence*.

Chapter 6

Partial Order Reduction

Partial Order Reduction (POR) or Partial Order Methods is an optimization technique to identify behavioral paths of a system that are identical with respect to a given property defined over the system. Following Patrice Godefroid's proposal of POR as an approach to tackle state space explosion in the context of concurrent program verification in 1995 [God95], there have been various efforts [SUL00, Sto02, FG05] in proposing alternative partial order methods based on runtime information and on high level coding (locking) patterns. However, there have been very few efforts [DHRR04] focused on leveraging program analysis information as a means of realizing partial order reduction.

As partial order methods rely on dependence relation between various actions/statements of a program, it seems natural to leverage information from static dependence analysis to calculate the partial ordering. If the cost of static dependence analysis is low in comparison with the runtime detection of dependence, then a static dependence analysis driven partial order reduction can provide interesting cost benefits in either time or space or both.

In an attempt to leverage the information calculated by the equivalence class analysis proposed in Chapter 3 beyond the realm of program slicing, I have successfully applied program dependences to calculate conditional stubborn sets to efficiently optimize state space exploration. Also, I have developed a stateful program dependence-based dynamic partial order reduction algorithm [FG05] that can handle cyclic state space.[1]

A preliminary exposition of the above contributions along with supportive empirical evidence gathered based on the implementations in Bandera, Bogor, and Indus projects are presented in this chapter.

6.1 Background

In this chapter, I use and build on the foundations laid down by Dwyer et al.[DHRR04] for presenting a partial order reduction technique for Java programs expressed as a state transition system.[2]

[1] Parts of this effort involved collaborations with John Hatcliff and Robby at the Department of Computing and Information Sciences, Kansas State University.

[2] Some of the contents of this section is an adaptation of the contents from Background section of the FMSD article from Dwyer et al.[DHRR04].

The foundation relies on the strategies used by Stoller [Sto02] to phrase Java systems as transition systems; hence, the same strategies can be used to represent CJava programs with transition systems.

Definitions A *state transition system* [EMCGP99] Σ is a quadruple (S, Ξ, S_0, L) consisting of a set of states S, a set of transitions Ξ such that $\forall \alpha \in \Xi . \alpha \subseteq S \times S$, a set of initial states S_0, and a labeling function L that maps a state s to a set of primitive propositions that are true at s.

For CJava programs, each state holds the stack frame for each thread (program counters and local variables for each stack frame), global variables (i.e. static fields), and a representation of the heap. Intuitively, each $\alpha \in \Xi$ represents a statement/step/command (e.g. execution of a bytecode) that can be taken by a particular thread $t \in T$. In general, α is defined on multiple "input states", since the transition may be carried out not only in a state s but also in another state s' that only differs from s in that it presents the result of another thread t' performing a transition on s.

For a transition $\alpha \in \Xi$, we say that α is *enabled* in a state s if there is a state s' such that $\alpha(s, s')$ holds. Otherwise, α is disable in s. Intuitively, a transition α for a thread t may be disabled if the program counter for t is not at the statement represented by α, if α represents an `synchronized` statement that is blocked waiting to acquire a lock, or if t is currently in the *wait set* of an object (i.e. t has surrendered the lock of o and put itself to sleep by calling `wait` on o). The set of transitions enabled in s is $enabled(s)$, and the set of transitions enabled in s belonging to thread t is $enabled(s, t)$. $pc(s, t)$ denotes the program counter of a thread t in a state s. $current(s, t)$ denotes the set of transitions associated with the current control point $pc(s, t)$ of thread t (this set will include $enabled(s, t)$ as well as any transitions of t at $pc(s, t)$ that may be disabled). Also, $current(s)$ represents the union of current transitions at s for all active threads.

A transition is *deterministic* if, for every state s, there is utmost one state s' such that $\alpha(s, s')$. When α is deterministic, we write $s' = \alpha(s)$ instead of $\alpha(s, s')$. Following Clarke et al. [EMCGP99], I will only consider deterministic transitions. Note that this does *not* eliminate non-determinism in a thread (e.g. as might result from abstraction) — the non-determinism is simply represented by multiple enabled transitions for a thread. For each transition α, we assume that we can determine, among other things, a unique identifier for a thread t that executes α, and the set of variables and fields that are read or written by α. A *path* π from a state s is a finite or infinite sequence such that $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_0} \ldots$ such that $s = s_0$ and $\forall i . \alpha_i(s_i) = s_{i+1}$.

Assumption Each transition of the system accesses utmost one field. This reflects the granularity of Java transitions at bytecode level.

Independence The notion of independent transition is usually expressed using an *independence relation* I between transitions. Specifically, $(\alpha, \beta) \in I$ if, *for all state* s, the execution order of α and β can be interchanged when they are both enabled in s. An independence relation is required to be symmetric and anti-reflexive and to satisfy the below given two conditions.

- $\alpha, \beta \in enabled(s) \implies \alpha \in enabled(\beta(s))$
- $\alpha, \beta \in enabled(s) \implies \alpha(\beta(s)) = \beta(\alpha(s))$

The first condition ensures that one transition does not disable the other while the second condition ensures that the execution order of the transitions does not influence the resulting state (i.e. they always lead to the same state). Figure 6.1 illustrates of the effect of independent transitions on the shape of the state space.

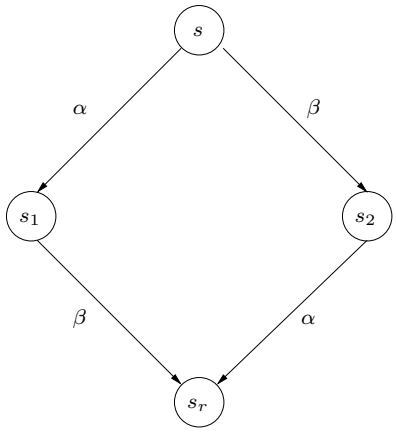

Figure 6.1: Independent transitions.

An Alternative View In the presented transition system, a transition is defined on multiple "input states" (i.e. $\alpha \subseteq S \times S$). A layered alternative yet equivalent view would be to treat $\alpha \in S \times S$ and $\Xi \subseteq S \times S$ along with a total function $L_\Xi : \alpha \rightarrow R$ that maps each transition to a command (label/statement/step) where R is the set of commands in the system. Some efforts [KP92, God95] based on this alternative view and propose the notion of conditional (in)dependence — *the command (transition) may be (in)dependent based on the state from which it is being traversed* (e.g. an field access can be independent in some states while not in other states). Due to the static nature of the approaches proposed in this chapter, the exposition in this chapter relies on the alternative view of transition systems.

Selective Search Algorithm (SSA) Given a state transition system Σ, the parametric algorithm in Figure 6.2 explores the state space of the system. The algorithm performs depth first search on the state space until it has visited every state. The parameter function SELECT influences the search by controlling/selecting a subset of transitions from the enabled set to be explored at a given state. The set of selected transitions will be referred to as *sufficient set*. The sufficient set in state s is denoted by $\mathit{sufficient}(s)$. A minimal correctness requirement of SELECT parameter is $\mathit{sufficient}(),=)\emptyset \Leftrightarrow \mathit{enabled}(),=)\emptyset$; this ensures non-existent transitions (behavior) are not introduced into the system.

The algorithm is generic as it does not handle corner cases (e.g. starvation due to successive scheduling of an infinite independent process). Instead, I assume that general techniques (e.g. bound on successive scheduling of the same thread) will be employed in the SELECT argument to handle various corner cases.

Exhaustive state space exploration can be performed by instantiating SSA with SELECT returning the enabled set $\mathit{enabled}()$. Various partial order reduction can be realized by appropriately instantiating SELECT. One such instantiation is described in the next section.

```
SELECTIVESEARCH(s_0)
 1  visited ← {s_0}
 2  stack ← ⊥
 3  push(stack, s_0)
 4  visited ← DFS(s_0, visited, stack)
 5  return

DFS(s, visited, stack)
 1  sufficient(s) ← SELECT(s)
 2  while sufficient(s) ≠ ∅
 3  do α ← remove(sufficient(s))
 4      s' ← α(s)
 5      if s' ∉ visited
 6        then visited ← visited ∪ {s'}
 7             push(stack, s')
 8             visited ← DFS(s', visited, stack)
 9             pop(stack)
10  return visited
```

Figure 6.2: The parametric selective search algorithm.

6.2 Static Program Dependence-based Conditional Stubborn Sets (SPD-CSS)

The term *dependence* was used in the context of programs and it is now being used in the context of state space exploration (program behavior). Both forms of dependence represent a similar and related aspect; hence, these forms can be leveraged across contexts. In this section, I shall describe how the static program dependence can be leveraged in the context of calculating a reduced state space that is representative of the full state space.

6.2.1 Conditional Stubborn Sets (CSS)

Given a concurrent system Σ and a selective search algorithm, Godefroid [God95] identified that sufficient sets can be calculated by leveraging persistence property of enabled transitions. A sufficient set based on persistence is referred to as a *Persistent Set*.

The notion of ***persistent sets*** is — *in a state s, the subset T of the enabled transition set E is persistent in s if every transition in $E-T$ and every transition reachable from s via transitions not in T are (conditionally) independent of the transitions in T (in the states in which they are enabled).*

In his dissertation, Godefroid described three algorithms to calculate different realizations of persistent sets based on the notions of *conflicting transitions* and *stubborn sets*. He then proposed a weaker fourth realization of persistent sets known as ***conditional stubborn sets***[3] — *in a state s, a conditional stubborn set is a non-empty subset of the enabled transitions such that each stubborn transition depends on either another stubborn transition or on a transition reachable from s through a stubborn transition.* This realization subsumed the previous three realizations, i.e. yielded

[3]This notion is similar to the notion of *detecting conditional dependences using collapses* proposed by Katz and Peled [KP92]

smaller persistent sets. Although an accurate realization, Godefroid acknowledged that it was purely theoretical due to the lack of a practical algorithm to predict the future transitions from a given state.

Godefroid also proved that the reduced state space resulting from the application of persistent sets (in all four forms) was equivalent to the full state space in terms of safety properties (i.e. safety properties are preserved in the reduced state space).

6.2.2 Transition Dependences

Program dependences introduced in previous chapters capture the relation between program points in terms of reproducing the observed behavior at program points. These program dependences partially/indirectly capture the scheduling choice that lead to the observed behavior. On the other hand, the notion of dependence introduced in this chapter captures the relation between program points that lead to scheduling choices that affect the observed behavior. For clarity, the new notion of dependence is referred to as *transition dependence.*

Given the above definitions and distinctions, in a concurrent CJava program, transition dependence can occur only in the following four cases.

- *When two threads contend to acquire the lock on a shared object.* In this situation, given the locking semantics of CJava[4], only one of the threads will arbitrarily succeed. Hence, *concurrent lock acquisition operations on a shared object are/induce transition dependent.*
- *When two threads concurrently write to the same field (cell) of a shared object (array).* Similar to the previous case, the memory model semantics of CJava[5] will impose an arbitrary ordering of the operation. Hence, *concurrent write operations to a common field (cell) of a shared object are/induce transition dependent.*
- *When a thread writes a field (cell) of a shared object (array) from which another thread concurrently reads the same field (cell).* By following a reasoning similar to the previous case, *concurrent read and write operations on a field (cell) of a shared object are/induce transition dependent.*
- *When a thread notifies the threads waiting on a lock.* Depending on if the notification operation happens before (lost notification) or after the wait operation, the blocking period of the waiting thread may change and this may affect the observed behavior of the system. Hence, *wait and notify operations on a shared object are/induce mutually transition dependent.*

6.2.3 The Approach

Given the above definitions of CSS and transition dependence, the techniques from the previous chapters can be combined to realize a simple algorithm to construct conditional stubborn sets. The realization is broken down into two phases.

[4]The locking semantics of CJava is that same as that of Java.
[5]The memory model semantics of CJava is the same as that of Java.

```
CSS-SELECT(s)
1  result ← {choose(enabled(s))}
2  for each α ∈ enabled(s)
3  do if ISINDEPENDENT(α)
4        then result ← {α}
5             return result
6        else for each β ∈ result
7             do if ISDEPENDENTON(α, β) ∨ ISDEPENDENTONSUCCESSORS(α, β)
8                   then result ← result ∪ {α}
9  return result
```

Figure 6.3: A conditional stubborn set calculating parameter of the selective search algorithm. *choose* function selects an element from the given set.

Phase 1 The transition dependence for a given CJava program can be calculated by leveraging the lock coupling (Section 3.4.2), read-write and write-write coupling (Section 3.6.3), and wait-notify coupling (Section 3.6.2) information from the equivalence class analysis. Similarly, the escape information from the same analysis can be used to statically calculate independent transitions/statements in the program. Further, by considering a call graph of the system that subsumes the call graph for each thread in the system, a may-follow relation between program points of the system can be trivially calculated by combining the call graph information with intra-procedural control flow graph information.

As all of the program analysis information is execution context independent, the information over-approximates the actual runtime coupling/relations; hence, the information can be used to construct a possibly reduced yet safe state space of the system.

Phase 2 Using the above information, a CSS calculating SELECT parameter can be instantiated as done in Figure 6.3. The helper functions used in the algorithm are defined below.

ISINDEPENDENT: $\Xi \rightarrow \{true, false\}$ This function checks if the given transition is globally independent based on static transition dependence information.

ISDEPENDENTON: $\Xi \times \Xi \rightarrow \{true, false\}$ This function checks if transitions are mutually dependent based on static transition dependence information.

ISDEPENDENTONSUCCESSORS: $\Xi \times \Xi \rightarrow \{true, false\}$ This function checks if the first transition is dependent on any of the successors of the second transition based on static transition dependence information. The successors of a transition α are transitions executed subsequently in the thread that is executing α. Given that the dependence information is static, the verdict provided by this function is not state specific.[6]

Comparatively, ISINDEPENDENT classifies a transition as independent only if the (command assocaited with the) transition is independent of all transitions in all state while I classifies a transition as independent with respect to another transition only if two transitions are *mutually independent*. Although the globalness of independence information provided by ISINDEPENDENT fails

[6]If either ISINDEPENDENT, ISDEPENDENTON, or ISDEPENDENTONSUCCESSORS uses dynamic information, then the provided dependence information would be *conditional (in)dependence*.

to capture the mutual independence, the loss of accuracy is compensated by the mutual dependence information provided IsDependentOn and IsDependentOnSuccessors functions.

As only static information about the (structure of the) program is used in this approach, (in)dependence is calculated based merely on the command (label) associated with the transition.

Correctness The correctness of the approach trivially follows from the proofs established by Godefroid and the safe over-approximation of 1) the execution time dependences by static program dependence and 2) the execution time will-follow relation by call graph based may-follow relation.

Complexity This approach will contribute additional cost for processing each state in the state space. If c_s is the cost of processing a sufficient set of size S (the worst case size of sufficient sets) and M is the number of states in the reduced state space, the $O_t(\textit{SPD-CSS}) = c_s \times M$. If c_e is the cost of processing the enabled set of size E (the worst case size of enabled sets) and N is the number of states in the full state space, then the time complexity of the naive exploration algorithm will be $O_t(\textit{SelectiveSearch}) = c_e \times N$. As $M \leq N$ and $c_s \geq c_e$, *SPD-CSS* will be effective in situations where $M \ll N$ and/or $c_s \approx c_e$. As for space complexity, it is identical to that of SSA.

6.3 Stateful Dynamic POR (SDPOR)

Assuming the entire state space for a program is available, the calculation of CSS would only consider transitions that are mutually dependent in the concrete domain of execution as opposed to considering transitions that are possibly mutually dependent in an abstract domain of analysis. Due to abstraction, the calculation for CSS based on abstraction (static) information will be sub-optimal in comparison with the calculation based on concrete (dynamic) information.

This issue was first addressed by Flanagan and Godefroid [FG05] by proposing a partial order reduction technique that was dynamic, i.e. relied on the information calculated during (as opposed to being calculated prior to) state space exploration.

The basic idea of the technique was to aggressively construct the most optimal sufficient set (a singleton) and then to expand the sufficient set as deemed necessary by the dynamically discovered dependence information. The dynamic discovery relied on the maintenance of the path from the initial state s_0 to the current state s. For each explored transition α, the dependence on previously explored transition β in the current path is calculated. Upon inferring dependence, the sufficient set at the predecessor state s_j is expanded to include the enabled transition (if any) from the thread executing α in s. The expanded transitions are processed when the exploration backtracks to s_j. Flanagan and Godefroid realized the technique in a stateless state space exploration algorithm that worked on acyclic state space.

A simple stateful counterpart to their algorithm that is based on static program dependence, is optimized based on equivalence classes, and is capable of handling cyclic state space is described in this section.

```
E-SelectiveSearch(s_0)
 1  visited ← {s_0}
 2  stack ← ⊥
 3  push(stack, s_0)
 4  visited ← E-DFS(s_0, visited, stack)
 5  return

E-DFS(s, visited, stack)
 1  sufficient(s) ← Select(s)
 2  while sufficient(s) ≠ ∅
 3  do α ← remove(sufficient(s))
 4     ExpandSufficientSet(α, s, stack)
 5     s' ← α(s)
 6     if s' ∉ visited
 7        then visited ← visited ∪ {s'}
 8             push(stack, ⟨s, sufficient(s)⟩)
 9             visited ← E-DFS(s', visited, stack)
10             pop(stack)
11  return visited
```

Figure 6.4: An extended version of the selective search algorithm (Figure 6.2). The changes to the original algorithm are presented in blue.

6.3.1 The Algorithm

An extended version of the selective search algorithm and the parameter to realize stateful dynamic POR-based selection are given in Figure 6.4 and Figure 6.5, respectively.

Implication of Statefulness In stateless state space exploration, fully explored states may be revisited as explored states are forgotten. Hence, the re-visitation will correctly force the dynamic expansion of the alias sets. However, in stateful state space exploration, every state is fully explored only once by remembering fully explored states. Hence, in a naive realization of stateful DPOR algorithm, upon arriving at a fully explored state, the mutual dependence between transitions reachable from the state and the transitions leading to the state is not considered; hence, it can lead to incorrect and over-aggressive reduction of state space.

To address this, at each explored state, the stateful DPOR algorithm remembers the transitions reachable from a state along with their executing threads. Upon arriving at a fully explored state, the algorithm considers the reachable transitions (that will not be re-explored) to discover dynamic dependences to expand sufficient sets at predecessor states; hence, the algorithm will provide correct reductions as it mimics the behavior of stateless DPOR algorithm.

This observation is encoded in line three of Figure 6.5.

Correctness Argument The correctness of the above algorithm relies on the correctness of discovering dynamic dependences involving transitions between fully explored states. As the transitions and states reachable from a fully explored state s is fixed independent of the path leading to s, it is safe and correct to consider reachable transitions to discover the dynamic dependences. Hence, the

```
DPOR-Select(s)
 1  return {choose(enabled(s))}

DPOR-ExpandSufficientSet(α, s, stack)
 1  s' ← α(s)
 2  if s' ∈ visited ∧ ⟨s', _⟩ ∉ stack
 3     then temp ← getReachableTransitions(s')
 4     else  temp ← {⟨α, s⟩}
 5  for  each ⟨β, s_i⟩ ∈ temp
 6  do for  each ⟨s_j, sufficient(s_j)⟩ ∈ stack (in top-down order)
 7       do d ← getTransitionFor(getThreadFor(β, s_i), s_j)
 8          if γ(s_j, s_t) ∧ ⟨s_t, _⟩ ∈ stack ∧ IsDependentOn(β, γ) ∧ d ≠ ∅
 9             then e ← {choose(d\sufficient(s_j))}
10                  if e ≠ ∅
11                     then sufficient(s_j) ← sufficient(s_j) ∪ e
12                          return
```

Figure 6.5: A stateful dynamic POR/CSS realizing parameter of the extended selective search algorithm. *getReachableTransitions(s)* returns the reachable transitions from the fully explored state s paired with their originating state; empty set is returned if s is not fully explored. *getThreadFor(α, s)* returns the thread executing transition α in state s and *getTransitionFor(t, s)* returns the set of enabled transitions of thread t in state s, if any.

correctness of SDPOR follows from its emulation of DPOR in a stateful setting.

Complexity Depending on the system under exploration, the worst case time complexity of SDPOR can be similar or worse than that of full state space exploration algorithm due to the additional costs of dependence discovery. In terms of space, the maintenance of reachable transition-thread pairs at each state will contribute an additive cost of $O(|S|)$; hence, the worst case space complexity of SDPOR will be more expensive that the worst case space complexiy of full state space exploration algorithm by a multiplicative factor of $|S|$.

Although worst case time/space complexity is greater than that of full state space exploration algorithm, the empirical data (presented later in the chapter) indicates that complexity in real situations are better than or comparable to that of other POR-based state space exploration algorithms.

6.3.2 Full Enabled Set Coverage (FESC)

Consider the state space in Figure 6.6 (a). There is a path from s to s_i where s_i is on the current path to s. The transitions reachable from states s_i in the shaded area and those on the current path to s will not be considered to expand the sufficient sets at states in the cycle involving s_i and s. This can lead to incorrect/unsound reductions. The issue can be addressed by ensuring that every enabled transitions at a state is reached from that state. To accomplish this, the algorithm maintains a set of non-sufficient enabled transitions at each state being explored. Upon backtracking into a state by exhausting the sufficient set, the algorithm checks if all of the non-sufficient enabled transitions were reached from the state. If not, then an arbitrary non-sufficient enabled transition is selected as a sufficient transition and the reachable state space is explored. This process is repeated till all of the enabled transitions at a state are explored. Hence, a state is marked as fully explored

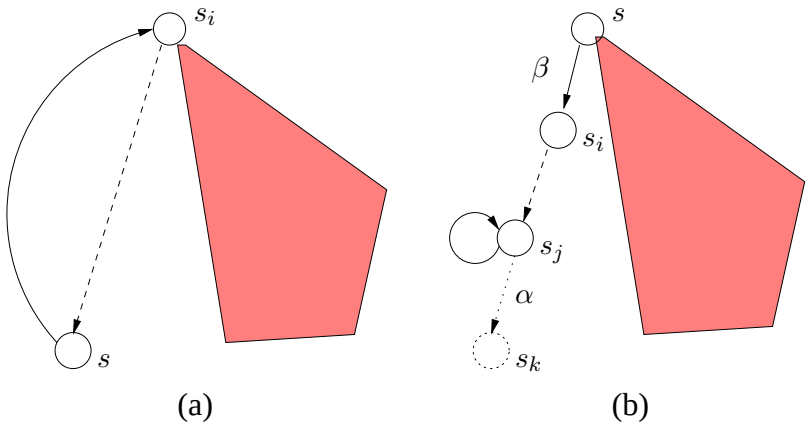

Figure 6.6: State spaces that warrant FESC corrections. The nodes represent the states, the solid edges represent transitions, the dashed edges represent the current path between the represented states, the dotted edges and nodes represent unexplored transition and states, and the shaded area represents the ignored part of the state space.

(visited) only after every enabled transition at that state is explored.

In case of cyclic state space (as illustrated in Figure 6.6 (b)), suppose α and β were dependent. If they were being executed in different threads and $\{\beta\}$ was chosen as the sufficient set at s, then this sufficient set should be expanded to explore the state space represented by the shaded area. However, due to the cycle at s_j, α is never executed and the sufficient set at s is incorrectly not expanded. A similar situation can occur in acyclic state space when all paths from a state via the sufficient transitions end in errors. This situation can be remedied by employing this technique to ensure full exploration/coverage of every enabled transition.

6.3.3 Pure Dynamic Dependences (PDD)

As the exact state of the system is available at each state, a purely dynamic approach based on the state information can be used to accurately calculate dependence. The approach would rely on comparing the appropriate parts of the transition to detect dependences.

Locking If the transitions involve lock acquisition operations, then the dependence is triggered if both transitions are attempting to acquire the lock on the same object.

Array Access If one of the transition is writing into an array cell while the other transition is reading from an array cell, then the dependence is triggered if the transitions are accessing the same cell within the same array.

Field Access If one of the transition is writing into a field while the other transition is reading from the field, then the dependence is triggered if the transitions are accessing the same field of the same object.

To enable the above detection, the algorithm maintains the variable to value binding for variables in transitions involving field and array access, and lock acquisition operations. Similar to reachable

transitions maintained to be statefully correct, the variable to value binding is maintained for each reachable transition at each fully explored state. While this provides high level accuracy, the space required to maintain variable to value binding information is proportional to the size of the state space; hence, this approach will be memory intensive in case of systems with large state space.

As for the correctness of the algorithm based on this approach, it trivially follows as all of the dependences leading to alternative behaviors (as discussed in Section 6.2.2) are preserved.

As an optimization, SDD (introduced in the next section) can be leveraged to detect dynamic dependence guided by static program dependence information and then validate the dynamic dependence based on dynamic information.

6.3.4 Pseudo Dynamic Dependences (SDD)

An alternative to PDD is to use static program dependence information. In this approach, dependence is triggered when the transitions are mutually statically dependent. As the number of transitions is finite, the space requirement to maintain dependence information for these transitions will be finite and independent of the size of the state space; hence, the approach will scale well even in the case of systems with large state space.

However, as the triggered dependences are based on static information, they may lead to false positives. Hence, the accuracy of this approach will be lower than that of the purely dynamic approach.

As for the correctness of the algorithm based on this approach, it will trivially follow provided the static program dependences are a sound abstraction of the dynamic program dependences.

6.3.5 Dependence-based Equivalence Classes (DEC)

In SDPOR, the current transition is processed along with every preceding transition in the current execution path. This is sub-optimal as it is possible that an array access operation is checked against a lock operation for dependence. As these operations are incompatible in terms of dependence, we can resort to incompatibility detection to avoid such unnecessary comparisons. Even this optimization will be sub-optimal as the the entire execution path is still traversed. To address this issue, the transitions can be partitioned into equivalence classes based on dependences they participate in and the execution path projections based on these equivalence classes can be traversed. Although this reduces the complexity by a constant factor, this optimization will reduce the overall cost in large state spaces and in case of long execution paths.

6.4 Empirical Evaluation

6.4.1 Implementation

The proposed POR techniques was realized by implementing the sufficient set selection algorithm as a scheduling strategist in Bogor[7] model checker. Additional strategists to combine the proposed techniques with other POR techniques was also implemented. The static dependence information

[7] http://bogor.projects.cis.ksu.edu

calculation has been layered on top of the equivalence class implementation available in the Indus[8] project. These implementations have been composed in Bandera[9] to realize a pipeline that accepts Java class files as inputs, performs slicing to preserve deadlocks, generates Bogor compatible model of the slice, and checks model for property violation.

6.4.2 Experimental Setup

Various examples from the Bandera project were used as test inputs to evaluate the proposed techniques. Brief descriptions of these examples are given below.

Alarm Clock (AC) A clock continually updates time and notifies clients registered for alarms. AC1, AC2, and AC3 are three variations of this example.

Bounded Buffer (BB) Two clients exchange data via add and remove operations through a bounded buffer. BB1, BB4, and BB8 are three variations of this example.

Deadlock (DL) The classic deadlock example involving two locks being obtained in different order. DL1, DL2, and DL3 are three variations of this example.

Dining Philosopers (DP) The classic problem of dining philosophers. DP1, DP2, DP3, DP4, DP5, and DP6 are six variations of this example.

Disk Scheduler (DS) A disk scheduler is shared by a few disk Readers to concurrent read various cylinders on a disk. DS1, DS2, DS4, and DS7 are four variations of this example.

Molecular Dynamic (MD) A parallelized simulation of molecular dynamics (available as part of Java Grande benchmark). MD3 is a variation of this example.

Ray Tracing (RT) A parallelized 3D ray tracing program. RT3 is a variation of this example.

Pipeline (PL) A simulation of a work pipeline in which each stage is handled by a different thread. PL1 is a variation of this example.

Producer-Consumer (PC) Producers communicate data to consumers via a common buffer. Unlike in the bounded buffer program, the data flow is unidirectional. PC3 and PC4 are two variations of this example.

Readers-Writers (RW) The classic problem of concurrent readers and writers. RW1, RW2, RW3, RW4, and RW5 are five variations of this example.

Replicated Workers (RP) A realization of the common replicated workers pattern. RP12, RP13, RP14, RP15, and RP18 are five variations of this example.

Sleeping Barbers (SB) The classic problem of sleeping barbers. SB1, SB2, and SB4 are three variations of this example.

In terms of complexity, the code base of the examples is small (in the order of few KB of application class bytecodes), but they are inherently complex as each example involves at least three concurrent threads. More than one variant is considered for all but one example.

[8] http://indus.projects.cis.ksu.edu
[9] http://bandera.projects.cis.ksu.edu

For each variant, nine experiments were conducted. These experiments were identical in terms of the input program and the generated slice and model. They differed in the kind of POR technique employed during model checking. The nine different POR configurations are given below.

E The POR technique (EPOR) that relies on dynamic detection of independent transitions based on the non-reachability of shared objects [DHRR04] was applied.

SC Conditional stubborn set (SPD-CSS) based on static program dependence information was applied.

SC+E SPD-CSS and EPOR techniques were applied in order.

SD Stateful dynamic POR technique (SDPOR) based on PDD was applied along with DEC optimizations.

E+SD EPOR and SDPOR (as in SD configuration) were applied in order.

SC+F SPD-CSS was applied with full enabled set coverage.

SC+E+F SPD-CSS and EPOR were applied in order with FESC.

SD+F SDPOR (as in SD configuration) was applied with FESC.

E+SD+F EPOR and SDPOR (as in E+SD configuration) were applied with FESC.

In all of the nine configurations, heap and process symmetry reductions [RDHI03] and collapse compression optimizations were applied along with the technique to aggregate transitions involving local variables.

All experiments were executed on the JVM available as part of Sun JDK 1.5.0_08 with 15GB of maximum heap space on multi-processor Linux boxes. As the pipeline and its components were single threaded, only one processor was used for computation. All experiments were allowed to run for up to 10 hours (36,000 seconds). Experiments that did not finish within the allotted time were terminated.

6.4.3 System Level Experimental Results

For each input program and each configuration, the raw exploration data was collected in terms (aspects) of the number of explored states *(States)*, matched states *(Matched)*, explored transitions *(Transitions)* and uncovered errors *(Errors)*. The total time and maximum memory required to execute the entire analysis and model checking pipeline *(Time)* was also recored.

The data for each aspect of each input program and configuration combination is given in the tables Table 6.1 thru Table 6.9. The data for each aspect is split into two tables: a) the data corresponding to completing executions and b) the data corresponding to terminated executions (due to time limit).

Time The data in Table 6.1 corresponds to the total time taken to execute the entire pipeline on each input program in different configurations. There were a few input programs that were terminated after 10 hours of execution, and these are not presented in the table.

Conf	SC	SD	SC+E	E+SD	E	SC+F	SD+F	SC+F+E	E+SD+F
AC1	10128	1574	7351	588	36013	36052	36065	36049	11151
AC2	684	474	442	147	476	1182	830	691	297
AC3	844	566	425	138	424	966	622	443	152
BB1	69	56	45	46	42	49	49	52	49
BB4	66	60	51	50	67	158	157	90	76
BB8	60	58	50	48	52	44	48	44	50
DL1	52	52	53	48	50	39	53	39	43
DL2	75	48	44	53	40	47	37	37	37
DL3	64	47	46	54	41	40	39	39	50
DP1	54	57	39	50	47	40	50	42	41
DP2	57	57	49	49	37	45	43	49	51
DP3	64	52	50	38	33	54	52	56	51
DP4	65	58	45	47	53	46	41	50	52
DP5	69	51	47	50	49	65	59	56	54
DP6	64	40	41	44	34	64	65	55	53
MD3	36019	36247	5469	1564	36012	36081	36288	5497	1629
RT3	36011	13314	36014	1830	24263	36034	13287	36027	1827
PL1	1446	128	714	55	119	1746	126	748	53
PC3	75	53	47	49	36	45	42	40	43
PC4	61	60	52	49	46	62	41	43	45
RW1	53	48	41	49	36	65	53	50	49
RW2	123	131	94	95	60	113	151	82	95
RW3	69	50	45	43	40	46	46	53	40
RW4	70	57	51	51	49	63	53	51	49
RW5	78	73	60	57	62	78	69	63	59
RP13	337	162	325	338	619	317	156	338	332
RP15	14357	5956	16302	2619	13515	19593	6816	18419	2654
RP18	629	377	621	478	506	628	364	640	474
SB1	63	48	41	34	64	42	34	39	33
SB4	151	201	135	130	2415	142	210	139	135

Table 6.1: Total time (in seconds) taken to execute various configuration on various input programs. All unmentioned input programs ran for the entire 10 hours.

The data indicates that the proposed approaches require time comparable to that required by EPOR-based approach. In case of long running model checks, the combination of EPOR and SDPOR based on pure dynamic approach (SD+E) affects 96-69% and 99-80% reduction in state space exploration time with and without FESC, respectively. SDPOR approach without EPOR provides comparable reductions as well. In many examples, the time taken by SPD-CSS (SC) approach is comparable to that of EPOR (E) approach.

In terms of time, in non-FESC setting, the proposed approaches are comparable or better than the most optimal EPOR approach (available in Bogor).

Space The data in Table 6.2 corresponds to the maximum memory required to execute the entire pipeline on each input program in different configurations. The data was collected by sampling the JVM at 5 second interval for the maximum allocated memory. Allocated memory is the memory procured by the JVM from the OS, and not all of the allocated memory is used by the heap. Hence, the data may be inaccurate.

In case of completed configurations with non-trivial state space, while there are cases where the proposed approaches fair better than E configuration and vice versa, the memory consumption of the proposed approaches are comparable to that of the EPOR approach. In case of FESC configurations, the configurations follow the complexity analysis and consume more memory as they cover larger

state space.

In almost all terminated FESC configurations, the maximum allowed heap space is consumed. Hence, it is safe to conclude that FESC configurations on programs with non-trivial state space will be heavily memory intensive. In other terminated non-FESC configurations, SDPOR configurations are more memory intensive than the other configurations; this follows the complexity analysis of SDPOR. Interestingly, every terminated non-FESC SC configurations require less memory in comparison with the non-FESC E configuration. Hence, SC configurations may be better suited for memory-scarce environments.

States and Transitions The data in Table 6.3 corresponds to the number of visited states while model checking the input programs in various configurations. The data relative to E configuration is given in Table 6.4. Similarly, the number of matched states (revisited fully explored states) and number of executed transitions while model checking are given in Table 6.5 (Table 6.6) and Table 6.7 (Table 6.8), respectively.

In case of completed configurations, from the data in Table 6.4a, more states are explored in SC and SC+F configurations due to the inaccuracies in the static program dependence. This is also true for SD and SD+F configurations in the context of programs with trivial state space. The configurations that combine EPOR and the proposed approaches almost always perform better than the E configuration.

In case of terminated configurations, the above observations hold in the allotted execution time (Table 6.4b). Hence, it is most likely, by projection, the above observations will hold if these configurations are executed to completion.

In case of input programs with zero errors, as expected, the state count in non-FESC configuration are very close or identical to that of their FESC counterpart.

All of the above observations also hold for matched states and transition data as well.

Errors The data in Table 6.9 corresponds to the number of errors uncovered in the input programs in various configurations. There were a few input programs with zero errors, and these are not presented in the table.

From the data corresponding to the completely executed configurations (in Table 6.9a), when FESC is not used, the number of errors detected by the proposed approaches is always less than or equal to those detected by EPOR approach. When FESC is used, the number of errors detected by the proposed approaches is lower than those detected by EPOR approach for AC1 but higher for BB4. This clearly indicates that there are programs for which either approaches can perform relatively better.

In the case of terminated configurations, the data indicates that SPD-CSS (SC) approach quickly uncovers errors in the input programs; SPD-CSS uncovers errors quicker than the more optimal SDPOR (SD) approach. This behavior may be due to the non-determinism in the construction of the sufficient sets (the data for DS4 supports this observation). In cases (DS7) where all configurations detect errors, more errors are detected in the SC configuration in comparison with the E configuration. However, due to incompletion and based on the previous results, it is possible that the E configuration will detect more errors than the other configuration upon completion.

Based on the preliminary data and theoretical reasoning, I conjecture that the proposed approaches will uncover errors quickly with lower number of false positives in comparison with the

Conf	SC	SD	SC+E	E+SD	E	SC+F	SD+F	SC+F+E	E+SD+F
AC1	2163	2228	3103	3124	4209	6178	6280	6905	7414
AC2	2079	1016	1009	516	769	6311	7184	7243	7113
AC3	1013	1419	779	578	495	1774	1865	1859	1698
BB1	594	934	1197	1144	94	1252	98	1252	1148
BB4	60	73	93	93	92	100	98	100	100
BB8	59	72	91	91	91	92	92	92	92
DL1	60	72	92	91	91	92	92	92	92
DL2	59	71	91	91	90	91	91	91	91
DL3	59	71	91	91	90	91	91	91	91
DP1	533	71	91	91	91	117	117	117	117
DP2	615	72	91	91	91	128	128	128	128
DP3	533	72	91	91	91	128	128	128	128
DP4	903	92	92	92	92	867	194	194	194
DP5	596	92	93	93	92	820	152	153	153
DP6	515	92	93	93	92	692	151	151	151
MD3	5937	14842	6135	5932	5890	10539	14407	6207	5966
RT3	5526	6381	5960	5607	5783	6942	6640	6666	5652
PL1	1358	1151	829	232	799	1641	1269	1481	248
PC3	430	73	93	93	92	95	95	95	95
PC4	1046	1143	1352	1351	95	1356	1356	1460	1356
RW1	654	1022	106	106	106	1283	1152	107	107
RW2	1422	981	563	717	505	1182	826	475	628
RW3	144	92	93	93	93	94	94	94	94
RW4	987	1039	94	94	93	1353	1143	95	95
RW5	1019	625	60	61	59	693	630	64	65
RP13	2227	2074	1322	1286	872	1409	1347	1631	1956
RP15	5119	13922	6405	7468	6382	12220	14390	10279	7954
RP18	2736	1480	1486	2151	2306	2971	3126	3612	4041
SB1	392	74	96	95	3314	632	715	129	127
SB4	1150	604	851	1015	628	830	760	849	824

(a)

Conf	SC	SD	SC+E	E+SD	E	SC+F	SD+F	SC+F+E	E+SD+F
DS1	3075	12172	6221	14043	6579	8462	14268	13351	14300
DS2	2033	12356	6665	14247	5549	10150	14252	13792	14293
DS4	4447	10549	7910	13811	7261	12701	14272	13918	14315
DS7	2618	8645	4969	7226	3599	7247	7967	7358	7282
RP12	7325	9671	7541	14613	8189	14940	11424	14495	14735
RP14	6378	9378	6990	9033	7357	12882	12178	11224	11003
SB2	7284	11667	8253	10583	8723	12376	14894	12529	12682

(b)

Table 6.2: Maximum memory (in MB) consumed to execute in various configuration on various input programs. (a) is the memory consumption for input programs on which the end-to-end execution completed within 10 hours for at least one configuration. (b) is the memory consumption for input programs on which the end-to-end execution was terminated for all configurations.

Conf	SC	SD	SC+E	E+SD	E	SC+F	SD+F	SC+F+E	E+SD+F
AC1	161172	47522	78869	9240	256479	35922	35478	18928	8248
AC2	18051	12766	5568	1857	11889	18051	12766	5568	1857
AC3	27470	16053	6815	1946	11030	27470	16053	6815	1946
BB1	1586	929	794	644	90	1586	929	794	644
BB4	46	44	7	5	122	257	255	109	50
BB8	152	149	21	23	28	152	149	21	23
DL1	173	113	115	89	178	173	135	115	111
DL2	47	34	8	8	8	47	34	8	8
DL3	61	43	7	4	7	61	46	7	7
DP1	746	304	30	16	30	746	304	30	16
DP2	798	311	51	18	51	798	325	51	28
DP3	798	311	51	18	51	798	325	51	28
DP4	1618	373	30	16	30	1621	376	30	16
DP5	1910	442	8	9	153	1930	448	8	9
DP6	1910	442	8	9	153	1930	448	8	9
MD3	11908	9150	4419	2478	200622	10383	8463	4419	2478
RT3	1831	22909	2298	171	4616	1812	22909	2186	171
PL1	301293	12710	69682	82	7379	301293	12710	69682	82
PC3	291	287	48	48	57	291	287	48	48
PC4	1007	582	88	88	103	1007	582	88	88
RW1	1403	861	102	72	124	1415	875	104	72
RW2	13338	9100	5186	2880	1214	13342	10335	4947	4215
RW3	1403	861	102	72	124	1415	875	104	72
RW4	1661	979	121	88	134	1671	984	121	89
RW5	5113	2616	173	177	58	5161	2625	180	183
RP13	5760	1619	1788	1587	6577	5760	1619	1788	1587
RP15	1043247	251698	568928	95657	612954	1043247	251698	568928	95657
RP18	48232	20550	23408	14497	26854	48232	20550	23408	14497
SB1	1870	743	442	54	11205	1870	803	442	114
SB4	31343	22866	19349	11167	213865	31346	23298	19353	11599

(a)

Conf	SC	SD	SC+E	E+SD	E	SC+F	SD+F	SC+F+E	E+SD+F
DS1	484320	280717	245323	154538	318652	107674	186871	170504	110420
DS2	586207	312253	302288	175784	439430	126848	192269	193605	119919
DS4	927413	199748	256656	122645	282599	245509	164164	151804	96442
DS7	802545	247030	136911	77217	157684	111268	74080	75774	43290
RP12	2854899	320514	1757174	636581	1010626	1849804	316597	1526923	571418
RP14	1496403	523174	916175	387088	480884	1222803	503619	850187	377415
SB2	3567474	1025894	1458629	599683	722112	2035248	1001193	1167223	568748

(b)

Table 6.3: The count of encountered states in various configuration on various input programs. (a) is the state count for input programs on which the end-to-end execution completed within 10 hours for at least one configuration. (b) is the state count for input programs on which the end-to-end execution was terminated for all configurations.

Conf	SC	SD	SC+E	E+SD	SC+F	SD+F	SC+F+E	E+SD+F
AC1	-95307	-208957	-177610	-247239	-220557	-221001	-237551	-248231
AC2	6162	877	-6321	-10032	6162	877	-6321	-10032
AC3	16440	5023	-4215	-9084	16440	5023	-4215	-9084
BB1	1496	839	704	554	1496	839	704	554
BB4	-76	-78	-115	-117	135	133	-13	-72
BB8	124	121	-7	-5	124	121	-7	-5
DL1	-5	-65	-63	-89	-5	-43	-63	-67
DL2	39	26	0	0	39	26	0	0
DL3	54	36	0	-3	54	39	0	0
DP1	716	274	0	-14	716	274	0	-14
DP2	747	260	0	-33	747	274	0	-23
DP3	747	260	0	-33	747	274	0	-23
DP4	1588	343	0	-14	1591	346	0	-14
DP5	1757	289	-145	-144	1777	295	-145	-144
DP6	1757	289	-145	-144	1777	295	-145	-144
MD3	-188714	-191472	-196203	-198144	-190239	-192159	-196203	-198144
RT3	-2785	18293	-2318	-4445	-2804	18293	-2430	-4445
PL1	293914	5331	62303	-7297	293914	5331	62303	-7297
PC3	234	230	-9	-9	234	230	-9	-9
PC4	904	479	-15	-15	904	479	-15	-15
RW1	1279	737	-22	-52	1291	751	-20	-52
RW2	12124	7886	3972	1666	12128	9121	3733	3001
RW3	1279	737	-22	-52	1291	751	-20	-52
RW4	1527	845	-13	-46	1537	850	-13	-45
RW5	5055	2558	115	119	5103	2567	122	125
RP13	-817	-4958	-4789	-4990	-817	-4958	-4789	-4990
RP15	430293	-361256	-44026	-517297	430293	-361256	-44026	-517297
RP18	21378	-6304	-3446	-12357	21378	-6304	-3446	-12357
SB1	-9335	-10462	-10763	-11151	-9335	-10402	-10763	-11091
SB4	-182522	-190999	-194516	-202698	-182519	-190567	-194512	-202266

(a)

Conf	SC	SD	SC+E	E+SD	SC+F	SD+F	SC+F+E	E+SD+F
DS1	165668	-37935	-73329	-164114	-210978	-131781	-148148	-208232
DS2	146777	-127177	-137142	-263646	-312582	-247161	-245825	-319511
DS4	644814	-82851	-25943	-159954	-37090	-118435	-130795	-186157
DS7	644861	89346	-20773	-80467	-46416	-83604	-81910	-114394
RP12	1844273	-690112	746548	-374045	839178	-694029	516297	-439208
RP14	1015519	42290	435291	-93796	741919	22735	369303	-103469
SB2	2845362	303782	736517	-122429	1313136	279081	445111	-153364

(b)

Table 6.4: The E configuration relative count of encountered states in various configuration on various input programs. (a) is the state count for input programs on which the end-to-end execution completed within 10 hours for at least one configuration. (b) is the state count for input programs on which the end-to-end execution was terminated for all configurations.

Conf	SC	SD	SC+E	E+SD	E	SC+F	SD+F	SC+F+E	E+SD+F
AC1	114976	14334	52419	1441	415793	26061	12167	13901	2324
AC2	11475	5099	3889	485	13067	11475	5099	3889	485
AC3	21477	7469	4863	342	11239	21477	7469	4863	342
BB1	835	309	413	233	90	838	343	420	265
BB4	27	25	1	0	88	242	239	104	45
BB8	56	52	13	10	28	56	52	14	15
DL1	83	43	43	27	114	83	56	43	40
DL2	20	11	1	1	1	20	11	1	1
DL3	24	14	0	0	0	24	14	0	0
DP1	792	272	36	4	57	800	274	41	6
DP2	831	276	63	5	99	843	284	70	14
DP3	831	276	63	5	99	843	284	70	14
DP4	2056	334	36	4	57	2092	336	41	6
DP5	2304	417	2	1	346	2359	423	3	3
DP6	2304	417	2	1	346	2359	423	3	3
MD3	743	0	647	43	197416	471	0	647	43
RT3	156	5276	1119	5	3569	155	5276	1033	5
PL1	362901	13425	78117	17	15859	362901	13425	78117	17
PC3	87	84	31	31	44	87	84	31	31
PC4	615	284	64	64	85	615	284	64	64
RW1	1453	733	127	66	296	1493	770	139	79
RW2	10795	6438	3480	1606	2559	11328	7562	3758	2876
RW3	1453	733	127	66	296	1493	770	139	79
RW4	1647	835	154	79	299	1702	854	157	93
RW5	4744	1937	114	75	108	4831	1995	129	84
RP13	1764	747	641	495	6562	1764	747	641	495
RP15	753132	108054	328218	43268	1010878	753132	108054	328218	43268
RP18	34787	10783	13496	7225	40226	34787	10783	13496	7225
SB1	1650	663	147	8	22139	1650	683	147	24
SB4	24638	16879	12790	6408	493092	24770	17254	12922	6783

(a)

Conf	SC	SD	SC+E	E+SD	E	SC+F	SD+F	SC+F+E	E+SD+F
DS1	176323	78041	97748	60075	137135	37712	51978	67193	42890
DS2	213654	86737	121775	68830	188600	44942	53477	76500	46644
DS4	219615	44681	83012	38192	96813	57062	36282	48727	30033
DS7	528986	124269	78195	30770	129614	61175	28997	40260	17457
RP12	1773213	157402	1028687	309869	2168202	1143564	155417	893958	281616
RP14	701784	207986	421679	153376	961519	566817	200552	389611	149814
SB2	3870150	1223108	2250433	922882	3246748	2163348	981585	1825146	881842

(b)

Table 6.5: The count of encountered matched states in various configuration on various input programs. (a) is the matched state count for input programs on which the end-to-end execution completed within 10 hours for at least one configuration. (b) is the matched state count for input programs on which the end-to-end execution was terminated for all configurations.

Conf	SC	SD	SC+E	E+SD	SC+F	SD+F	SC+F+E	E+SD+F
AC1	-300817	-401459	-363374	-414352	-389732	-403626	-401892	-413469
AC2	-1592	-7968	-9178	-12582	-1592	-7968	-9178	-12582
AC3	10238	-3770	-6376	-10897	10238	-3770	-6376	-10897
BB1	745	219	323	143	748	253	330	175
BB4	-61	-63	-87	-88	154	151	16	-43
BB8	28	24	-15	-18	28	24	-14	-13
DL1	-31	-71	-71	-87	-31	-58	-71	-74
DL2	19	10	0	0	19	10	0	0
DL3	24	14	0	0	24	14	0	0
DP1	735	215	-21	-53	743	217	-16	-51
DP2	732	177	-36	-94	744	185	-29	-85
DP3	732	177	-36	-94	744	185	-29	-85
DP4	1999	277	-21	-53	2035	279	-16	-51
DP5	1958	71	-344	-345	2013	77	-343	-343
DP6	1958	71	-344	-345	2013	77	-343	-343
MD3	-196673	-197416	-196769	-197373	-196945	-197416	-196769	-197373
RT3	-3413	1707	-2450	-3564	-3414	1707	-2536	-3564
PL1	347042	-2434	62258	-15842	347042	-2434	62258	-15842
PC3	43	40	-13	-13	43	40	-13	-13
PC4	530	199	-21	-21	530	199	-21	-21
RW1	1157	437	-169	-230	1197	474	-157	-217
RW2	8236	3879	921	-953	8769	5003	1199	317
RW3	1157	437	-169	-230	1197	474	-157	-217
RW4	1348	536	-145	-220	1403	555	-142	-206
RW5	4636	1829	6	-33	4723	1887	21	-24
RP13	-4798	-5815	-5921	-6067	-4798	-5815	-5921	-6067
RP15	-257746	-902824	-682660	-967610	-257746	-902824	-682660	-967610
RP18	-5439	-29443	-26730	-33001	-5439	-29443	-26730	-33001
SB1	-20489	-21476	-21992	-22131	-20489	-21456	-21992	-22115
SB4	-468454	-476213	-480302	-486684	-468322	-475838	-480170	-486309

(a)

Conf	SC	SD	SC+E	E+SD	SC+F	SD+F	SC+F+E	E+SD+F
DS1	39188	-59094	-39387	-77060	-99423	-85157	-69942	-94245
DS2	25054	-101863	-66825	-119770	-143658	-135123	-112100	-141956
DS4	122802	-52132	-13801	-58621	-39751	-60531	-48086	-66780
DS7	399372	-5345	-51419	-98844	-68439	-100617	-89354	-112157
RP12	-394989	-2010800	-1139515	-1858333	-1024638	-2012785	-1274244	-1886586
RP14	-259735	-753533	-539840	-808143	-394702	-760967	-571908	-811705
SB2	623402	-2023640	-996315	-2323866	-1083400	-2265163	-1421602	-2364906

(b)

Table 6.6: The E configuration relative count of encountered matched states in various configuration on various input programs. (a) is the matched state count for input programs on which the end-to-end execution completed within 10 hours for at least one configuration. (b) is the matched state count for input programs on which the end-to-end execution was terminated for all configurations.

Conf	SC	SD	SC+E	E+SD	E	SC+F	SD+F	SC+F+E	E+SD+F
AC1	2617743	619812	1471300	149699	5599303	506774	425965	363324	125174
AC2	443403	273092	165798	38522	418901	443403	273092	165798	38522
AC3	828223	482608	238414	55827	499105	828223	482608	238414	55827
BB1	7772	4677	5576	4493	2448	7775	4711	5583	4525
BB4	301	282	101	83	869	727	707	306	230
BB8	810	796	379	408	507	810	796	380	413
DL1	665	388	491	324	809	665	512	491	448
DL2	182	132	77	77	77	182	132	77	77
DL3	198	133	86	65	86	198	154	86	86
DP1	2426	925	266	173	287	2434	927	271	175
DP2	2584	950	391	192	427	2596	990	398	243
DP3	2589	955	396	197	432	2601	995	403	248
DP4	6197	1268	321	228	342	6238	1275	326	230
DP5	7123	1564	142	144	1434	7200	1578	143	146
DP6	7115	1556	138	140	1426	7192	1570	139	142
MD3	7251107	2762960	529098	103302	8287084	6303446	2105125	529098	103302
RT3	1455033	435667	711010	42604	1247000	1437594	435667	702675	42604
PL1	1081729	46687	323185	553	76291	1081729	46687	323185	553
PC3	2189	2177	1363	1375	1637	2189	2177	1363	1375
PC4	9986	7488	3460	3469	4057	9986	7488	3460	3469
RW1	5480	3564	1404	989	1746	5578	3689	1432	1002
RW2	47032	30692	22955	12789	12312	47612	35327	22517	18766
RW3	5480	3564	1404	989	1746	5578	3689	1432	1002
RW4	7063	4426	1794	1311	2091	7190	4490	1797	1335
RW5	22702	12243	2975	2665	2116	23098	12346	3058	2787
RP13	98168	31180	63785	60595	362027	98168	31180	63785	60595
RP15	4823439	1047995	3259745	413817	5709125	4823439	1047995	3259745	413817
RP18	252853	105190	154699	96363	321962	252853	105190	154699	96363
SB1	5656	2090	1502	256	47317	5656	2300	1502	462
SB4	90649	63264	56343	31290	984758	90794	64574	56489	32600

(a)

Conf	SC	SD	SC+E	E+SD	E	SC+F	SD+F	SC+F+E	E+SD+F
DS1	13130000	8420000	10761570	6858336	14480000	2980661	5605795	7540000	4895085
DS2	15861823	9359041	13180000	7805424	19900000	3491532	5770000	8560000	5312019
DS4	29990000	6570000	12170000	5874295	13800000	8020000	5430000	7180000	4635284
DS7	14401190	4104736	4603123	2522397	5143947	2597494	1721830	2576842	1417963
RP12	9006579	967201	5700000	2055708	7522067	5880000	955702	4980000	1812125
RP14	5400914	2018866	3419295	1599607	6417914	4373007	1962390	3205777	1570605
SB2	12208052	3970000	6286847	2661671	10869856	6910000	3534082	5009925	2521013

(b)

Table 6.7: The count of executed transitions in various configuration on various input programs. (a) is the transition count for input programs on which the end-to-end execution completed within 10 hours for at least one configuration. (b) is the transition count for input programs on which the end-to-end execution was terminated for all configurations.

Conf	SC	SD	SC+E	E+SD	SC+F	SD+F	SC+F+E	E+SD+F
AC1	-2981560	-4979491	-4128003	-5449604	-5092529	-5173338	-5235979	-5474129
AC2	24502	-145809	-253103	-380379	24502	-145809	-253103	-380379
AC3	329118	-16497	-260691	-443278	329118	-16497	-260691	-443278
BB1	5324	2229	3128	2045	5327	2263	3135	2077
BB4	-568	-587	-768	-786	-142	-162	-563	-639
BB8	303	289	-128	-99	303	289	-127	-94
DL1	-144	-421	-318	-485	-144	-297	-318	-361
DL2	105	55	0	0	105	55	0	0
DL3	112	47	0	-21	112	68	0	0
DP1	2139	638	-21	-114	2147	640	-16	-112
DP2	2157	523	-36	-235	2169	563	-29	-184
DP3	2157	523	-36	-235	2169	563	-29	-184
DP4	5855	926	-21	-114	5896	933	-16	-112
DP5	5689	130	-1292	-1290	5766	144	-1291	-1288
DP6	5689	130	-1288	-1286	5766	144	-1287	-1284
MD3	-1035977	-5524124	-7757986	-8183782	-1983638	-6181959	-7757986	-8183782
RT3	208033	-811333	-535990	-1204396	190594	-811333	-544325	-1204396
PL1	1005438	-29604	246894	-75738	1005438	-29604	246894	-75738
PC3	552	540	-274	-262	552	540	-274	-262
PC4	5929	3431	-597	-588	5929	3431	-597	-588
RW1	3734	1818	-342	-757	3832	1943	-314	-744
RW2	34720	18380	10643	477	35300	23015	10205	6454
RW3	3734	1818	-342	-757	3832	1943	-314	-744
RW4	4972	2335	-297	-780	5099	2399	-294	-756
RW5	20586	10127	859	549	20982	10230	942	671
RP13	-263859	-330847	-298242	-301432	-263859	-330847	-298242	-301432
RP15	-885686	-4661130	-2449380	-5295308	-885686	-4661130	-2449380	-5295308
RP18	-69109	-216772	-167263	-225599	-69109	-216772	-167263	-225599
SB1	-41661	-45227	-45815	-47061	-41661	-45017	-45815	-46855
SB4	-894109	-921494	-928415	-953468	-893964	-920184	-928269	-952158

(a)

Conf	SC	SD	SC+E	E+SD	SC+F	SD+F	SC+F+E	E+SD+F
DS1	-1350000	-6060000	-3718430	-7621664	-11499339	-8874205	-6940000	-9584915
DS2	-4038177	-10540959	-6720000	-12094576	-16408468	-14130000	-11340000	-14587981
DS4	16190000	-7230000	-1630000	-7925705	-5780000	-8370000	-6620000	-9164716
DS7	9257243	-1039211	-540824	-2621550	-2546453	-3422117	-2567105	-3725984
RP12	1484512	-6554866	-1822067	-5466359	-1642067	-6566365	-2542067	-5709942
RP14	-1017000	-4399048	-2998619	-4818307	-2044907	-4455524	-3212137	-4847309
SB2	1338196	-6899856	-4583009	-8208185	-3959856	-7335774	-5859931	-8348843

(b)

Table 6.8: The E configuration relative count of executed transitions in various configuration on various input programs. (a) is the transition count for input programs on which the end-to-end execution completed within 10 hours for at least one configuration. (b) is the transition count for input programs on which the end-to-end execution was terminated for all configurations.

Conf	SC	SD	SC+E	E+SD	E	SC+F	SD+F	SC+F+E	E+SD+F
AC1	5531	250	1982	77	14668	3650	3476	3727	1144
BB4	5	4	2	1	32	216	215	104	46
DL1	2	2	2	2	2	2	2	2	2
DL2	1	1	1	1	1	1	1	1	1
DL3	2	1	2	1	2	2	2	2	2
DP1	1	1	1	1	1	1	1	1	1
DP2	1	1	1	1	1	1	1	1	1
DP3	1	1	1	1	1	1	1	1	1
DP4	1	1	1	1	1	1	1	1	1
DP5	1	1	1	1	1	1	1	1	1
DP6	1	1	1	1	1	1	1	1	1
SB1	6	2	6	2	6	6	6	6	6

(a)

Conf	SC	SD	SC+E	E+SD	E	SC+F	SD+F	SC+F+E	E+SD+F
DS1	6922	3	448	0	0	984	0	0	0
DS2	8672	5	865	1	0	1366	0	0	0
DS4	21	0	0	0	0	0	0	0	0
DS7	10604	2490	4346	2409	4155	2679	1977	2228	2027
SB2	1	1	1	1	1	1	1	1	1

(b)

Table 6.9: The count of errors detected in various configuration on various input programs. (a) is the error count for input programs on which the end-to-end execution completed within 10 hours for at least one configuration. (b) is the error count for input programs on which the end-to-end execution was terminated for all configurations. For unmentioned input programs, the error count was zero in all configurations.

optimal EPOR-based approach. However, given the small sample space and incomplete experiments (in terms of execution), more experimentation is required to empirical prove the above conjecture.

6.4.4 State Level Experimental Results

While the above evaluation is useful to assess the relative merit of the approaches at a system level, it is insufficient to evaluate the approaches at a micro (state) level. Further, it is insufficient to evaluate the approaches in case of terminated experiments. Hence, to facilitate a local and complimentary evaluation, local (reduction) data at the state level was collected. This data contains the total number of enabled *(E)*, sufficient *(A)*, and independent *(I)* transitions encountered at the visited states of the system. Further, the weighted average of the ratio of sufficient and enabled transition *(A/E)* at each visited states of the system was also collected. In case of configurations involving SDPOR, the total number of dynamic dependences discovered merely via static dependence information *(S)* and via pure dynamic approach *(D)* was collected and the ratio of between these numbers *(D/S)* was calculated. In the data for each input program, this data is presented in the (c) table.

Observations The relative data indicates that the proposed non-FESC SPD-CSS based approaches are comparable to EPOR based approach in case of programs with trivial state space and better in case of programs with non-trivial state space.

From the reduction data, the weighted average of the ratio of the size of the sufficient set to the size of enabled set at a state is similar in case of the SC and SD configurations. This is true independent of the completion and FESC nature of the configuration. Hence, the effect of the approaches distributes evenly across the reduced state space.

Similarly, the ratio of the number of dynamic dependences triggered via PDD and via static program dependences[10] is less than or equal to 0.01. This implies that, if possible, dynamic POR based on PDD should be preferred.

Although experiments dedicated to SDD approach were not conducted, based on the data from (S) and (D) and the relation between SDD and PDD, it follows that the data for SDD approach will be the same or better than that for static program dependence. Upon factoring in state local thread aliveness information, the SDD approach will most likely provide reduction that is better than SPD-CSS approach but still not better than PDD approach.

6.5 Related Work

As mentioned in the earlier sections, Godefroid presented a collection of POR techniques in his dissertation [God95]. He acknowledged that the conditional stubborn sets, the most optimal of the proposed partial order reduction techniques based on dependences to reduce the number of explored states, was purely theoretical due to lack of a practical approach to calculate transitions reachable from a state. The proposed technique (SPD-CSS) in Section 6.2 of this chapter leverages program analysis and addresses this limitation of conditional stubborn sets in a sound and relatively accurate manner.

Kurshan et al. [KLM+98] described a similar approach based on compile-time calculation of partial order reduction. Their approach was focused on ensuring the reduction techniques were non-intrusive of the state space exploration algorithm. Also, they focused on breadth first variant of the exploration algorithm. In comparison, the proposed approach does modify the exploration algorithm and is targeted towards the depth first variant of the algorithm.

Stoller [Sto02] described a collection of POR techniques focused on model checking Java programs. These techniques were based on common synchronization pattern in Java programs — *use of locks to protect shared variables*. The basic idea was to increase the instances of same-thread transition sequences (and lengthen them) by only allowing context switches before lock acquisition operations and not before access to variables protected by locks. These techniques relied on user input to classify objects appropriately as required by the techniques; hence, the input could be validated at runtime. The proposed reduction (SPD-CSS and SDPOR) based on lock acquisition-based transition dependence is similar to Stoller's techniques with the exceptions that 1) the proposed reduction is fully automated as it depends on program analysis and does not require user input and 2) the proposed reduction relies on a weaker premise of mutual exclusion via locking to prohibit context switches before access to variables.

An alternate approach [FQ03a, FF04] for detecting when thread interleaving can be avoided is based on the notion of left- and right-movers as proposed in Lipton's Reduction Theory [Ric75]. The basic idea is to detect and move transitions in an execution history to maximize the length of same-thread transitions (atomic) sub-sequence while honoring the ordering imposed by inter-thread communication. This approach relies on user input to annotate intra-thread transition sequences as being *atomic*; hence, the approach is amenable to runtime checking of annotations. Given the

[10] These dependences are not pruned based on state local thread aliveness information.

similarity between Stoller's approach and this approach, the comparison with the proposed approach carries over as is.

Dwyer et al. [DHRR04] proposed a dynamic escape analysis based approach to efficiently calculate relatively accurate independence information and effect partial order reduction. While this approach is applicable in state space exploration frameworks where dynamic escape analysis can be easily implemented, the proposed approach (SPD-CSS) can be applied in any framework that supports sufficient set calculation.

In the realm of dynamic POR, Godefroid and Flanagan [FQ03b] proposed the first stateless state space exploration algorithm for acyclic state space. In comparison, SDPOR is stateful, can handle cyclic state space, and it leverages cheaper static dependence information.

Chapter 7

Conclusion

7.1 Summary

We[1] identified the incorrectness of existing notions of control dependence in the context of modern programs with zero or more than one exit points. As a solution, we proposed a new trace-based definitions of control dependences that can seamlessly handle programs with zero or more exit points written in structure programming languages such as Java as well as unstructured programming languages such as JVM language and assembly language. The usefulness and correctness of these definitions was illustrated in the context of program slicing. The feasibility of these definitions was demonstrated by designing polynomial-time algorithms to calculate control dependences according to the new definitions; further, the algorithms were implemented and realized in the slicing framework available in the Indus project.

The interaction between processes of a concurrent system is crucial to understand the behavior of concurrent systems. In a shared memory computation model, such interaction is captured by the notion of interference dependence and ready dependence. In an object oriented environment, it seems that points-to information can be used to calculate interference and ready dependence. However, it is hard to be accurate with context-free points-to information, and contextual points-to information is expensive to calculate. To address this situation, we[2] proposed an equivalence class based analysis to discover information about shared data based inter-thread communication. Specifically, the analysis calculates escape, shared read-write and write-write access, lock coupling, and wait-notify coupling information that can be (was successfully) used to improve the accuracy of interference and ready dependence calculation. Leveraging the structural genericity of the analysis, we easily extended the analysis to calculate intra-thread inter-procedural aliasing information; the cost and the accuracy of this extended analysis is comparable to that of classic data flow style points-to analysis. Further, the results from the analysis were employed to improve the accuracy of program slicing and to realize an efficient and accurate partial order reduction in the context of program verification via software model checking.

As an alternative to the classic dependence graph based program slicing algorithm, I developed a simple parametric static program slicing algorithm that operates on dependence relations (and not graphs). Using this algorithm, I successfully realized algorithms for the most common forms

[1] This was a collaborative effort with Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer, and John Hatcliff.
[2] This was a collaborative effort with John Hatcliff.

of slicing: backward, forward, and control. With a minor alteration, the same algorithm was used to realize calling context sensitive algorithms for various forms of slicing. Also, given the gradual increase in forms of and interest in context sensitivity, I have proposed trace/property sensitivity, a general form of context sensitivity that can uniformly capture various forms of context sensitivities. As another contribution, I have discovered scoping as an approach to meaningfully and accurately scale program slicing and other analyses in the context of large software. Beyond these fundamental contributions, I have successfully implemented the proposed approaches/techniques in the first publicly available program slicing framework for Java in the Indus project. Results from experiments based on this implementation have indicated that the proposed approaches are a step in the enabling the application of program slicing and other analysis in the context of large scale software.

While many have proposed various partial order reduction based approaches to alleviate the cost of model checking concurrent systems, countably few have explored the possibility of leveraging program analysis information to affect partial order reduction during state space exploration. As an experiment, I devised an algorithm to leverage the information calculated by the previously described equivalence class based analysis to efficiently realize partial order reduction during software model checking. I also developed the first stateful dynamic partial order reduction technique based on the static program dependence information. The data from preliminary experiments indicate that these techniques are simpler to realize, cost effective, and comparably accurate to more sophisticated, accurate, and dynamic partial order reduction techniques.

In summary, I have proposed (and demonstrated) a collection of feasible approaches/techniques to address scalability and accuracy issues common to program analysis of concurrent object oriented programs.

As every innovator, I hope and believe these contributions will make an impact on mainstream software development.

7.2 Future Work

Measure In every topic studied in this research effort, there was a lack of measure, either absolute or relative, to evaluate and compare the relative merits of alternative techniques. The best available approach to measure was to implement the techniques and compare them based on various aspects of the results, e.g. the best approach to compare existing slicing algorithms with proposed slicing algorithm was to implement the algorithms and compare the size of resulting slice. However, the ideal approach would have been to have a model onto which the inputs and outputs of various techniques could be mapped and to evaluate the techniques in the context of such a model. Such comparison would be accurate (due to being based on common model), noise-free (due to the absence of any external and hidden influence possible during empirical evaluation) and easy (due to reuse of previous results established w.r.t. the common model).

In terms of realizing such a model, one possibility is for such a model to quantitatively capture various aspects of input programs in terms of a set of static features and a set of dynamic features that can be derived from the set of static features.

Independent of the approach, devising such a model that is simple to use is certainly an interesting, challenging, and relevant effort.

Modularity, Compositionality, Openness, and Heterogeneity The proposed approaches assumed that the whole program/system is available; however, this is untrue in case of systems

that composed at runtime via facilities such as dynamic class loading and reflection. Native code is another aspect that contributes to this issue. Further, as the size of the systems increases, the cost of various approaches will increase. A solution to curb the increasing cost is to design approaches that are modular and compositional.

However, the heterogeneity (e.g. Java-Python bridge) and openness (e.g. native code) of systems can be almost impossible to handle and, interestingly, almost all systems exhibit at least one of these traits. Hence, analysis should be capable of handling these traits.

In other words, future solutions need to be modular and compositional as well as specification-based in a manner that they can seamlessly handle open and/or heterogeneous systems.

Property Sensitivity Given the genericity of property sensitivity, most forms of sensitivities used by the program analysis can be cast as a form of property sensitivity. Hence, it would be an interesting effort to develop a formal framework to reason about property sensitivity. Given the trend towards techniques being more application, domain, and/or usage-pattern specific, a generic framework to reason about techniques based on their sensitivity towards properties would be helpful.

Trace-based Reasoning Many recent scalable and accurate program analyses track the state of the system at a level of abstraction that is closer to the actual runtime state of the system, e.g. calling context sensitive points-to analysis. As level of abstraction decreases, the analyses seem to be processing abstract execution history of the system. In other words, analyses are processing traces of the system. Hence, it would be appropriate and probably simpler to reason about program analyses in terms of traces (the appropriate abstraction formalism) as opposed reasoning using formalisms higher than the employed abstraction. An ideal solution would be to develop a trace-based framework for reasoning about program analysis.

Appendix A

Data From Slicing Experiments

The data collected in the Java Grande benchmark based slicing experiments are presented both as raw numbers and as graphs. Please refer to Section 5.5 for the conclusions drawn from the experiments.

A.1 Experimental Setup

As in the experiments in Section 3.7, Java Grande Benchmark programs were considered as input programs. For each program in the benchmark, 26 slices were generated using different algorithms and different configurations. These 26 slices comprised of three sets of slices.

The first set of 11 slices is used to evaluated various slicing algorithms in combination with varying levels of accuracy of interference and ready dependence information. The legends are given below.

BSA_t Backward slice with type-based interference and ready dependence information.

BSA_e Backward slice with escape-based interference and ready dependence information.

BSA Backward slice with entity-based interference and ready dependence information.

CCSBSA+ Optimized calling context sensitive backward slice with entity-based interference and ready dependence information.

CCSBSA+_t Optimized calling context sensitive backward slice with type-based interference and ready dependence information.

CCSBSA+_e Optimized calling context sensitive backward slice with escape-based interference and ready dependence information.

CCSBSA+_1D-PS Optimized calling context sensitive backward slice with entity-based interference and ready dependence information and calling context enriched seed criteria generation with the length of the seed calling context limited to 10.

2D-PSSA_4 Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and source and destination specific data-based property sensitivity with the length of property sensitive calling contexts limited to 4.

2D-PSSA_16 Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and source and destination specific data-based property sensitivity with the length of property sensitive calling contexts limited to 16.

2D-PSSA_256 Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and source and destination specific data-based property sensitivity with the length of property sensitive calling contexts limited to 256.

2D-PSSA_10000 Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and source and destination specific data-based property sensitivity with the length of property sensitive calling contexts limited to 10000.

The second set of 6 slices is used to evaluate the combination of accurate dependence calculation as described in the previous chapter along with property sensitivity in the context of calling context sensitive slicing.

1C-PSSA Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and control based property sensitivity with the length of the property specific calling context limited to 256.

1D-PSSA Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and destination specific data-based property sensitivity with the length of the property sensitive calling context limited to 256.

2D-PSSA Property sensitive calling context sensitive backward slice with entity-based interference and ready dependence information and source and destination specific data-based property sensitivity with the length of the property sensitive calling contextx limited to 256.

t_1C-PSSA Property sensitive calling context sensitive backward slice with type-based interference and ready dependence information and control based property sensitivity with the length of the property specific calling context limited to 256.

t_1D-PSSA Property sensitive calling context sensitive backward slice with type-based interference and ready dependence information and destination specific data-based property sensitivity with the length of the property sensitive calling context limited to 256.

t_2D-PSSA Property sensitive calling context sensitive backward slice with type-based interference and ready dependence information and source and destination specific data-based property sensitivity with the length of the property sensitive calling contextx limited to 256.

The third set of 9 slices is used evaluate the combination of various optimizations to equivalence class analysis along with the proposed slicing algorithms.

BSA_sf Backward slice with entity-based interference and ready dependence and static filtering optimized equivalence class analysis.

BSA_tf Backward slice with entity-based interference and ready dependence and type filtering optimized equivalence class analysis.

BSA_both Backward slice with entity-based interference and ready dependence and both static and type filtering optimized equivalence class analysis.

CCSBSA+_sf Optimized calling context sensitive backward slice with entity-based interference and ready dependence and static filtering optimized equivalence class analysis.

CCSBSA+_tf Optimized calling context sensitive backward slice with entity-based interference and ready dependence and type filtering optimized equivalence class analysis.

CCSBSA+_both Optimized calling context sensitive backward slice with entity-based interference and ready dependence and both static and type filtering optimized equivalence class analysis.

2D-PSSA_sf Property sensitive context sensitive backward slice with entity-based interference and ready dependence, static filtering optimization for equivalence class analysis, and source and destination specific data-based property sensitivity with the length of the property sensitive calling contextx limited to 256.

2D-PSSA_tf Property sensitive context sensitive backward slice with entity-based interference and ready dependence, static filtering optimization for equivalence class analysis, and source and destination specific data-based property sensitivity with the length of the property sensitive calling contextx limited to 256.

2D-PSSA_both Property sensitive context sensitive backward slice with entity-based interference and ready dependence, static filtering optimization for equivalence class analysis, and source and destination specific data-based property sensitivity with the length of the property sensitive calling contextx limited to 256.

For each slice generation, the following data was collected.

Time Three time measures were taken in each experiment. The first was the measure of the time taken to merely identify the slice. The second was the measure of the time taken to identify the slice and inject executability into the slice. This subsumed the first measure. The third measure accounted for the time taken by other analysis other than residualization and serialization. These measures are presented as three slash separated values under the column *Time*.

Memory Two memory measures were taken in each experiment. The first was the measure of memory consumed during slice identification and the second was the measure of memory consumed during slicing and other analysis except residualization and serialization. These measures are presented as two slash separated values under the column *Memory*.

Classes The number of classes in the slice.

Methods The number of methods in the slice.

Fields The number of fields in the slice.

Stmts The number of Jimple statements in the slice.

Exprs The number of Jimple expressions in the slice.

Bytecodes The number of compressed bytecodes in the slice.

The experiments were executed on an assertion enabled JVM available as part of JDK 1.6.0_b104 with 512MB of maximum heap space on a 1.4GHz and 1GB Linux box.

The time and memory measurements were collected by instrumenting the code via AspectJ[1].

[1] http://www.eclipse.org/aspectj

A.2 Experimental Data

Configuration	Time (s)	Memory (MB)	Classes	Methods	Fields	Stmts	Exprs	Bytecodes (Bytes)
Unsliced			121	301	212	5782	16195	
BSA_t	1/13/71	3/21	118	319	86	5962	16683	65651
BSA_e	1/15/78	3/20	118	319	86	5962	16683	65648
BSA	1/13/74	3/20	118	319	86	5962	16683	65648
CCSBSA+	1/12/67	2/20	94	262	82	5672	16136	55488
CCSBSA+_t	1/13/71	2/20	94	262	82	5672	16136	55488
CCSBSA+_e	1/13/76	2/20	94	262	82	5672	16136	55488
CCSBSA+_1D-PS	1/12/67	2/21	85	240	75	5487	15793	51626
2D-PSSA_4	1/12/68	2/21	85	237	74	5481	15782	51407
2D-PSSA_16	2/12/68	2/21	85	237	74	5481	15782	51407
2D-PSSA_256	1/12/68	2/21	85	237	74	5481	15782	51407
2D-PSSA_10000	2/12/68	2/21	85	237	74	5481	15782	51407
1C-PSSA	2/14/78	3/21	85	240	75	5487	15793	51592
1D-PSSA	2/14/78	2/21	85	238	74	5481	15782	51435
2D-PSSA	2/14/77	2/21	85	234	74	5387	15629	51077
t_1C-PSSA	2/15/76	3/21	85	240	75	5487	15793	51586
t_1D-PSSA	2/14/76	2/21	85	238	74	5481	15782	51429
t_2D-PSSA	2/14/75	2/21	85	237	74	5481	15782	51404
BSA_sf	1/13/70	3/21	118	319	86	5962	16683	65648
BSA_tf	1/13/73	3/20	118	319	86	5962	16683	65648
BSA_both	1/13/73	3/21	118	319	86	5962	16683	65648
CCSBSA+_sf	1/13/75	2/20	94	262	82	5672	16136	55488
CCSBSA+_tf	1/13/73	2/20	94	262	82	5672	16136	55488
CCSBSA+_both	1/13/73	2/20	94	262	82	5672	16136	55488
2D-PSSA_sf	2/14/77	2/21	85	234	74	5387	15629	51076
2D-PSSA_tf	2/14/77	2/21	85	234	74	5387	15629	51079
2D-PSSA_both	2/14/78	2/21	85	234	74	5387	15629	51076

Table A.1: Data from slicing *Bar* benchmark program from the Java Grande suite.

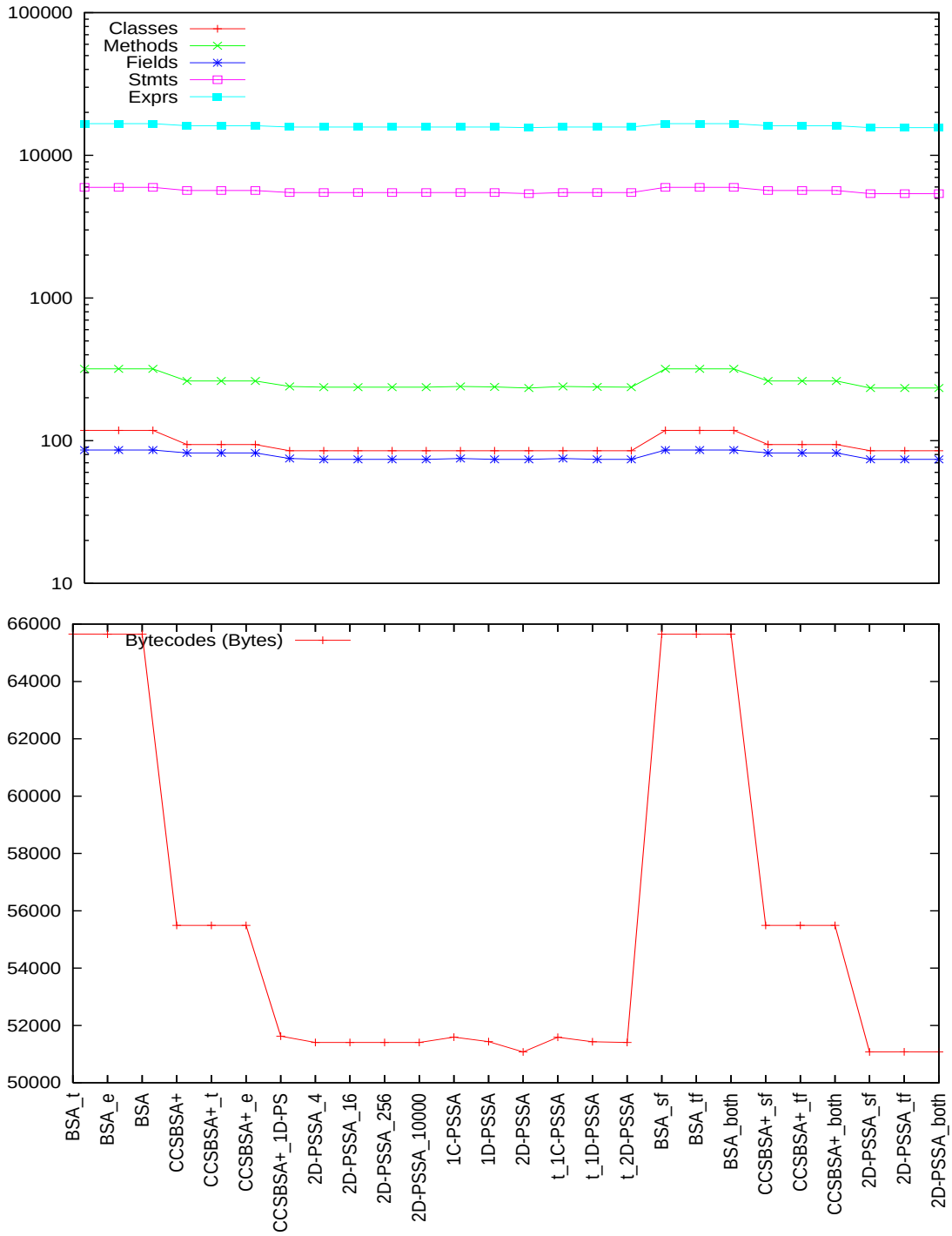

Figure A.1: Graphical representation of the data (Table A.1) from slicing *Bar* benchmark program from the Java Grande suite.

Configuration	Time (s)	Memory (MB)	Classes	Methods	Fields	Stmts	Exprs	Bytecodes (Bytes)
Unsliced			133	336	236	6515	18462	
BSA_t	1/14/75	3/24	129	371	105	6753	19098	73817
BSA_e	1/14/79	3/23	129	371	105	6753	19098	73799
BSA	1/14/77	3/24	129	371	105	6753	19098	73799
CCSBSA+	1/14/77	3/23	105	307	101	6439	18511	63477
CCSBSA+_t	1/14/73	3/23	105	308	101	6443	18522	63506
CCSBSA+_e	1/13/79	3/23	105	305	101	6431	18497	63427
CCSBSA+_1D-PS	1/14/79	3/24	96	283	94	6246	18154	59566
2D-PSSA_4	2/14/80	3/24	96	275	93	6142	17983	58964
2D-PSSA_16	2/15/78	3/24	96	278	93	6236	18136	59293
2D-PSSA_256	2/14/75	3/24	96	278	93	6236	18136	59293
2D-PSSA_10000	2/14/80	3/24	96	275	93	6142	17983	58964
1C-PSSA	3/15/81	3/24	96	281	94	6242	18147	59462
1D-PSSA	2/14/80	3/24	96	278	93	6236	18136	59184
2D-PSSA	2/14/76	3/24	96	275	93	6142	17983	58855
t_1C-PSSA	3/15/75	3/24	96	281	94	6242	18147	59461
t_1D-PSSA	2/15/76	3/24	96	278	93	6236	18136	59184
t_2D-PSSA	2/15/74	3/24	96	275	93	6142	17983	58855
BSA_sf	1/14/78	3/24	129	371	105	6753	19098	73799
BSA_tf	1/14/77	3/24	129	371	105	6753	19098	73799
BSA_both	1/14/77	3/24	129	371	105	6753	19098	73799
CCSBSA+_sf	1/14/78	3/23	105	305	101	6431	18497	63427
CCSBSA+_tf	1/14/77	3/23	105	305	101	6431	18497	63427
CCSBSA+_both	1/14/77	3/23	105	305	101	6431	18497	63427
2D-PSSA_sf	2/14/76	3/24	96	275	93	6142	17983	58855
2D-PSSA_tf	2/14/76	3/24	96	278	93	6236	18136	59293
2D-PSSA_both	2/14/80	3/24	96	275	93	6142	17983	58855

Table A.2: Data from slicing *Crp* benchmark program from the Java Grande suite.

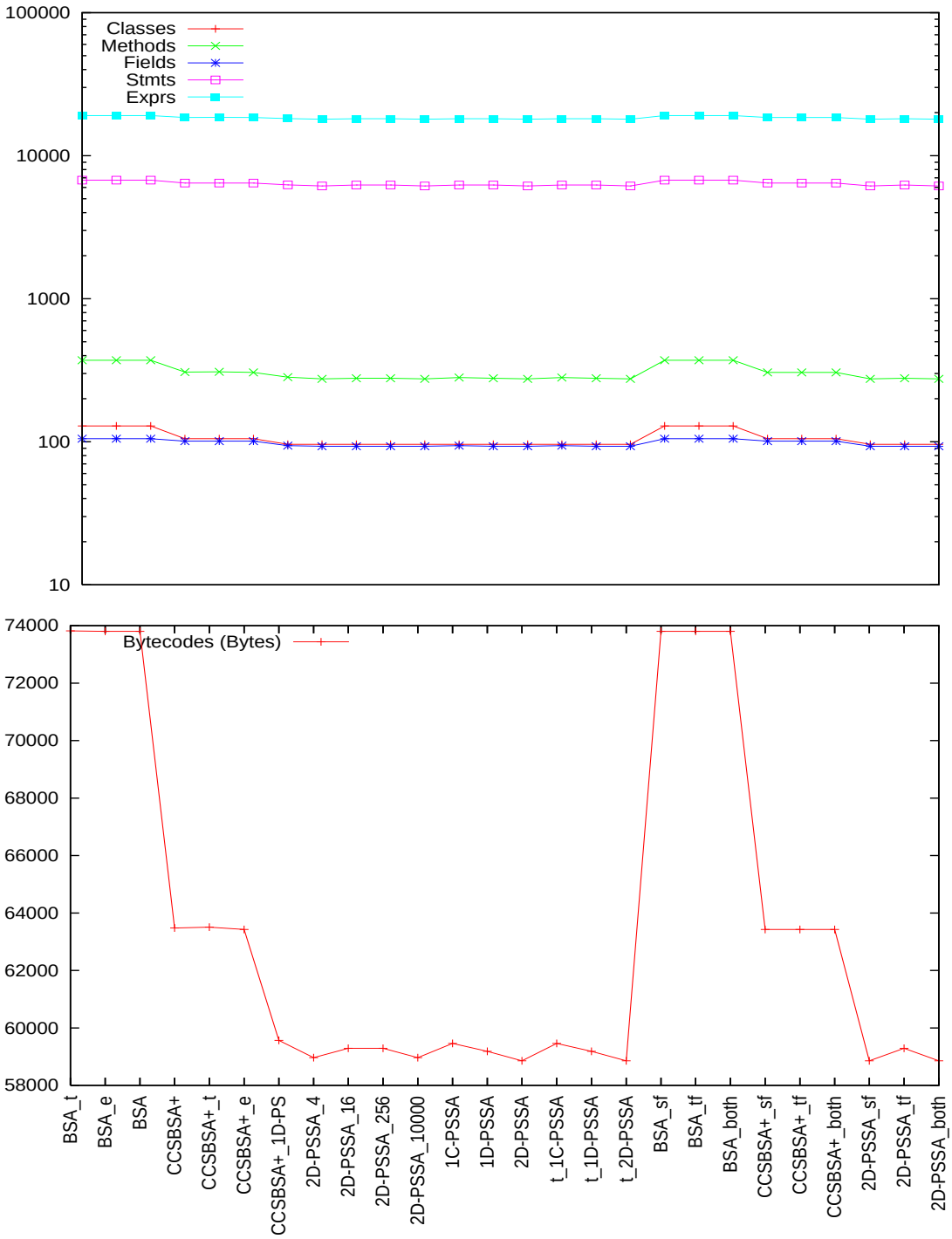

Figure A.2: Graphical representation of the data (Table A.2) from slicing *Crp* benchmark program from the Java Grande suite.

Configuration	Time (s)	Memory (MB)	Classes	Methods	Fields	Stmts	Exprs	Bytecodes (Bytes)
Unsliced			117	299	203	5731	16101	
BSA_t	1/13/73	3/20	114	318	78	5911	16589	64192
BSA_e	1/13/77	3/20	114	318	78	5911	16589	64189
BSA	1/13/75	3/20	114	318	78	5911	16589	64189
CCSBSA+	1/13/73	2/20	90	261	74	5621	16042	54035
CCSBSA+_t	1/13/72	2/20	90	261	74	5621	16042	54035
CCSBSA+_e	1/13/76	2/20	90	261	74	5621	16042	54035
CCSBSA+_1D-PS	1/15/79	2/20	81	239	67	5436	15699	50160
2D-PSSA_4	2/15/85	2/20	81	235	66	5424	15676	49883
2D-PSSA_16	2/15/84	2/20	81	235	66	5424	15676	49874
2D-PSSA_256	1/14/76	2/20	81	235	66	5424	15676	49874
2D-PSSA_10000	2/14/78	2/21	81	235	66	5424	15676	49874
1C-PSSA	2/14/74	2/21	81	239	67	5436	15699	50127
1D-PSSA	1/13/73	2/21	81	235	66	5424	15676	49871
2D-PSSA	1/13/73	2/21	81	232	66	5330	15523	49535
t_1C-PSSA	2/14/72	2/21	81	239	67	5436	15699	50127
t_1D-PSSA	2/13/71	2/21	81	235	66	5424	15676	49871
t_2D-PSSA	2/13/72	2/21	81	235	66	5424	15676	49871
BSA_sf	1/13/77	3/20	114	318	78	5911	16589	64189
BSA_tf	1/13/77	3/20	114	318	78	5911	16589	64189
BSA_both	1/13/75	3/20	114	318	78	5911	16589	64189
CCSBSA+_sf	1/13/73	2/20	90	261	74	5621	16042	54035
CCSBSA+_tf	1/13/77	2/20	90	261	74	5621	16042	54035
CCSBSA+_both	1/13/74	2/20	90	261	74	5621	16042	54035
2D-PSSA_sf	2/14/78	2/21	81	235	66	5424	15676	49871
2D-PSSA_tf	2/14/77	2/21	81	235	66	5424	15676	49874
2D-PSSA_both	1/13/73	2/21	81	235	66	5424	15676	49871

Table A.3: Data from slicing *FJ* benchmark program from the Java Grande suite.

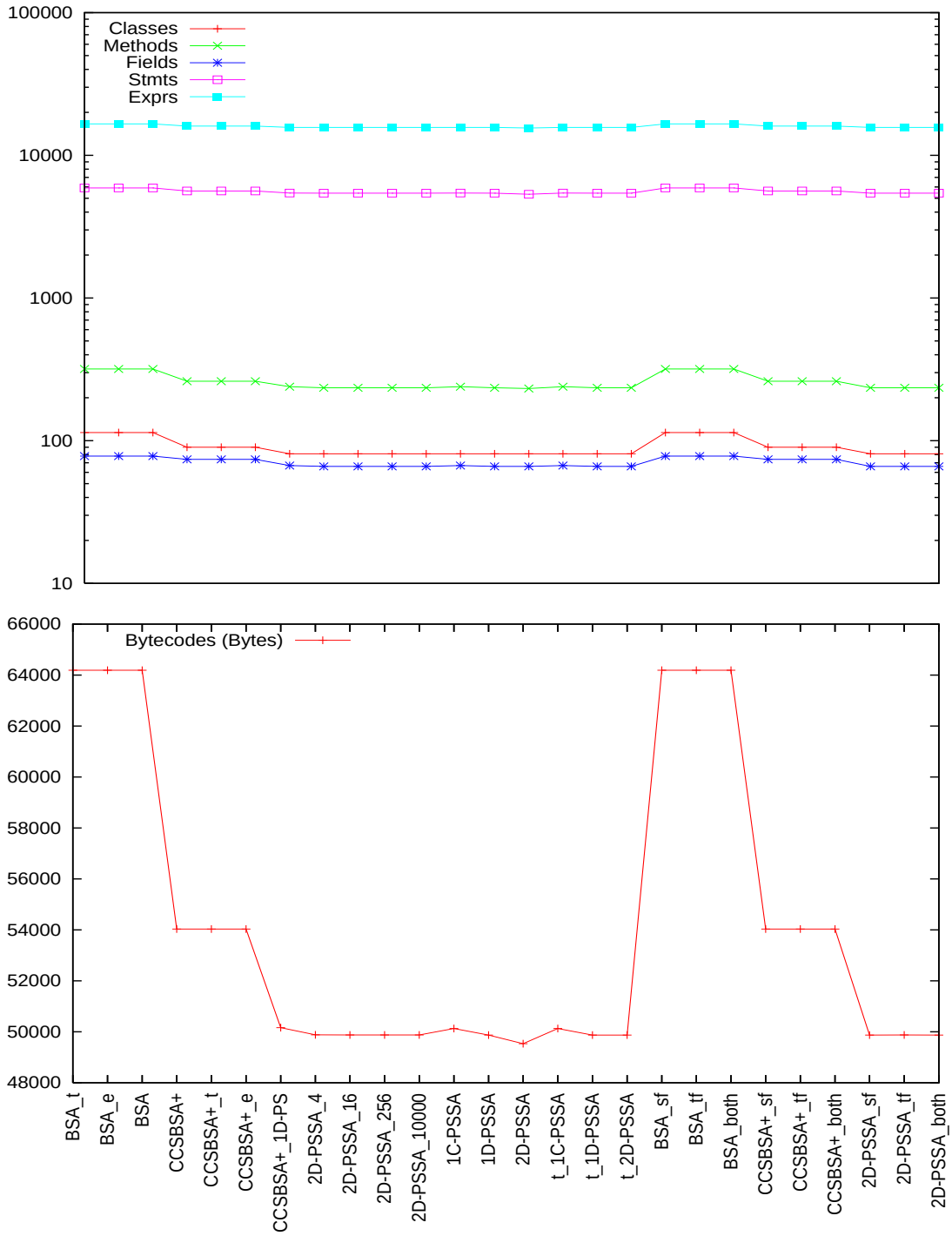

Figure A.3: Graphical representation of the data (Table A.3) from slicing *FJ* benchmark program from the Java Grande suite.

Configuration	Time (s)	Memory (MB)	Classes	Methods	Fields	Stmts	Exprs	Bytecodes (Bytes)
Unsliced			121	316	225	6477	18400	
BSA_t	1/13/74	3/23	118	339	100	6657	18888	68369
BSA_e	1/14/76	3/23	118	339	100	6657	18888	68329
BSA	1/13/78	3/23	118	339	100	6657	18888	68329
CCSBSA+	1/14/76	3/22	94	278	96	6353	18315	58076
CCSBSA+_t	1/13/74	3/22	94	280	96	6361	18333	58135
CCSBSA+_e	1/14/76	3/22	94	278	96	6353	18315	58076
CCSBSA+_1D-PS	1/14/76	3/23	85	256	89	6168	17972	54216
2D-PSSA_4	2/14/77	3/23	85	252	88	6156	17949	53944
2D-PSSA_16	2/14/76	3/23	85	252	88	6156	17949	53933
2D-PSSA_256	2/14/77	3/23	85	252	88	6156	17949	53933
2D-PSSA_10000	2/14/79	3/23	85	252	88	6156	17949	53921
1C-PSSA	2/15/78	3/23	85	256	89	6168	17972	54062
1D-PSSA	2/14/79	3/23	85	252	88	6156	17949	53918
2D-PSSA	2/14/77	3/23	85	249	88	6062	17796	53643
t_1C-PSSA	2/25/127	3/23	85	256	89	6168	17972	54050
t_1D-PSSA	2/21/147	3/23	85	252	88	6156	17949	53918
t_2D-PSSA	2/14/75	3/23	85	252	88	6156	17949	53918
BSA_sf	1/13/78	3/23	118	339	100	6657	18888	68329
BSA_tf	1/14/76	3/23	118	339	100	6657	18888	68329
BSA_both	1/14/76	3/23	118	339	100	6657	18888	68329
CCSBSA+_sf	1/14/76	3/23	94	278	96	6353	18315	58076
CCSBSA+_tf	1/14/76	3/22	94	278	96	6353	18315	58076
CCSBSA+_both	1/14/76	3/22	94	278	96	6353	18315	58076
2D-PSSA_sf	2/14/77	3/23	85	249	88	6062	17796	53643
2D-PSSA_tf	2/14/77	3/23	85	252	88	6156	17949	53933
2D-PSSA_both	2/14/78	3/23	85	249	88	6062	17796	53643

Table A.4: Data from slicing *LUF* benchmark program from the Java Grande suite.

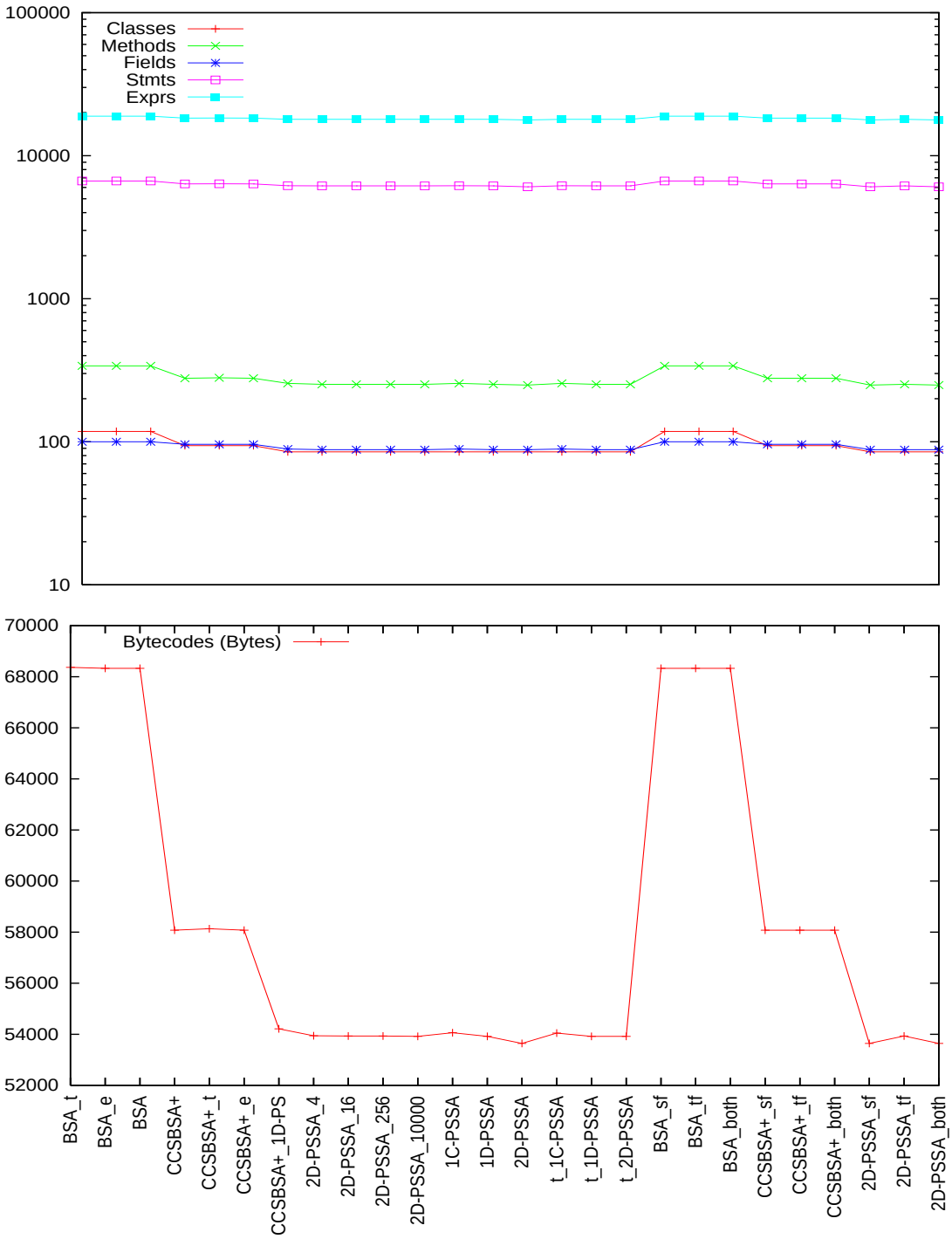

Figure A.4: Graphical representation of the data (Table A.4) from slicing *LUF* benchmark program from the Java Grande suite.

Configuration	Time (s)	Memory (MB)	Classes	Methods	Fields	Stmts	Exprs	Bytecodes (Bytes)
Unsliced			250	776	541	13934	37447	
BSA_t	5/27/137	10/59	250	868	299	14694	39326	165446
BSA_e	4/27/142	10/59	250	868	299	14694	39326	165419
BSA	4/27/140	9/59	250	868	299	14694	39326	165421
CCSBSA+	5/27/141	10/58	227	817	299	14451	38871	155121
CCSBSA+_t	5/27/139	10/59	227	815	299	14406	38736	155018
CCSBSA+_e	5/27/144	10/58	230	816	299	14411	38745	155958
CCSBSA+_1D-PS	5/27/145	10/61	226	816	298	14448	38865	154686
2D-PSSA_4	8/30/148	13/62	217	754	280	14083	38145	147125
2D-PSSA_16	8/30/148	12/61	220	769	284	14156	38281	149105
2D-PSSA_256	8/31/148	12/61	220	769	284	14156	38281	149105
2D-PSSA_10000	8/30/147	12/61	220	769	284	14156	38281	149105
1C-PSSA	12/35/154	13/68	224	788	292	14331	38658	151683
1D-PSSA	8/31/150	13/66	224	786	291	14325	38647	152175
2D-PSSA	8/31/151	13/67	220	755	280	14088	38154	147397
t_1C-PSSA	13/36/149	13/67	228	789	293	14336	38667	154004
t_1D-PSSA	8/32/147	14/67	228	787	292	14330	38656	153691
t_2D-PSSA	8/32/147	13/67	227	778	289	14276	38540	152607
BSA_sf	4/27/142	9/61	250	868	299	14694	39326	165421
BSA_tf	4/27/141	10/59	250	868	299	14694	39326	165421
BSA_both	4/27/143	9/61	250	868	299	14694	39326	165421
CCSBSA+_sf	5/27/143	9/61	227	817	299	14451	38871	155121
CCSBSA+_tf	5/27/143	10/58	230	818	299	14456	38880	156092
CCSBSA+_both	5/27/143	9/61	227	817	299	14451	38871	155121
2D-PSSA_sf	8/31/141	13/67	223	781	288	14254	38445	151719
2D-PSSA_tf	8/31/148	12/62	220	769	284	14156	38281	149108
2D-PSSA_both	8/31/150	12/68	221	768	283	14161	38293	149037

Table A.5: Data from slicing *MC* benchmark program from the Java Grande suite.

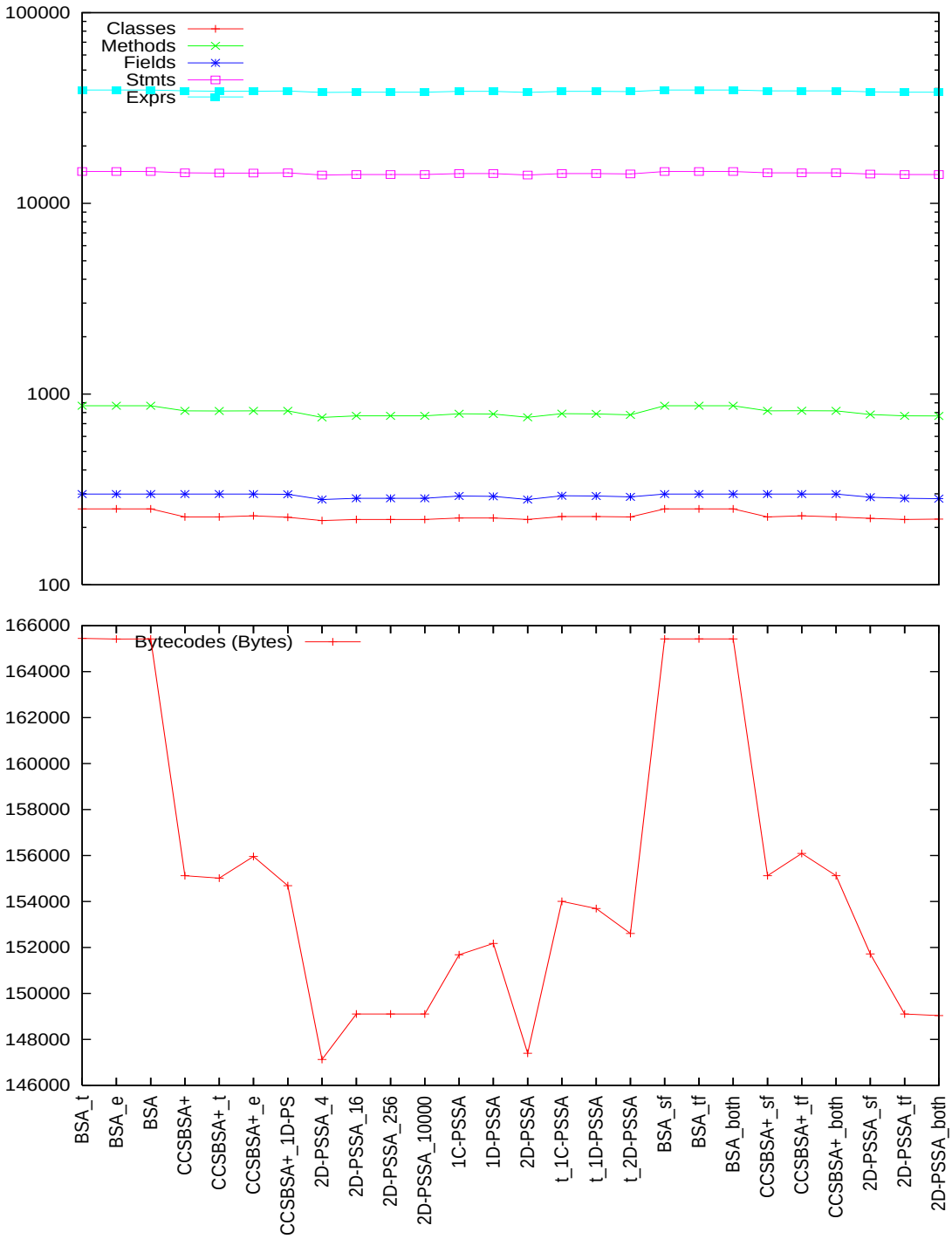

Figure A.5: Graphical representation of the data (Table A.5) from slicing *MC* benchmark program from the Java Grande suite.

Configuration	Time (s)	Memory (MB)	Classes	Methods	Fields	Stmts	Exprs	Bytecodes (Bytes)
Unsliced			123	324	283	7195	20647	
BSA_t	2/15/84	3/28	120	347	139	7375	21135	70563
BSA_e	2/16/86	3/28	120	347	139	7375	21135	70560
BSA	2/16/85	3/28	120	347	139	7375	21135	70560
CCSBSA+	2/15/85	3/28	96	286	134	7062	20531	60183
CCSBSA+_t	2/16/85	3/28	96	288	134	7070	20545	60236
CCSBSA+_e	2/16/86	3/28	96	286	134	7062	20531	60183
CCSBSA+_1D-PS	2/15/87	3/28	87	264	127	6877	20188	56322
2D-PSSA_4	3/17/89	3/28	87	260	126	6865	20165	56053
2D-PSSA_16	3/17/89	3/28	87	260	126	6865	20165	56042
2D-PSSA_256	3/17/89	3/28	87	260	126	6865	20165	56042
2D-PSSA_10000	3/17/89	3/28	87	260	126	6865	20165	56042
1C-PSSA	3/17/90	3/29	87	262	117	6873	20179	55377
1D-PSSA	3/17/89	3/29	87	260	126	6865	20165	56039
2D-PSSA	3/17/87	3/29	87	260	126	6865	20165	56039
t_1C-PSSA	4/18/88	3/29	87	264	127	6877	20188	56272
t_1D-PSSA	3/17/87	3/29	87	260	126	6865	20165	56039
t_2D-PSSA	3/17/87	3/29	87	257	126	6771	20012	55705
BSA_sf	2/16/86	3/28	120	347	139	7375	21135	70560
BSA_tf	2/16/85	3/28	120	347	139	7375	21135	70560
BSA_both	2/16/86	3/28	120	347	139	7375	21135	70560
CCSBSA+_sf	2/16/86	3/28	96	286	134	7062	20531	60183
CCSBSA+_tf	2/16/86	3/28	96	286	134	7062	20531	60183
CCSBSA+_both	2/16/86	3/28	96	286	134	7062	20531	60183
2D-PSSA_sf	3/17/90	3/28	87	257	126	6771	20012	55705
2D-PSSA_tf	3/17/89	3/28	87	260	126	6865	20165	56042
2D-PSSA_both	3/17/89	3/28	87	260	126	6865	20165	56039

Table A.6: Data from slicing *MD* benchmark program from the Java Grande suite.

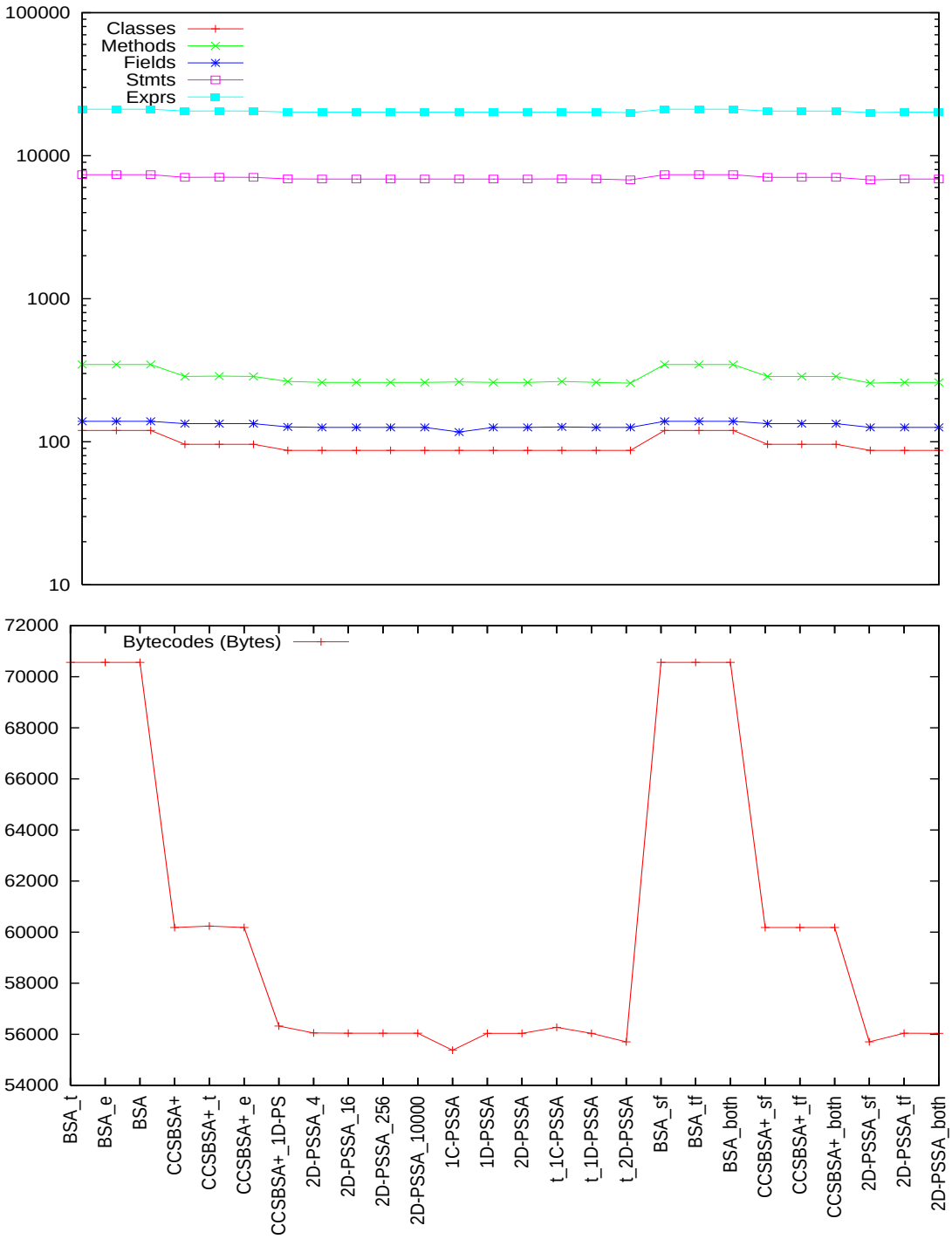

Figure A.6: Graphical representation of the data (Table A.6) from slicing *MD* benchmark program from the Java Grande suite.

Configuration	Time (s)	Memory (MB)	Classes	Methods	Fields	Stmts	Exprs	Bytecodes (Bytes)
Unsliced			132	363	266	6903	19422	
BSA_t	1/14/79	3/25	129	393	140	7087	19919	76501
BSA_e	1/14/83	3/25	129	393	140	7087	19919	76505
BSA	1/14/82	3/25	129	393	140	7087	19919	76523
CCSBSA+	1/14/82	3/24	105	332	136	6782	19343	66217
CCSBSA+_t	1/14/78	3/25	105	332	136	6782	19343	66217
CCSBSA+_e	1/15/80	3/24	105	332	136	6782	19343	66217
CCSBSA+_1D-PS	1/14/83	3/25	96	308	129	6593	18991	62328
2D-PSSA_4	2/15/84	3/25	96	304	128	6581	18968	62049
2D-PSSA_16	2/15/84	3/25	96	304	128	6581	18968	62040
2D-PSSA_256	2/15/84	3/25	96	304	128	6581	18968	62040
2D-PSSA_10000	2/15/84	3/25	96	304	128	6581	18968	62040
1C-PSSA	3/16/85	3/26	96	308	129	6593	18991	62293
1D-PSSA	2/15/84	3/26	96	304	128	6581	18968	62037
2D-PSSA	2/15/84	3/26	96	304	128	6581	18968	62042
t_1C-PSSA	3/16/83	3/26	96	308	128	6593	18991	62195
t_1D-PSSA	2/16/82	3/26	96	304	128	6581	18968	62037
t_2D-PSSA	2/16/82	3/26	96	301	128	6487	18815	61704
BSA_sf	1/14/82	3/25	129	393	140	7087	19919	76498
BSA_tf	1/14/83	3/25	129	393	140	7087	19919	76498
BSA_both	1/14/83	3/25	129	393	140	7087	19919	76498
CCSBSA+_sf	1/14/82	3/25	105	332	136	6782	19343	66217
CCSBSA+_tf	1/14/82	3/24	105	332	136	6782	19343	66217
CCSBSA+_both	1/14/83	3/25	105	332	136	6782	19343	66217
2D-PSSA_sf	2/15/83	3/26	96	304	128	6581	18968	62037
2D-PSSA_tf	2/15/84	3/25	96	301	128	6487	18815	61707
2D-PSSA_both	2/15/84	3/26	96	301	128	6487	18815	61704

Table A.7: Data from slicing *RT* benchmark program from the Java Grande suite.

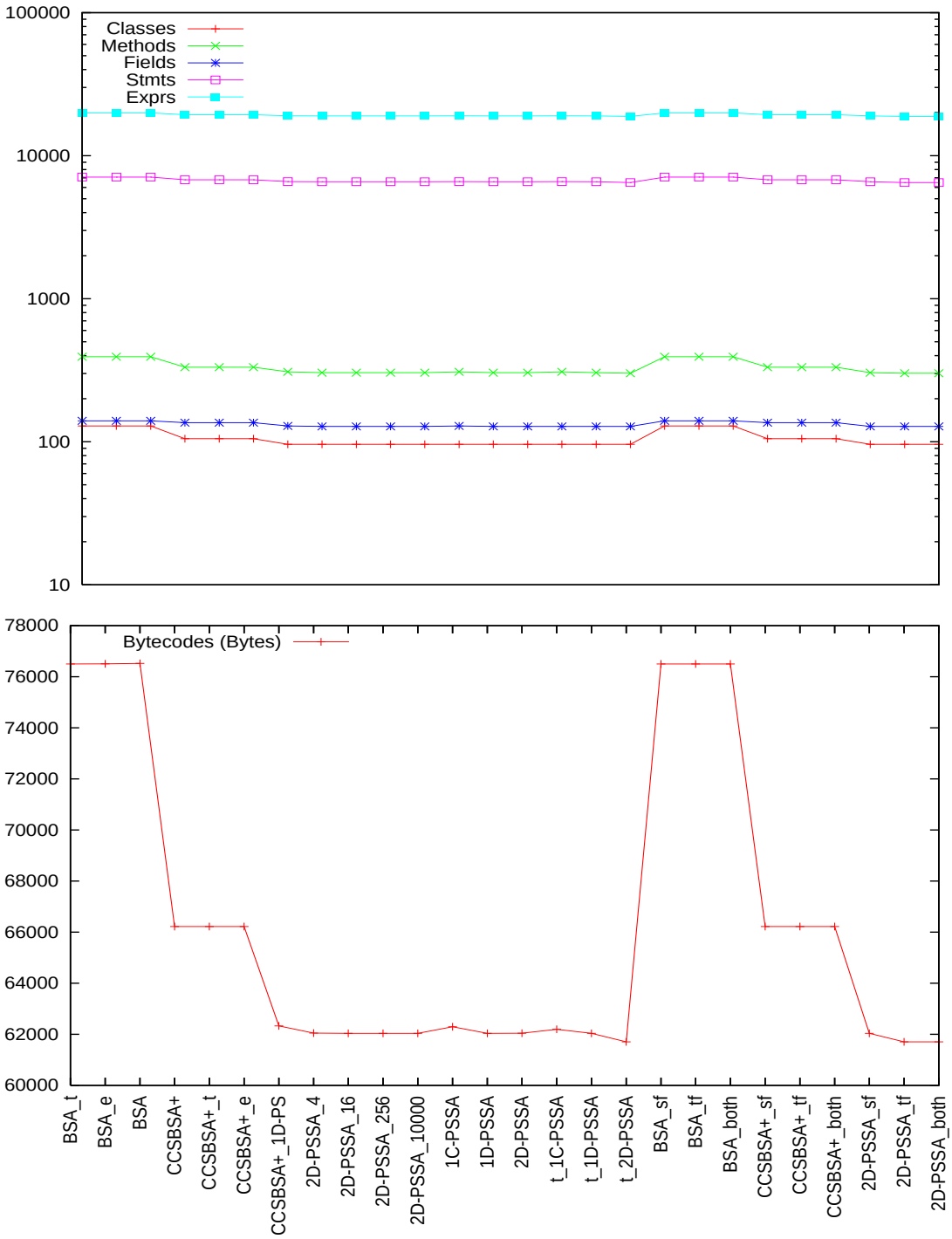

Figure A.7: Graphical representation of the data (Table A.7) from slicing *RT* benchmark program from the Java Grande suite.

Configuration	Time (s)	Memory (MB)	Classes	Methods	Fields	Stmts	Exprs	Bytecodes (Bytes)
Unsliced			119	313	208	5922	16701	
BSA_t	1/15/79	3/21	116	336	85	6102	17189	66183
BSA_e	1/15/85	3/21	116	336	85	6102	17189	66180
BSA	1/15/82	3/21	116	336	85	6102	17189	66180
CCSBSA+	1/15/82	2/21	92	279	81	5814	16648	56057
CCSBSA+_t	1/15/80	2/21	92	277	81	5806	16634	56004
CCSBSA+_e	1/15/85	2/21	92	277	81	5806	16634	56004
CCSBSA+_1D-PS	1/15/85	2/21	83	255	74	5621	16291	52143
2D-PSSA_4	2/16/86	2/21	83	251	73	5609	16268	51857
2D-PSSA_16	2/13/81	3/21	83	251	73	5609	16268	51847
2D-PSSA_256	2/13/76	2/21	83	251	73	5609	16268	51851
2D-PSSA_10000	2/14/77	3/21	83	251	73	5609	16268	51847
1C-PSSA	2/14/78	3/22	83	250	74	5609	16272	51988
1D-PSSA	2/14/77	2/21	83	248	73	5601	16256	51783
2D-PSSA	2/14/75	2/21	83	245	73	5507	16103	51449
t_1C-PSSA	2/14/73	3/22	83	250	74	5609	16272	51988
t_1D-PSSA	2/14/73	2/22	83	248	73	5601	16256	51783
t_2D-PSSA	2/14/73	2/22	83	248	73	5601	16256	51783
BSA_sf	1/13/76	3/21	116	336	85	6102	17189	66180
BSA_tf	1/13/74	3/21	116	336	85	6102	17189	66180
BSA_both	1/13/76	3/21	116	336	85	6102	17189	66180
CCSBSA+_sf	1/13/75	2/21	92	279	81	5814	16648	56057
CCSBSA+_tf	1/13/74	2/21	92	277	81	5806	16634	56004
CCSBSA+_both	1/13/74	2/21	92	279	81	5814	16648	56057
2D-PSSA_sf	2/13/77	2/21	83	245	73	5507	16103	51449
2D-PSSA_tf	2/14/77	3/21	83	251	73	5609	16268	51847
2D-PSSA_both	2/14/77	2/21	83	245	73	5507	16103	51449

Table A.8: Data from slicing *Ser* benchmark program from the Java Grande suite.

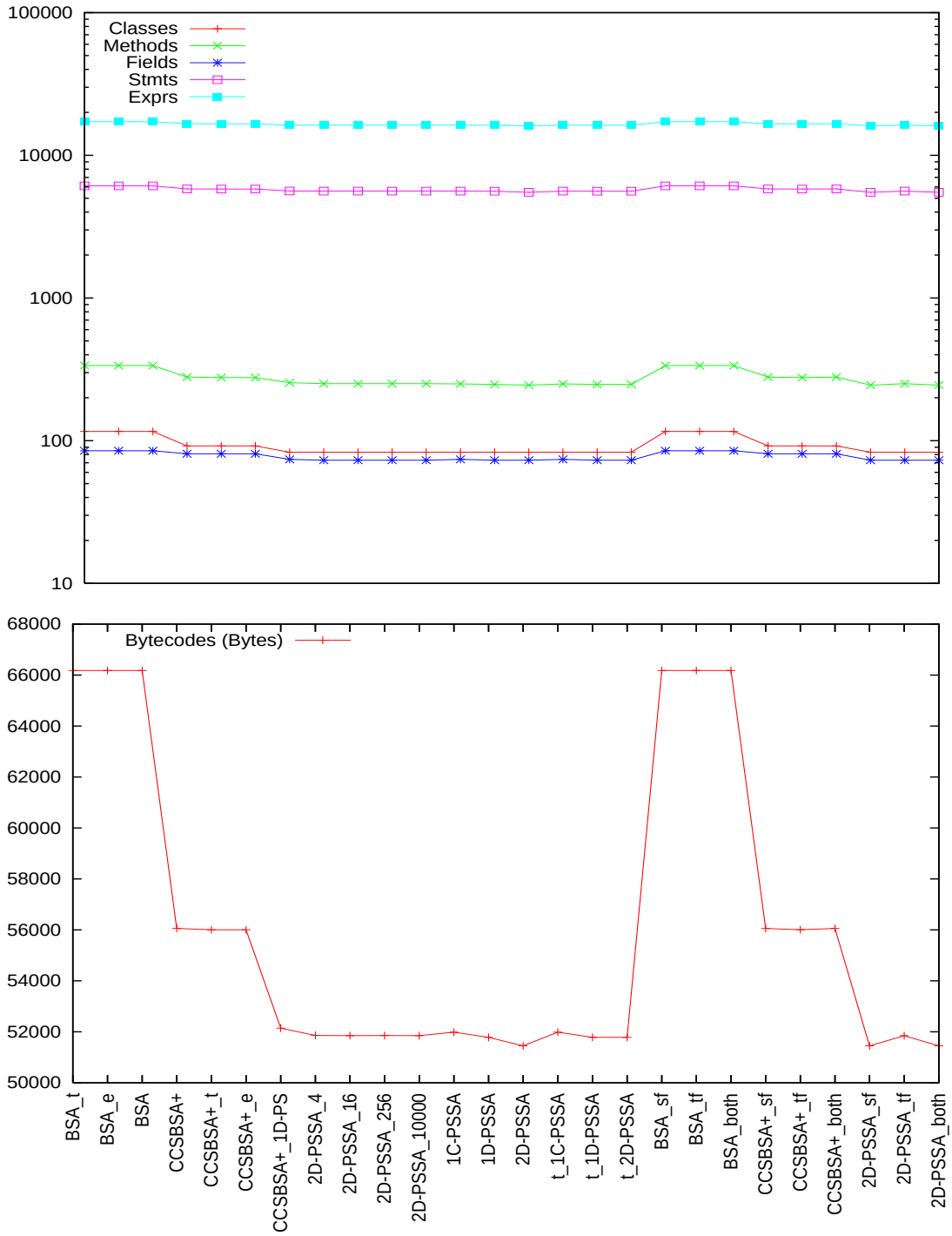

Figure A.8: Graphical representation of the data (Table A.8) from slicing *Ser* benchmark program from the Java Grande suite.

Configuration	Time (s)	Memory (MB)	Classes	Methods	Fields	Stmts	Exprs	Bytecodes (Bytes)
Unsliced			133	337	246	6328	17755	
BSA_t	1/14/75	3/23	129	372	114	6566	18391	73666
BSA_e	1/13/74	3/23	129	372	114	6566	18391	73663
BSA	1/14/76	3/23	129	372	114	6566	18391	73663
CCSBSA+	1/14/78	3/23	105	310	110	6264	17830	63367
CCSBSA+_t	1/14/75	3/23	105	310	110	6264	17830	63367
CCSBSA+_e	1/13/75	3/23	105	310	110	6264	17830	63367
CCSBSA+_1D-PS	1/14/77	3/23	96	286	103	6071	17473	59454
2D-PSSA_4	2/14/77	3/23	96	282	96	6059	17450	58864
2D-PSSA_16	2/14/75	3/23	96	282	96	6059	17450	58853
2D-PSSA_256	2/14/77	3/23	96	282	96	6059	17450	58853
2D-PSSA_10000	2/14/78	3/23	96	282	96	6059	17450	58853
1C-PSSA	2/15/79	3/24	96	286	97	6071	17473	59100
1D-PSSA	2/15/79	3/24	96	282	96	6059	17450	58759
2D-PSSA	2/15/78	3/23	96	279	96	5965	17297	58429
t_1C-PSSA	2/15/77	3/24	96	286	97	6071	17473	59100
t_1D-PSSA	2/15/77	3/24	96	282	96	6059	17450	58759
t_2D-PSSA	2/14/77	3/24	96	282	96	6059	17450	58764
BSA_sf	1/14/79	3/23	129	372	114	6566	18391	73663
BSA_tf	1/14/80	3/23	129	372	114	6566	18391	73663
BSA_both	1/13/79	3/23	129	372	114	6566	18391	73663
CCSBSA+_sf	1/13/79	3/23	105	310	110	6264	17830	63367
CCSBSA+_tf	1/13/78	3/23	105	310	110	6264	17830	63367
CCSBSA+_both	1/13/79	3/23	105	310	110	6264	17830	63367
2D-PSSA_sf	2/14/76	3/23	96	282	96	6059	17450	58764
2D-PSSA_tf	2/15/78	3/23	96	282	96	6059	17450	58853
2D-PSSA_both	2/15/79	3/23	96	279	96	5965	17297	58429

Table A.9: Data from slicing *SMM* benchmark program from the Java Grande suite.

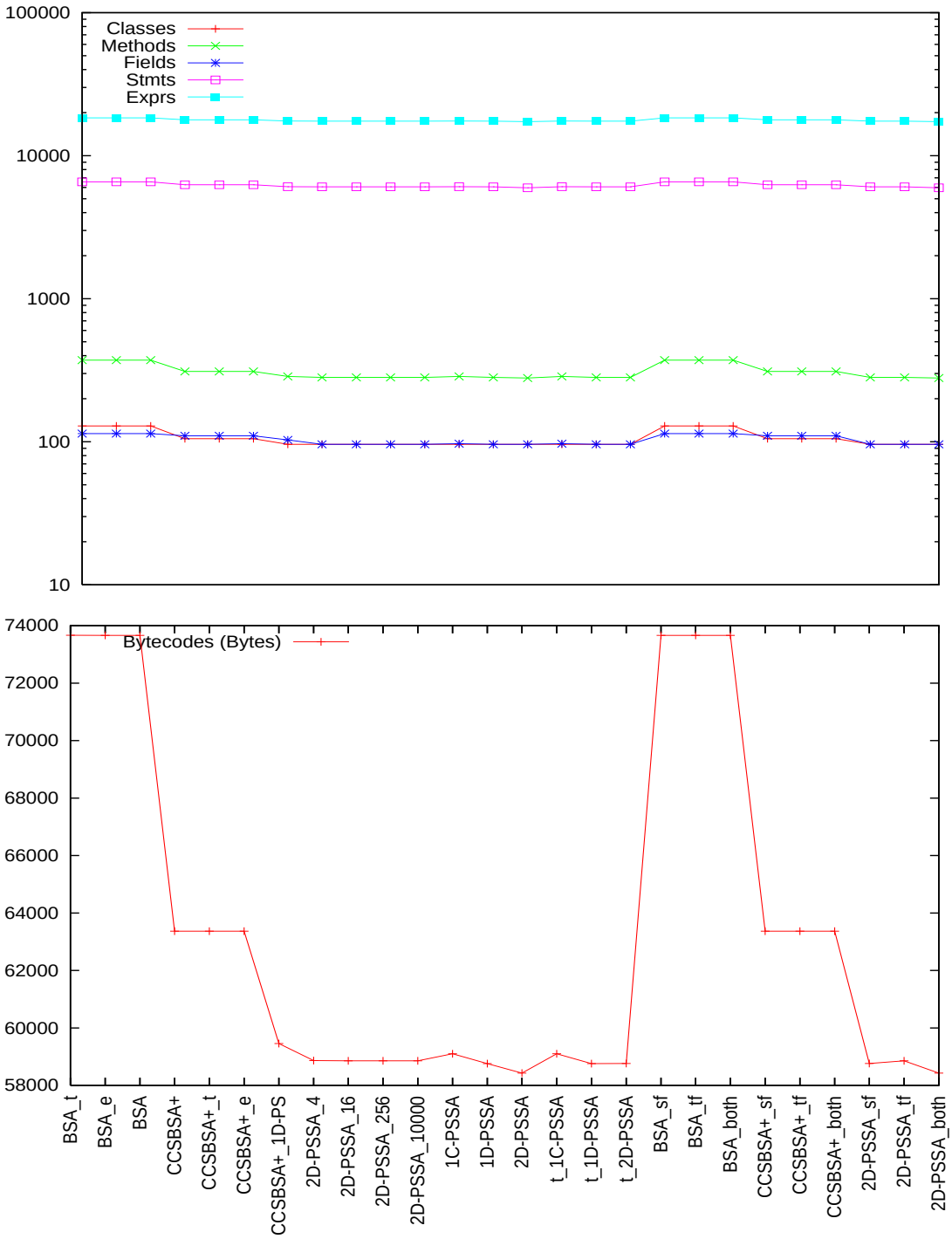

Figure A.9: Graphical representation of the data (Table A.9) from slicing *SMM* benchmark program from the Java Grande suite.

Configuration	Time (s)	Memory (MB)	Classes	Methods	Fields	Stmts	Exprs	Bytecodes (Bytes)
Unsliced			133	335	233	6264	17627	
BSA_t	1/13/74	3/23	129	368	101	6500	18261	72790
BSA_e	1/15/75	3/22	129	368	101	6500	18261	72787
BSA	1/15/83	3/22	129	368	101	6500	18261	72787
CCSBSA+	1/15/83	3/22	105	306	97	6198	17700	62486
CCSBSA+_t	1/15/79	3/22	105	306	97	6198	17700	62486
CCSBSA+_e	1/15/82	3/22	105	306	97	6198	17700	62486
CCSBSA+_1D-PS	1/15/81	3/23	96	282	90	6005	17343	58573
2D-PSSA_4	2/16/82	3/23	96	278	88	5993	17320	58254
2D-PSSA_16	2/16/82	3/23	96	278	88	5993	17320	58247
2D-PSSA_256	2/16/83	3/23	96	278	88	5993	17320	58247
2D-PSSA_10000	2/16/83	3/23	96	278	88	5993	17320	58247
1C-PSSA	3/17/84	3/23	96	282	89	6005	17343	58391
1D-PSSA	2/16/84	3/23	96	278	88	5993	17320	58138
2D-PSSA	2/16/83	3/23	96	278	88	5993	17320	58138
t_1C-PSSA	2/16/84	3/23	96	282	90	6005	17343	58535
t_1D-PSSA	2/16/84	3/23	96	278	89	5993	17320	58199
t_2D-PSSA	2/16/83	3/23	96	278	89	5993	17320	58199
BSA_sf	1/15/84	3/23	129	368	101	6500	18261	72787
BSA_tf	1/15/84	3/22	129	368	101	6500	18261	72787
BSA_both	1/15/85	3/23	129	368	101	6500	18261	72787
CCSBSA+_sf	1/15/84	3/22	105	306	97	6198	17700	62486
CCSBSA+_tf	1/15/85	3/22	105	306	97	6198	17700	62486
CCSBSA+_both	1/15/85	3/22	105	306	97	6198	17700	62486
2D-PSSA_sf	2/16/83	3/23	96	278	89	5993	17320	58188
2D-PSSA_tf	2/16/83	3/23	96	278	88	5993	17320	58247
2D-PSSA_both	2/16/84	3/23	96	278	88	5993	17320	58138

Table A.10: Data from slicing *SOR* benchmark program from the Java Grande suite.

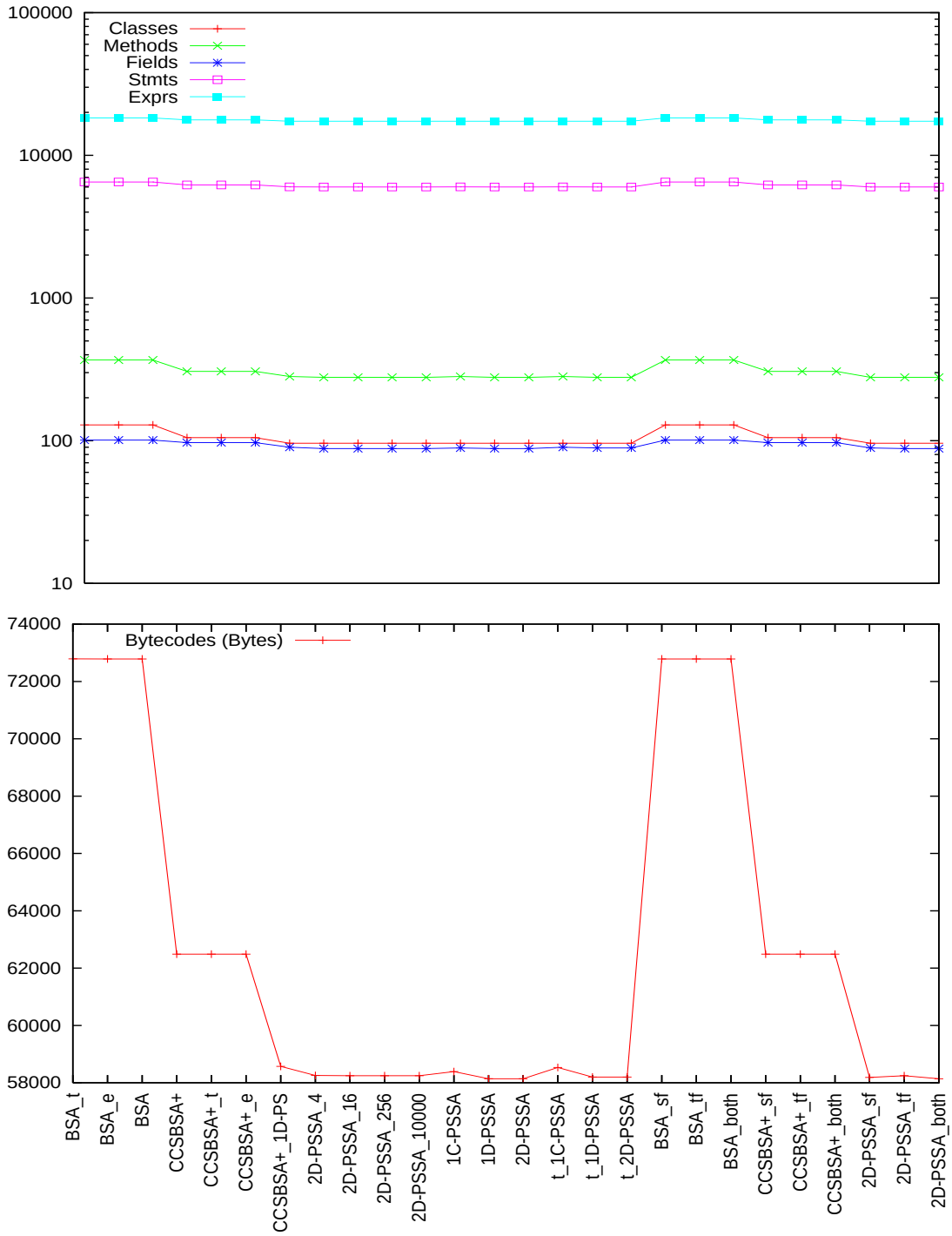

Figure A.10: Graphical representation of the data (Table A.10) from slicing *SOR* benchmark program from the Java Grande suite.

Configuration	Time (s)	Memory (MB)	Classes	Methods	Fields	Stmts	Exprs	Bytecodes (Bytes)
Unsliced			119	297	209	5742	16115	
BSA_t	1/12/68	3/20	116	315	83	5922	16603	65076
BSA_e	1/13/75	3/20	116	315	83	5922	16603	65074
BSA	1/12/69	3/20	116	315	83	5922	16603	65074
CCSBSA+	1/12/69	2/20	92	258	79	5632	16056	54928
CCSBSA+_t	1/13/70	2/20	92	258	79	5632	16056	54928
CCSBSA+_e	1/13/74	2/20	92	258	79	5632	16056	54928
CCSBSA+_1D-PS	1/12/69	2/20	83	236	72	5447	15713	51066
2D-PSSA_4	1/12/69	2/21	83	233	71	5441	15702	50843
2D-PSSA_16	1/12/69	2/21	83	233	71	5441	15702	50843
2D-PSSA_256	1/12/69	2/21	83	233	71	5441	15702	50843
2D-PSSA_10000	1/13/74	2/21	83	233	71	5441	15702	50843
1C-PSSA	2/14/76	2/21	83	236	72	5447	15713	51032
1D-PSSA	1/13/75	2/21	83	233	71	5441	15702	50840
2D-PSSA	1/13/75	2/21	83	230	71	5347	15549	50506
t_1C-PSSA	2/14/72	3/21	83	236	72	5447	15713	51032
t_1D-PSSA	2/13/71	2/21	83	233	71	5441	15702	50840
t_2D-PSSA	2/13/71	2/21	83	230	71	5347	15549	50506
BSA_sf	1/13/74	3/21	116	315	83	5922	16603	65074
BSA_tf	1/13/74	3/20	116	315	83	5922	16603	65074
BSA_both	1/13/75	3/21	116	315	83	5922	16603	65074
CCSBSA+_sf	1/12/74	2/20	92	258	79	5632	16056	54928
CCSBSA+_tf	1/13/74	2/20	92	258	79	5632	16056	54928
CCSBSA+_both	1/12/74	2/20	92	258	79	5632	16056	54928
2D-PSSA_sf	1/13/75	2/21	83	230	71	5347	15549	50506
2D-PSSA_tf	1/13/75	2/21	83	233	71	5441	15702	50843
2D-PSSA_both	1/13/75	2/21	83	230	71	5347	15549	50506

Table A.11: Data from slicing *Syn* benchmark program from the Java Grande suite.

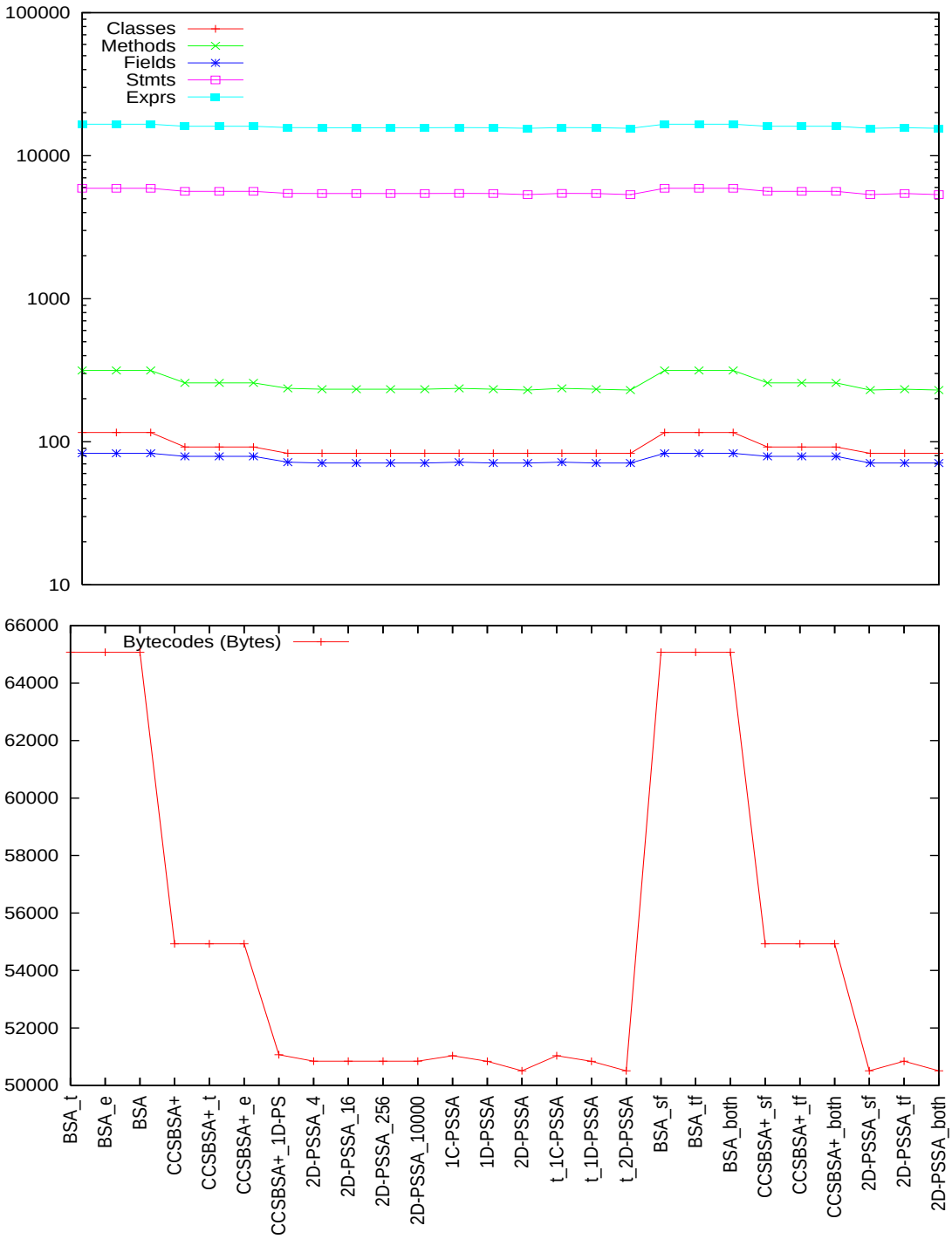

Figure A.11: Graphical representation of the data (Table A.11) from slicing *Syn* benchmark program from the Java Grande suite.

Appendix B

Data From POR Experiments

The data collected in the partial order reduction related experiments are presented. Please refer to Section 6.4 for the conclusions drawn from the experiments.

B.1 Data Description

For each input program and each configuration, the raw exploration data was collected in terms (aspects) of the number of explored states *(States)*, matched states *(Matched)*, explored transitions *(Transitions)* and uncovered errors *(Errors)*. The total time *(Time)* and maximum memory *(Memory)* required to execute the entire analysis and model checking pipeline was also recored. In the data for each input program, this data is presented in the (a) table.

To determine the relative merit of the proposed approach, the data from the non-E configurations were compared with the data from the E configuration by subtracting the former from the latter, e.g. *States* in SC was subtracted from *States* in E. In the data for each input program, this data is presented in the (b) table.

While the above evaluation is useful to assess the relative merit of the approaches at a system level, it is insufficient to evaluate the approaches at a micro (state) level. Further, it is insufficient to evaluate the approaches in case of terminated experiments. Hence, to facilitate a local and complimentary evaluation, local (reduction) data at the state level was collected. This data contains the total number of enabled *(E)*, ample *(A)*, and independent *(I)* transitions encountered at the visited states of the system. Further, the weighted average of the ratio of ample and enabled transition *(A/E)* at each visited states of the system was also collected. In case of configurations involving SDPOR, the total number of dynamic dependences discovered merely via static dependence information *(S)* and via pure dynamic approach *(D)* was collected and the ratio of between these numbers *(D/S)* was calculated. In the data for each input program, this data is presented in the (c) table.

The data is separated based on the completion of execution of at least one of the configuration.

B.2 Completed Configurations

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	10128	2163	161172	114976	2617743	5531
SD	1574	2228	47522	14334	619812	250
SC+E	7351	3103	78869	52419	1471300	1982
E+SD	588	3124	9240	1441	149699	77
E	36013	4209	256479	415793	5599303	14668
SC+F	36052	6178	35922	26061	506774	3650
SD+F	36065	6280	35478	12167	425965	3476
SC+F+E	36049	6905	18928	13901	363324	3727
E+SD+F	11151	7414	8248	2324	125174	1144

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	-25885	-2046	-95307	-300817	-2981560	-9137
SD	-34439	-1981	-208957	-401459	-4979491	-14418
SC+E	-28662	-1106	-177610	-363374	-4128003	-12686
E+SD	-35425	-1085	-247239	-414352	-5449604	-14591
SC+F	39	1969	-220557	-389732	-5092529	-11018
SD+F	52	2071	-221001	-403626	-5173338	-11192
SC+F+E	36	2696	-237551	-401892	-5235979	-10941
E+SD+F	-24862	3205	-248231	-413469	-5474129	-13524

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	6050668	2617641	2340181	3710487	277460	0.09	0	0	0
SD	1438791	614811	560289	878502	54522	0.09	854233	4981	0.01
SC+E	3086258	1476912	1321733	1764525	155179	0.12	0	0	0
E+SD	165678	148590	136662	29016	11928	0.90	84256	1111	0.01
SC+F	1123359	504036	445600	677759	58436	0.12	0	0	0
SD+F	946075	419355	382243	563832	37112	0.10	504248	3178	0.01
SC+F+E	752479	361384	328420	424059	32964	0.14	0	0	0
E+SD+F	135661	123174	112860	22801	10314	0.90	76378	885	0.01

(c)

Table B.1: The raw and EPOR-relative exploration data and reduction data from alarm clock AC1 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	684	2079	18051	11475	443403	0
SD	474	1016	12766	5099	273092	0
SC+E	442	1009	5568	3889	165798	0
E+SD	147	516	1857	485	38522	0
E	476	769	11889	13067	418901	0
SC+F	1182	6311	18051	11475	443403	0
SD+F	830	7184	12766	5099	273092	0
SC+F+E	691	7243	5568	3889	165798	0
E+SD+F	297	7113	1857	485	38522	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	208	1310	6162	-1592	24502	0
SD	-2	247	877	-7968	-145809	0
SC+E	-34	240	-6321	-9178	-253103	0
E+SD	-329	-253	-10032	-12582	-380379	0
SC+F	706	5542	6162	-1592	24502	0
SD+F	354	6415	877	-7968	-145809	0
SC+F+E	215	6474	-6321	-9178	-253103	0
E+SD+F	-179	6344	-10032	-12582	-380379	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	967950	443359	396090	571860	47269	0.18	0	0	0
SD	624603	272112	250020	374583	22092	0.11	364679	987	0.00
SC+E	280233	167247	138646	141587	28601	0.40	0	0	0
E+SD	41470	38179	33896	7574	4283	0.92	37811	340	0.01
SC+F	967950	443359	396090	571860	47269	0.18	0	0	0
SD+F	624603	272112	250020	374583	22092	0.11	364679	987	0.00
SC+F+E	280233	167247	138646	141587	28601	0.40	0	0	0
E+SD+F	41470	38179	33896	7574	4283	0.92	37811	340	0.01

(c)

Table B.2: The raw and EPOR-relative exploration data and reduction data from alarm clock AC2 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	844	1013	27470	21477	828223	0
SD	566	1419	16053	7469	482608	0
SC+E	425	779	6815	4863	238414	0
E+SD	138	578	1946	342	55827	0
E	424	495	11030	11239	499105	0
SC+F	966	1774	27470	21477	828223	0
SD+F	622	1865	16053	7469	482608	0
SC+F+E	443	1859	6815	4863	238414	0
E+SD+F	152	1698	1946	342	55827	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	420	518	16440	10238	329118	0
SD	142	924	5023	-3770	-16497	0
SC+E	1	284	-4215	-6376	-260691	0
E+SD	-286	83	-9084	-10897	-443278	0
SC+F	542	1279	16440	10238	329118	0
SD+F	198	1370	5023	-3770	-16497	0
SC+F+E	19	1364	-4215	-6376	-260691	0
E+SD+F	-272	1203	-9084	-10897	-443278	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	1868586	828060	757631	1110955	70429	0.14	0	0	0
SD	1127213	479693	453561	673652	26132	0.08	565788	2878	0.01
SC+E	419007	240040	206271	212736	33769	0.33	0	0	0
E+SD	60114	55624	51214	8900	4410	0.92	22137	200	0.01
SC+F	1868586	828060	757631	1110955	70429	0.14	0	0	0
SD+F	1127213	479693	453561	673652	26132	0.08	565788	2878	0.01
SC+F+E	419007	240040	206271	212736	33769	0.33	0	0	0
E+SD+F	60114	55624	51214	8900	4410	0.92	22137	200	0.01

(c)

Table B.3: The raw and EPOR-relative exploration data and reduction data from alarm clock AC3 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	69	594	1586	835	7772	0
SD	56	934	929	309	4677	0
SC+E	45	1197	794	413	5576	0
E+SD	46	1144	644	233	4493	0
E	42	94	90	90	2448	0
SC+F	49	1252	1586	838	7775	0
SD+F	49	98	929	343	4711	0
SC+F+E	52	1252	794	420	5583	0
E+SD+F	49	1148	644	265	4525	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	27	500	1496	745	5324	0
SD	14	840	839	219	2229	0
SC+E	3	1103	704	323	3128	0
E+SD	4	1050	554	143	2045	0
SC+F	7	1158	1496	748	5327	0
SD+F	7	4	839	253	2263	0
SC+F+E	10	1158	704	330	3135	0
E+SD+F	7	1054	554	175	2077	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	11059	7771	4196	6863	3575	0.54	0	0	0
SD	6723	4476	2610	4113	1866	0.49	95171	200	0.00
SC+E	7720	5642	3133	4587	2509	0.60	0	0	0
E+SD	4974	4324	2558	2416	1766	0.85	95002	168	0.00
SC+F	11059	7771	4196	6863	3575	0.54	0	0	0
SD+F	6723	4476	2610	4113	1866	0.49	97550	200	0.00
SC+F+E	7720	5642	3133	4587	2509	0.60	0	0	0
E+SD+F	4974	4324	2558	2416	1766	0.85	95544	168	0.00

(c)

Table B.4: The raw and EPOR-relative exploration data and reduction data from bounded buffer BB1 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	66	60	46	27	301	5
SD	60	73	44	25	282	4
SC+E	51	93	7	1	101	2
E+SD	50	93	5	0	83	1
E	67	92	122	88	869	32
SC+F	158	100	257	242	727	216
SD+F	157	98	255	239	707	215
SC+F+E	90	100	109	104	306	104
E+SD+F	76	100	50	45	230	46

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	-1	-32	-76	-61	-568	-27
SD	-7	-19	-78	-63	-587	-28
SC+E	-16	1	-115	-87	-768	-30
E+SD	-17	1	-117	-88	-786	-31
SC+F	91	8	135	154	-142	184
SD+F	90	6	133	151	-162	183
SC+F+E	23	8	-13	16	-563	72
E+SD+F	9	8	-72	-43	-639	14

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	489	295	197	292	98	0.43	0	0	0
SD	457	277	182	275	95	0.45	0	0	0
SC+E	165	109	77	88	32	0.48	0	0	0
E+SD	86	81	62	24	19	0.94	0	0	0
SC+F	1014	506	408	606	98	0.24	0	0	0
SD+F	982	488	393	589	95	0.25	286	0	0
SC+F+E	368	211	179	189	32	0.24	0	0	0
E+SD+F	188	183	164	24	19	0.97	22	0	0

(c)

Table B.5: The raw and EPOR-relative exploration data and reduction data from bounded buffer BB4 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	60	59	152	56	810	0
SD	58	72	149	52	796	0
SC+E	50	91	21	13	379	0
E+SD	48	91	23	10	408	0
E	52	91	28	28	507	0
SC+F	44	92	152	56	810	0
SD+F	48	92	149	52	796	0
SC+F+E	44	92	21	14	380	0
E+SD+F	50	92	23	15	413	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	8	-32	124	28	303	0
SD	6	-19	121	24	289	0
SC+E	-2	0	-7	-15	-128	0
E+SD	-4	0	-5	-18	-99	0
SC+F	-8	1	124	28	303	0
SD+F	-4	1	121	24	289	0
SC+F+E	-8	1	-7	-14	-127	0
E+SD+F	-2	1	-5	-13	-94	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	1054	809	397	657	412	0.69	0	0	0
SD	1032	789	392	640	397	0.68	2546	6	0.00
SC+E	462	386	199	263	187	0.80	0	0	0
E+SD	423	402	211	212	191	0.95	1088	5	0.00
SC+F	1054	809	397	657	412	0.69	0	0	0
SD+F	1032	789	392	640	397	0.68	2546	6	0.00
SC+F+E	462	386	199	263	187	0.80	0	0	0
E+SD+F	423	402	211	212	191	0.95	1124	5	0.00

(c)

Table B.6: The raw and EPOR-relative exploration data and reduction data from bounded buffer BB8 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	52	60	173	83	665	2
SD	52	72	113	43	388	2
SC+E	53	92	115	43	491	2
E+SD	48	91	89	27	324	2
E	50	91	178	114	809	2
SC+F	39	92	173	83	665	2
SD+F	53	92	135	56	512	2
SC+F+E	39	92	115	43	491	2
E+SD+F	43	92	111	40	448	2

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	2	-31	-5	-31	-144	0
SD	2	-19	-65	-71	-421	0
SC+E	3	1	-63	-71	-318	0
E+SD	-2	0	-89	-87	-485	0
SC+F	-11	1	-5	-31	-144	0
SD+F	3	1	-43	-58	-297	0
SC+F+E	-11	1	-63	-71	-318	0
E+SD+F	-7	1	-67	-74	-361	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	930	664	348	582	316	0.59	0	0	0
SD	532	352	191	341	161	0.49	368	35	0.10
SC+E	662	494	265	397	229	0.63	0	0	0
E+SD	372	292	168	204	124	0.73	244	31	0.13
SC+F	930	664	348	582	316	0.59	0	0	0
SD+F	693	463	262	431	201	0.50	511	44	0.09
SC+F+E	662	494	265	397	229	0.63	0	0	0
E+SD+F	503	403	239	264	164	0.75	387	40	0.10

(c)

Table B.7: The raw and EPOR-relative exploration data and reduction data from deadlock DL1 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	75	59	47	20	182	1
SD	48	71	34	11	132	1
SC+E	44	91	8	1	77	1
E+SD	53	91	8	1	77	1
E	40	90	8	1	77	1
SC+F	47	91	47	20	182	1
SD+F	37	91	34	11	132	1
SC+F+E	37	91	8	1	77	1
E+SD+F	37	91	8	1	77	1

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	35	-31	39	19	105	0
SD	8	-19	26	10	55	0
SC+E	4	1	0	0	0	0
E+SD	13	1	0	0	0	0
SC+F	7	1	39	19	105	0
SD+F	-3	1	26	10	55	0
SC+F+E	-3	1	0	0	0	0
E+SD+F	-3	1	0	0	0	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	268	181	107	161	74	0.55	0	0	0
SD	186	128	76	110	52	0.59	99	3	0.03
SC+E	102	80	50	52	30	0.74	0	0	0
E+SD	80	74	50	30	24	0.93	19	2	0.11
SC+F	268	181	107	161	74	0.55	0	0	0
SD+F	186	128	76	110	52	0.59	99	3	0.03
SC+F+E	102	80	50	52	30	0.74	0	0	0
E+SD+F	80	74	50	30	24	0.93	19	2	0.11

(c)

Table B.8: The raw and EPOR-relative exploration data and reduction data from deadlock DL2 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	64	59	61	24	198	2
SD	47	71	43	14	133	1
SC+E	46	91	7	0	86	2
E+SD	54	91	4	0	65	1
E	41	90	7	0	86	2
SC+F	40	91	61	24	198	2
SD+F	39	91	46	14	154	2
SC+F+E	39	91	7	0	86	2
E+SD+F	50	91	7	0	86	2

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	23	-31	54	24	112	0
SD	6	-19	36	14	47	-1
SC+E	5	1	0	0	0	0
E+SD	13	1	-3	0	-21	-1
SC+F	-1	1	54	24	112	0
SD+F	-2	1	39	14	68	0
SC+F+E	-2	1	0	0	0	0
E+SD+F	9	1	0	0	0	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	282	197	89	193	108	0.58	0	0	0
SD	198	132	62	136	70	0.52	64	0	0
SC+E	115	91	50	65	41	0.75	0	0	0
E+SD	69	64	40	29	24	0.94	5	0	0
SC+F	282	197	89	193	108	0.58	0	0	0
SD+F	219	152	72	147	80	0.58	87	0	0
SC+F+E	115	91	50	65	41	0.75	0	0	0
E+SD+F	89	84	50	39	34	0.95	14	0	0

(c)

Table B.9: The raw and EPOR-relative exploration data and reduction data from deadlock DL3 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	54	533	746	792	2426	1
SD	57	71	304	272	925	1
SC+E	39	91	30	36	266	1
E+SD	50	91	16	4	173	1
E	47	91	30	57	287	1
SC+F	40	117	746	800	2434	1
SD+F	50	117	304	274	927	1
SC+F+E	42	117	30	41	271	1
E+SD+F	41	117	16	6	175	1

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	7	442	716	735	2139	0
SD	10	-20	274	215	638	0
SC+E	-8	0	0	-21	-21	0
E+SD	3	0	-14	-53	-114	0
SC+F	-7	26	716	743	2147	0
SD+F	3	26	274	217	640	0
SC+F+E	-5	26	0	-16	-16	0
E+SD+F	-6	26	-14	-51	-112	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	5800	2425	922	4878	1503	0.12	0	0	0
SD	2203	901	348	1855	553	0.15	12511	23	0.00
SC+E	581	289	144	437	145	0.46	0	0	0
E+SD	214	168	110	104	58	0.87	199	4	0.02
SC+F	5800	2425	922	4878	1503	0.12	0	0	0
SD+F	2203	901	348	1855	553	0.15	12620	23	0.00
SC+F+E	581	289	144	437	145	0.46	0	0	0
E+SD+F	214	168	110	104	58	0.87	208	4	0.02

(c)

Table B.10: The raw and EPOR-relative exploration data and reduction data from dining philosophers DP1 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	57	615	798	831	2584	1
SD	57	72	311	276	950	1
SC+E	49	91	51	63	391	1
E+SD	49	91	18	5	192	1
E	37	91	51	99	427	1
SC+F	45	128	798	843	2596	1
SD+F	43	128	325	284	990	1
SC+F+E	49	128	51	70	398	1
E+SD+F	51	128	28	14	243	1

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	20	524	747	732	2157	0
SD	20	-19	260	177	523	0
SC+E	12	0	0	-36	-36	0
E+SD	12	0	-33	-94	-235	0
SC+F	8	37	747	744	2169	0
SD+F	6	37	274	185	563	0
SC+F+E	12	37	0	-29	-29	0
E+SD+F	14	37	-23	-85	-184	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	6175	2583	977	5198	1606	0.13	0	0	0
SD	2256	925	350	1906	575	0.16	13171	24	0.00
SC+E	874	414	189	685	225	0.38	0	0	0
E+SD	240	186	112	128	74	0.86	271	5	0.02
SC+F	6175	2583	977	5198	1606	0.13	0	0	0
SD+F	2373	957	367	2006	590	0.15	13886	27	0.00
SC+F+E	874	414	189	685	225	0.38	0	0	0
E+SD+F	323	228	133	190	95	0.81	608	9	0.01

(c)

Table B.11: The raw and EPOR-relative exploration data and reduction data from dining philosophers DP2 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	64	533	798	831	2589	1
SD	52	72	311	276	955	1
SC+E	50	91	51	63	396	1
E+SD	38	91	18	5	197	1
E	33	91	51	99	432	1
SC+F	54	128	798	843	2601	1
SD+F	52	128	325	284	995	1
SC+F+E	56	128	51	70	403	1
E+SD+F	51	128	28	14	248	1

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	31	442	747	732	2157	0
SD	19	-19	260	177	523	0
SC+E	17	0	0	-36	-36	0
E+SD	5	0	-33	-94	-235	0
SC+F	21	37	747	744	2169	0
SD+F	19	37	274	185	563	0
SC+F+E	23	37	0	-29	-29	0
E+SD+F	18	37	-23	-85	-184	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	6179	2587	990	5189	1597	0.13	0	0	0
SD	2260	929	363	1897	566	0.17	13171	24	0.00
SC+E	878	418	202	676	216	0.39	0	0	0
E+SD	244	190	125	119	65	0.86	271	5	0.02
SC+F	6179	2587	990	5189	1597	0.13	0	0	0
SD+F	2377	961	380	1997	581	0.16	13886	27	0.00
SC+F+E	878	418	202	676	216	0.39	0	0	0
E+SD+F	327	232	146	181	86	0.81	608	9	0.01

(c)

Table B.12: The raw and EPOR-relative exploration data and reduction data from dining philosophers DP3 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	65	903	1618	2056	6197	1
SD	58	92	373	334	1268	1
SC+E	45	92	30	36	321	1
E+SD	47	92	16	4	228	1
E	53	92	30	57	342	1
SC+F	46	867	1621	2092	6238	1
SD+F	41	194	376	336	1275	1
SC+F+E	50	194	30	41	326	1
E+SD+F	52	194	16	6	230	1

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	12	811	1588	1999	5855	0
SD	5	0	343	277	926	0
SC+E	-8	0	0	-21	-21	0
E+SD	-6	0	-14	-53	-114	0
SC+F	-7	775	1591	2035	5896	0
SD+F	-12	102	346	279	933	0
SC+F+E	-3	102	0	-16	-16	0
E+SD+F	-1	102	-14	-51	-112	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	14926	6196	2524	12402	3672	0.09	0	0	0
SD	3246	1244	564	2682	680	0.11	14549	23	0.00
SC+E	768	390	199	569	191	0.38	0	0	0
E+SD	275	223	165	110	58	0.89	199	4	0.02
SC+F	14939	6204	2526	12413	3678	0.09	0	0	0
SD+F	3265	1249	566	2699	683	0.10	14664	24	0.00
SC+F+E	768	390	199	569	191	0.38	0	0	0
E+SD+F	275	223	165	110	58	0.89	208	4	0.02

(c)

Table B.13: The raw and EPOR-relative exploration data and reduction data from dining philosophers DP4 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	69	596	1910	2304	7123	1
SD	51	92	442	417	1564	1
SC+E	47	93	8	2	142	1
E+SD	50	93	9	1	144	1
E	49	92	153	346	1434	1
SC+F	65	820	1930	2359	7200	1
SD+F	59	152	448	423	1578	1
SC+F+E	56	153	8	3	143	1
E+SD+F	54	153	9	3	146	1

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	20	504	1757	1958	5689	0
SD	2	0	289	71	130	0
SC+E	-2	1	-145	-344	-1292	0
E+SD	1	1	-144	-345	-1290	0
SC+F	16	728	1777	2013	5766	0
SD+F	10	60	295	77	144	0
SC+F+E	7	61	-145	-343	-1291	0
E+SD+F	5	61	-144	-343	-1288	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	17599	7121	2961	14638	4160	0.08	0	0	0
SD	3967	1539	706	3261	833	0.12	16910	23	0.00
SC+E	197	151	103	94	48	0.72	0	0	0
E+SD	153	141	104	49	37	0.93	18	1	0.06
SC+F	17696	7146	2980	14716	4166	0.08	0	0	0
SD+F	4001	1547	711	3290	836	0.12	17153	24	0.00
SC+F+E	197	151	103	94	48	0.72	0	0	0
E+SD+F	153	141	104	49	37	0.93	27	2	0.07

(c)

Table B.14: The raw and EPOR-relative exploration data and reduction data from dining philosophers DP5 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	64	515	1910	2304	7115	1
SD	40	92	442	417	1556	1
SC+E	41	93	8	2	138	1
E+SD	44	93	9	1	140	1
E	34	92	153	346	1426	1
SC+F	64	692	1930	2359	7192	1
SD+F	65	151	448	423	1570	1
SC+F+E	55	151	8	3	139	1
E+SD+F	53	151	9	3	142	1

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	30	423	1757	1958	5689	0
SD	6	0	289	71	130	0
SC+E	7	1	-145	-344	-1288	0
E+SD	10	1	-144	-345	-1286	0
SC+F	30	600	1777	2013	5766	0
SD+F	31	59	295	77	144	0
SC+F+E	21	59	-145	-343	-1287	0
E+SD+F	19	59	-144	-343	-1284	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	17591	7113	2953	14638	4160	0.08	0	0	0
SD	3959	1531	698	3261	833	0.12	16910	23	0.00
SC+E	193	147	99	94	48	0.71	0	0	0
E+SD	149	137	100	49	37	0.93	18	1	0.06
SC+F	17688	7138	2972	14716	4166	0.08	0	0	0
SD+F	3993	1539	703	3290	836	0.12	17153	24	0.00
SC+F+E	193	147	99	94	48	0.71	0	0	0
E+SD+F	149	137	100	49	37	0.93	27	2	0.07

(c)

Table B.15: The raw and EPOR-relative exploration data and reduction data from dining philosophers DP6 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	36019	5937	11908	743	7251107	0
SD	36247	14842	9150	0	2762960	0
SC+E	5469	6135	4419	647	529098	0
E+SD	1564	5932	2478	43	103302	0
E	36012	5890	200622	197416	8287084	0
SC+F	36081	10539	10383	471	6303446	0
SD+F	36288	14407	8463	0	2105125	0
SC+F+E	5497	6207	4419	647	529098	0
E+SD+F	1629	5966	2478	43	103302	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	7	47	-188714	-196673	-1035977	0
SD	235	8952	-191472	-197416	-5524124	0
SC+E	-30543	245	-196203	-196769	-7757986	0
E+SD	-34448	42	-198144	-197373	-8183782	0
SC+F	69	4649	-190239	-196945	-1983638	0
SD+F	276	8517	-192159	-197416	-6181959	0
SC+F+E	-30515	317	-196203	-196769	-7757986	0
E+SD+F	-34383	76	-198144	-197373	-8183782	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	7268871	7254294	4619080	2649791	2635214	1.00	0	0	0
SD	5520859	2762954	2755684	2765175	7270	0.00	45275	0	0
SC+E	758550	632726	289442	469108	343284	0.76	0	0	0
E+SD	109860	103254	58298	51562	44956	0.94	3501994	41	0.00
SC+F	6319678	6306597	4111038	2208640	2195559	1.00	0	0	0
SD+F	4205189	2105119	2097879	2107310	7240	0.00	44975	0	0
SC+F+E	758550	632726	289442	469108	343284	0.76	0	0	0
E+SD+F	109860	103254	58298	51562	44956	0.94	3501994	41	0.00

(c)

Table B.16: The raw and EPOR-relative exploration data and reduction data from molecular dynamics MD3 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	36011	5526	1831	156	1455033	0
SD	13314	6381	22909	5276	435667	0
SC+E	36014	5960	2298	1119	711010	0
E+SD	1830	5607	171	5	42604	0
E	24263	5783	4616	3569	1247000	0
SC+F	36034	6942	1812	155	1437594	0
SD+F	13287	6640	22909	5276	435667	0
SC+F+E	36027	6666	2186	1033	702675	0
E+SD+F	1827	5652	171	5	42604	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	11748	-257	-2785	-3413	208033	0
SD	-10949	598	18293	1707	-811333	0
SC+E	11751	177	-2318	-2450	-535990	0
E+SD	-22433	-176	-4445	-3564	-1204396	0
SC+F	11771	1159	-2804	-3414	190594	0
SD+F	-10976	857	18293	1707	-811333	0
SC+F+E	11764	883	-2430	-2536	-544325	0
E+SD+F	-22436	-131	-4445	-3564	-1204396	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	1457291	1455148	1057730	399561	397418	1.00	0	0	0
SD	841343	435325	403207	438136	32118	0.06	82471867	336	0.00
SC+E	1098039	802388	519537	578502	282851	0.58	0	0	0
E+SD	43966	42595	32894	11072	9701	0.97	56594	3	0.00
SC+F	1439849	1437709	1045264	394585	392445	1.00	0	0	0
SD+F	841343	435325	403207	438136	32118	0.06	82471867	336	0.00
SC+F+E	1081541	792354	513095	568446	279259	0.59	0	0	0
E+SD+F	43966	42595	32894	11072	9701	0.97	56594	3	0.00

(c)

Table B.17: The raw and EPOR-relative exploration data and reduction data from ray tracer RT3 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	1446	1358	301293	362901	1081729	0
SD	128	1151	12710	13425	46687	0
SC+E	714	829	69682	78117	323185	0
E+SD	55	232	82	17	553	0
E	119	799	7379	15859	76291	0
SC+F	1746	1641	301293	362901	1081729	0
SD+F	126	1269	12710	13425	46687	0
SC+F+E	748	1481	69682	78117	323185	0
E+SD+F	53	248	82	17	553	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	1327	559	293914	347042	1005438	0
SD	9	352	5331	-2434	-29604	0
SC+E	595	30	62303	62258	246894	0
E+SD	-64	-567	-7297	-15842	-75738	0
SC+F	1627	842	293914	347042	1005438	0
SD+F	7	470	5331	-2434	-29604	0
SC+F+E	629	682	62303	62258	246894	0
E+SD+F	-66	-551	-7297	-15842	-75738	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	2732043	1081728	435388	2296655	646340	0.05	0	0	0
SD	129663	44038	20943	108720	23095	0.03	2218888	2648	0.00
SC+E	865735	362456	144030	721705	218426	0.07	0	0	0
E+SD	722	538	342	380	196	0.83	2837	14	0.00
SC+F	2732043	1081728	435388	2296655	646340	0.05	0	0	0
SD+F	129663	44038	20943	108720	23095	0.03	2218888	2648	0.00
SC+F+E	865735	362456	144030	721705	218426	0.07	0	0	0
E+SD+F	722	538	342	380	196	0.83	2837	14	0.00

(c)

Table B.18: The raw and EPOR-relative exploration data and reduction data from pipeline PL1 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	75	430	291	87	2189	0
SD	53	73	287	84	2177	0
SC+E	47	93	48	31	1363	0
E+SD	49	93	48	31	1375	0
E	36	92	57	44	1637	0
SC+F	45	95	291	87	2189	0
SD+F	42	95	287	84	2177	0
SC+F+E	40	95	48	31	1363	0
E+SD+F	43	95	48	31	1375	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	39	338	234	43	552	0
SD	17	-19	230	40	540	0
SC+E	11	1	-9	-13	-274	0
E+SD	13	1	-9	-13	-262	0
SC+F	9	3	234	43	552	0
SD+F	6	3	230	40	540	0
SC+F+E	4	3	-9	-13	-274	0
E+SD+F	7	3	-9	-13	-262	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	2839	2188	1066	1773	1122	0.70	0	0	0
SD	2818	2153	1060	1758	1093	0.69	5531	23	0.00
SC+E	1668	1380	665	1003	715	0.79	0	0	0
E+SD	1383	1350	671	712	679	0.98	3207	24	0.01
SC+F	2839	2188	1066	1773	1122	0.70	0	0	0
SD+F	2818	2153	1060	1758	1093	0.69	5531	23	0.00
SC+F+E	1668	1380	665	1003	715	0.79	0	0	0
E+SD+F	1383	1350	671	712	679	0.98	3207	24	0.01

(c)

Table B.19: The raw and EPOR-relative exploration data and reduction data from producer-consumer PC3 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	61	1046	1007	615	9986	0
SD	60	1143	582	284	7488	0
SC+E	52	1352	88	64	3460	0
E+SD	49	1351	88	64	3469	0
E	46	95	103	85	4057	0
SC+F	62	1356	1007	615	9986	0
SD+F	41	1356	582	284	7488	0
SC+F+E	43	1460	88	64	3460	0
E+SD+F	45	1356	88	64	3469	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	15	951	904	530	5929	0
SD	14	1048	479	199	3431	0
SC+E	6	1257	-15	-21	-597	0
E+SD	3	1256	-15	-21	-588	0
SC+F	16	1261	904	530	5929	0
SD+F	-5	1261	479	199	3431	0
SC+F+E	-3	1365	-15	-21	-597	0
E+SD+F	-1	1261	-15	-21	-588	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	14235	9985	6014	8221	3971	0.57	0	0	0
SD	10274	7285	4529	5745	2756	0.59	25889	202	0.01
SC+E	4325	3514	1991	2334	1523	0.76	0	0	0
E+SD	3542	3419	1997	1545	1422	0.96	8888	49	0.01
SC+F	14235	9985	6014	8221	3971	0.57	0	0	0
SD+F	10274	7285	4529	5745	2756	0.59	25889	202	0.01
SC+F+E	4325	3514	1991	2334	1523	0.76	0	0	0
E+SD+F	3542	3419	1997	1545	1422	0.96	8888	49	0.01

(c)

Table B.20: The raw and EPOR-relative exploration data and reduction data from producer-consumer PC4 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	53	654	1403	1453	5480	0
SD	48	1022	861	733	3564	0
SC+E	41	106	102	127	1404	0
E+SD	49	106	72	66	989	0
E	36	106	124	296	1746	0
SC+F	65	1283	1415	1493	5578	0
SD+F	53	1152	875	770	3689	0
SC+F+E	50	107	104	139	1432	0
E+SD+F	49	107	72	79	1002	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	17	548	1279	1157	3734	0
SD	12	916	737	437	1818	0
SC+E	5	0	-22	-169	-342	0
E+SD	13	0	-52	-230	-757	0
SC+F	29	1177	1291	1197	3832	0
SD+F	17	1046	751	474	1943	0
SC+F+E	14	1	-20	-157	-314	0
E+SD+F	13	1	-52	-217	-744	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	10923	5479	2380	8543	3099	0.35	0	0	0
SD	6544	3389	1650	4894	1739	0.42	121289	215	0.00
SC+E	2080	1486	817	1263	669	0.77	0	0	0
E+SD	1108	951	588	520	363	0.92	17209	56	0.00
SC+F	11030	5543	2418	8612	3125	0.35	0	0	0
SD+F	6666	3481	1706	4960	1775	0.43	129951	216	0.00
SC+F+E	2105	1503	828	1277	675	0.77	0	0	0
E+SD+F	1108	951	588	520	363	0.92	18790	60	0.00

(c)

Table B.21: The raw and EPOR-relative exploration data and reduction data from readers-writers RW1 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	123	1422	13338	10795	47032	0
SD	131	981	9100	6438	30692	0
SC+E	94	563	5186	3480	22955	0
E+SD	95	717	2880	1606	12789	0
E	60	505	1214	2559	12312	0
SC+F	113	1182	13342	11328	47612	0
SD+F	151	826	10335	7562	35327	0
SC+F+E	82	475	4947	3758	22517	0
E+SD+F	95	628	4215	2876	18766	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	63	917	12124	8236	34720	0
SD	71	476	7886	3879	18380	0
SC+E	34	58	3972	921	10643	0
E+SD	35	212	1666	-953	477	0
SC+F	53	677	12128	8769	35300	0
SD+F	91	321	9121	5003	23015	0
SC+F+E	22	-30	3733	1199	10205	0
E+SD+F	35	123	3001	317	6454	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	111536	47021	21506	90030	25515	0.15	0	0	0
SD	73866	28259	14133	59733	14126	0.11	8059290	2471	0.00
SC+E	52055	23489	11912	40143	11577	0.20	0	0	0
E+SD	17889	11854	6883	11006	4971	0.66	2254047	973	0.00
SC+F	111317	46815	21517	89800	25298	0.15	0	0	0
SD+F	84418	32110	16225	68193	15885	0.10	11334045	2800	0.00
SC+F+E	50191	22396	11496	38695	10900	0.20	0	0	0
E+SD+F	25939	17129	9787	16152	7342	0.65	4278026	1312	0.00

(c)

Table B.22: The raw and EPOR-relative exploration data and reduction data from readers-writers RW2 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	69	144	1403	1453	5480	0
SD	50	92	861	733	3564	0
SC+E	45	93	102	127	1404	0
E+SD	43	93	72	66	989	0
E	40	93	124	296	1746	0
SC+F	46	94	1415	1493	5578	0
SD+F	46	94	875	770	3689	0
SC+F+E	53	94	104	139	1432	0
E+SD+F	40	94	72	79	1002	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	29	51	1279	1157	3734	0
SD	10	-1	737	437	1818	0
SC+E	5	0	-22	-169	-342	0
E+SD	3	0	-52	-230	-757	0
SC+F	6	1	1291	1197	3832	0
SD+F	6	1	751	474	1943	0
SC+F+E	13	1	-20	-157	-314	0
E+SD+F	0	1	-52	-217	-744	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	10923	5479	2380	8543	3099	0.35	0	0	0
SD	6544	3389	1650	4894	1739	0.42	121289	215	0.00
SC+E	2080	1486	817	1263	669	0.77	0	0	0
E+SD	1108	951	588	520	363	0.92	17209	56	0.00
SC+F	11030	5543	2418	8612	3125	0.35	0	0	0
SD+F	6666	3481	1706	4960	1775	0.43	129951	216	0.00
SC+F+E	2105	1503	828	1277	675	0.77	0	0	0
E+SD+F	1108	951	588	520	363	0.92	18790	60	0.00

(c)

Table B.23: The raw and EPOR-relative exploration data and reduction data from readers-writers RW3 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	70	987	1661	1647	7063	0
SD	57	1039	979	835	4426	0
SC+E	51	94	121	154	1794	0
E+SD	51	94	88	79	1311	0
E	49	93	134	299	2091	0
SC+F	63	1353	1671	1702	7190	0
SD+F	53	1143	984	854	4490	0
SC+F+E	51	95	121	157	1797	0
E+SD+F	49	95	89	93	1335	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	21	894	1527	1348	4972	0
SD	8	946	845	536	2335	0
SC+E	2	1	-13	-145	-297	0
E+SD	2	1	-46	-220	-780	0
SC+F	14	1260	1537	1403	5099	0
SD+F	4	1050	850	555	2399	0
SC+F+E	2	2	-13	-142	-294	0
E+SD+F	0	2	-45	-206	-756	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	12948	7062	3135	9813	3927	0.46	0	0	0
SD	7459	4252	2032	5427	2220	0.51	148940	223	0.00
SC+E	2409	1862	1006	1403	856	0.82	0	0	0
E+SD	1432	1278	751	681	527	0.93	24306	46	0.00
SC+F	13053	7140	3178	9875	3962	0.46	0	0	0
SD+F	7518	4299	2059	5459	2240	0.52	156159	226	0.00
SC+F+E	2409	1862	1006	1403	856	0.82	0	0	0
E+SD+F	1445	1288	757	688	531	0.93	25621	46	0.00

(c)

Table B.24: The raw and EPOR-relative exploration data and reduction data from readers-writers RW4 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	78	1019	5113	4744	22702	0
SD	73	625	2616	1937	12243	0
SC+E	60	60	173	114	2975	0
E+SD	57	61	177	75	2665	0
E	62	59	58	108	2116	0
SC+F	78	693	5161	4831	23098	0
SD+F	69	630	2625	1995	12346	0
SC+F+E	63	64	180	129	3058	0
E+SD+F	59	65	183	84	2787	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	16	960	5055	4636	20586	0
SD	11	566	2558	1829	10127	0
SC+E	-2	1	115	6	859	0
E+SD	-5	2	119	-33	549	0
SC+F	16	634	5103	4723	20982	0
SD+F	7	571	2567	1887	10230	0
SC+F+E	1	5	122	21	942	0
E+SD+F	-3	6	125	-24	671	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	45673	22701	11068	34605	11633	0.35	0	0	0
SD	23187	11613	6314	16873	5299	0.35	1231920	652	0.00
SC+E	4178	3029	1688	2490	1341	0.76	0	0	0
E+SD	2915	2600	1552	1363	1048	0.93	77497	82	0.00
SC+F	46124	23032	11245	34879	11787	0.36	0	0	0
SD+F	23258	11658	6338	16920	5320	0.36	1219570	651	0.00
SC+F+E	4265	3100	1725	2540	1375	0.77	0	0	0
E+SD+F	3037	2713	1621	1416	1092	0.93	82508	82	0.00

(c)

Table B.25: The raw and EPOR-relative exploration data and reduction data from readers-writers RW5 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	337	2227	5760	1764	98168	0
SD	162	2074	1619	747	31180	0
SC+E	325	1322	1788	641	63785	0
E+SD	338	1286	1587	495	60595	0
E	619	872	6577	6562	362027	0
SC+F	317	1409	5760	1764	98168	0
SD+F	156	1347	1619	747	31180	0
SC+F+E	338	1631	1788	641	63785	0
E+SD+F	332	1956	1587	495	60595	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	-282	1355	-817	-4798	-263859	0
SD	-457	1202	-4958	-5815	-330847	0
SC+E	-294	450	-4789	-5921	-298242	0
E+SD	-281	414	-4990	-6067	-301432	0
SC+F	-302	537	-817	-4798	-263859	0
SD+F	-463	475	-4958	-5815	-330847	0
SC+F+E	-281	759	-4789	-5921	-298242	0
E+SD+F	-287	1084	-4990	-6067	-301432	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	129497	98163	71277	58220	26886	0.67	0	0	0
SD	37132	30547	20861	16271	9686	0.78	189609	630	0.00
SC+E	80099	64186	45758	34341	18428	0.75	0	0	0
E+SD	62388	60185	43827	18561	16358	0.96	173626	405	0.00
SC+F	129497	98163	71277	58220	26886	0.67	0	0	0
SD+F	37132	30547	20861	16271	9686	0.78	189609	630	0.00
SC+F+E	80099	64186	45758	34341	18428	0.75	0	0	0
E+SD+F	62388	60185	43827	18561	16358	0.96	173626	405	0.00

(c)

Table B.26: The raw and EPOR-relative exploration data and reduction data from replicated workers RP13 input program.

B.3 Terminated Configurations

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	14357	5119	1043247	753132	4823439	0
SD	5956	13922	251698	108054	1047995	0
SC+E	16302	6405	568928	328218	3259745	0
E+SD	2619	7468	95657	43268	413817	0
E	13515	6382	612954	1010878	5709125	0
SC+F	19593	12220	1043247	753132	4823439	0
SD+F	6816	14390	251698	108054	1047995	0
SC+F+E	18419	10279	568928	328218	3259745	0
E+SD+F	2654	7954	95657	43268	413817	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	842	-1263	430293	-257746	-885686	0
SD	-7559	7540	-361256	-902824	-4661130	0
SC+E	2787	23	-44026	-682660	-2449380	0
E+SD	-10896	1086	-517297	-967610	-5295308	0
SC+F	6078	5838	430293	-257746	-885686	0
SD+F	-6699	8008	-361256	-902824	-4661130	0
SC+F+E	4904	3897	-44026	-682660	-2449380	0
E+SD+F	-10861	1572	-517297	-967610	-5295308	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	9466092	4823436	2741042	6725050	2082394	0.26	0	0	0
SD	1888152	951921	578447	1309705	373474	0.29	121278050	96143	0.00
SC+E	6736469	3422551	1967867	4768602	1454684	0.26	0	0	0
E+SD	503251	379445	220569	282682	158876	0.73	19058347	34379	0.00
SC+F	9466092	4823436	2741042	6725050	2082394	0.26	0	0	0
SD+F	1888152	951921	578447	1309705	373474	0.29	121278050	96143	0.00
SC+F+E	6736469	3422551	1967867	4768602	1454684	0.26	0	0	0
E+SD+F	503251	379445	220569	282682	158876	0.73	19058347	34379	0.00

(c)

Table B.27: The raw and EPOR-relative exploration data and reduction data from replicated workers RP15 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	629	2736	48232	34787	252853	0
SD	377	1480	20550	10783	105190	0
SC+E	621	1486	23408	13496	154699	0
E+SD	478	2151	14497	7225	96363	0
E	506	2306	26854	40226	321962	0
SC+F	628	2971	48232	34787	252853	0
SD+F	364	3126	20550	10783	105190	0
SC+F+E	640	3612	23408	13496	154699	0
E+SD+F	474	4041	14497	7225	96363	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	123	430	21378	-5439	-69109	0
SD	-129	-826	-6304	-29443	-216772	0
SC+E	115	-820	-3446	-26730	-167263	0
E+SD	-28	-155	-12357	-33001	-225599	0
SC+F	122	665	21378	-5439	-69109	0
SD+F	-142	820	-6304	-29443	-216772	0
SC+F+E	134	1306	-3446	-26730	-167263	0
E+SD+F	-32	1735	-12357	-33001	-225599	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	463428	252850	146396	317032	106454	0.34	0	0	0
SD	185304	97196	60228	125076	36968	0.34	5617528	8004	0.00
SC+E	288128	160836	92825	195303	68011	0.37	0	0	0
E+SD	112458	91342	57566	54892	33776	0.81	3934334	5034	0.00
SC+F	463428	252850	146396	317032	106454	0.34	0	0	0
SD+F	185304	97196	60228	125076	36968	0.34	5617528	8004	0.00
SC+F+E	288128	160836	92825	195303	68011	0.37	0	0	0
E+SD+F	112458	91342	57566	54892	33776	0.81	3934334	5034	0.00

(c)

Table B.28: The raw and EPOR-relative exploration data and reduction data from replicated workers RP18 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	63	392	1870	1650	5656	6
SD	48	74	743	663	2090	2
SC+E	41	96	442	147	1502	6
E+SD	34	95	54	8	256	2
E	64	3314	11205	22139	47317	6
SC+F	42	632	1870	1650	5656	6
SD+F	34	715	803	683	2300	6
SC+F+E	39	129	442	147	1502	6
E+SD+F	33	127	114	24	462	6

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	-1	-2922	-9335	-20489	-41661	0
SD	-16	-3240	-10462	-21476	-45227	-4
SC+E	-23	-3218	-10763	-21992	-45815	0
E+SD	-30	-3219	-11151	-22131	-47061	-4
SC+F	-22	-2682	-9335	-20489	-41661	0
SD+F	-30	-2599	-10402	-21456	-45017	0
SC+F+E	-25	-3185	-10763	-21992	-45815	0
E+SD+F	-31	-3187	-11091	-22115	-46855	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	12587	5655	2059	10528	3596	0.21	0	0	0
SD	4346	1981	609	3737	1372	0.20	11558	108	0.01
SC+E	3248	1560	736	2512	824	0.33	0	0	0
E+SD	339	246	135	204	111	0.77	92	9	0.10
SC+F	12587	5655	2059	10528	3596	0.21	0	0	0
SD+F	4618	2167	681	3937	1486	0.24	16482	126	0.01
SC+F+E	3248	1560	736	2512	824	0.33	0	0	0
E+SD+F	579	432	207	372	225	0.74	248	27	0.11

(c)

Table B.29: The raw and EPOR-relative exploration data and reduction data from sleeping barbers SB1 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	151	1150	31343	24638	90649	0
SD	201	604	22866	16879	63264	0
SC+E	135	851	19349	12790	56343	0
E+SD	130	1015	11167	6408	31290	0
E	2415	628	213865	493092	984758	0
SC+F	142	830	31346	24770	90794	0
SD+F	210	760	23298	17254	64574	0
SC+F+E	139	849	19353	12922	56489	0
E+SD+F	135	824	11599	6783	32600	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	-2264	522	-182522	-468454	-894109	0
SD	-2214	-24	-190999	-476213	-921494	0
SC+E	-2280	223	-194516	-480302	-928415	0
E+SD	-2285	387	-202698	-486684	-953468	0
SC+F	-2273	202	-182519	-468322	-893964	0
SD+F	-2205	132	-190567	-475838	-920184	0
SC+F+E	-2276	221	-194512	-480170	-928269	0
E+SD+F	-2280	196	-202266	-486309	-952158	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	203245	90598	33559	169686	57039	0.14	0	0	0
SD	140961	55928	22176	118785	33752	0.06	18708267	7341	0.00
SC+E	122152	56852	22530	99622	34322	0.17	0	0	0
E+SD	47683	27564	12399	35284	15165	0.49	6769341	3714	0.00
SC+F	203281	90612	33569	169712	57043	0.14	0	0	0
SD+F	144472	57053	22675	121797	34378	0.06	19156149	7487	0.00
SC+F+E	122191	56867	22540	99651	34327	0.17	0	0	0
E+SD+F	49993	28689	12898	37095	15791	0.48	7212881	3860	0.00

(c)

Table B.30: The raw and EPOR-relative exploration data and reduction data from sleeping barbers SB4 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	36028	3075	484320	176323	13130000	6922
SD	36316	12172	280717	78041	8420000	3
SC+E	36075	6221	245323	97748	10761570	448
E+SD	36308	14043	154538	60075	6858336	0
E	36115	6579	318652	137135	14480000	0
SC+F	36189	8462	107674	37712	2980661	984
SD+F	36347	14268	186871	51978	5605795	0
SC+F+E	36280	13351	170504	67193	7540000	0
E+SD+F	36310	14300	110420	42890	4895085	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	-87	-3504	165668	39188	-1350000	0
SD	201	5593	-37935	-59094	-6060000	0
SC+E	-40	-358	-73329	-39387	-3718430	0
E+SD	193	7464	-164114	-77060	-7621664	0
SC+F	74	1883	-210978	-99423	-11499339	0
SD+F	232	7689	-131781	-85157	-8874205	0
SC+F+E	165	6772	-148148	-69942	-6940000	0
E+SD+F	195	7721	-208232	-94245	-9584915	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	14307229	13130041	7218084	7089145	5911957	0.92	0	0	0
SD	9046789	8355377	4833261	4213528	3522116	0.92	45743350	64668	0.00
SC+E	11554895	10772540	6151656	5403239	4620884	0.93	0	0	0
E+SD	6865277	6803916	3935919	2929358	2867997	0.99	33516766	54461	0.00
SC+F	3237344	2980704	1656118	1581226	1324586	0.93	0	0	0
SD+F	6022115	5562942	3218717	2803398	2344225	0.92	29535614	42886	0.00
SC+F+E	8091637	7546938	4321143	3770494	3225795	0.93	0	0	0
E+SD+F	4899828	4856372	2808718	2091110	2047654	0.99	23200199	38743	0.00

(c)

Table B.31: The raw and EPOR-relative exploration data and reduction data from disk scheduler DS1 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	36030	2033	586207	213654	15861823	8672
SD	36242	12356	312253	86737	9359041	5
SC+E	36064	6665	302288	121775	13180000	865
E+SD	36199	14247	175784	68830	7805424	1
E	36081	5549	439430	188600	19900000	0
SC+F	36131	10150	126848	44942	3491532	1366
SD+F	36256	14252	192269	53477	5770000	0
SC+F+E	36184	13792	193605	76500	8560000	0
E+SD+F	36228	14293	119919	46644	5312019	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	-51	-3516	146777	25054	-4038177	0
SD	161	6807	-127177	-101863	-10540959	0
SC+E	-17	1116	-137142	-66825	-6720000	0
E+SD	118	8698	-263646	-119770	-12094576	0
SC+F	50	4601	-312582	-143658	-16408468	0
SD+F	175	8703	-247161	-135123	-14130000	0
SC+F+E	103	8243	-245825	-112100	-11340000	0
E+SD+F	147	8744	-319511	-141956	-14587981	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	17294081	15861867	8703982	8590099	7157885	0.93	0	0	0
SD	10056316	9287145	5372443	4683873	3914702	0.92	51005785	71949	0.00
SC+E	14165557	13193854	7518151	6647406	5675703	0.93	0	0	0
E+SD	7813389	7743107	4477593	3335796	3265514	0.99	38710023	62361	0.00
SC+F	3794063	3491574	1935889	1858174	1555685	0.93	0	0	0
SD+F	6198527	5725851	3313147	2885380	2412704	0.92	30483575	44180	0.00
SC+F+E	9186673	8568005	4904530	4282143	3663475	0.93	0	0	0
E+SD+F	5317229	5269891	3047699	2269530	2222192	0.99	25331009	42158	0.00

(c)

Table B.32: The raw and EPOR-relative exploration data and reduction data from disk scheduler DS2 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	36050	4447	927413	219615	29990000	21
SD	36256	10549	199748	44681	6570000	0
SC+E	36085	7910	256656	83012	12170000	0
E+SD	36226	13811	122645	38192	5874295	0
E	36117	7261	282599	96813	13800000	0
SC+F	36275	12701	245509	57062	8020000	0
SD+F	36367	14272	164164	36282	5430000	0
SC+F+E	36264	13918	151804	48727	7180000	0
E+SD+F	36338	14315	96442	30033	4635284	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	-67	-2814	644814	122802	16190000	0
SD	139	3288	-82851	-52132	-7230000	0
SC+E	-32	649	-25943	-13801	-1630000	0
E+SD	109	6550	-159954	-58621	-7925705	0
SC+F	158	5440	-37090	-39751	-5780000	0
SD+F	250	7011	-118435	-60531	-8370000	0
SC+F+E	147	6657	-130795	-48086	-6620000	0
E+SD+F	221	7054	-186157	-66780	-9164716	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	33244817	29990097	17137631	16107186	12852466	0.94	0	0	0
SD	7265168	6525446	3766129	3499039	2759317	0.89	38910802	44621	0.00
SC+E	13444299	12180543	6954328	6489971	5226215	0.90	0	0	0
E+SD	5880275	5832322	3366566	2513709	2465756	0.99	32683258	42042	0.00
SC+F	8868206	8019155	4597822	4270384	3421333	0.89	0	0	0
SD+F	6001490	5393296	3115502	2885988	2277794	0.89	31589272	36775	0.00
SC+F+E	7929954	7186737	4101738	3828216	3084999	0.90	0	0	0
E+SD+F	4640117	4602231	2657909	1982208	1944322	0.99	25567266	33124	0.00

(c)

Table B.33: The raw and EPOR-relative exploration data and reduction data from disk scheduler DS4 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	36024	2618	802545	528986	14401190	10604
SD	36114	8645	247030	124269	4104736	2490
SC+E	36035	4969	136911	78195	4603123	4346
E+SD	36082	7226	77217	30770	2522397	2409
E	36037	3599	157684	129614	5143947	4155
SC+F	36080	7247	111268	61175	2597494	2679
SD+F	36109	7967	74080	28997	1721830	1977
SC+F+E	36081	7358	75774	40260	2576842	2228
E+SD+F	36111	7282	43290	17457	1417963	2027

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	-13	-981	644861	399372	9257243	6449
SD	77	5046	89346	-5345	-1039211	-1665
SC+E	-2	1370	-20773	-51419	-540824	191
E+SD	45	3627	-80467	-98844	-2621550	-1746
SC+F	43	3648	-46416	-68439	-2546453	-1476
SD+F	72	4368	-83604	-100617	-3422117	-2178
SC+F+E	44	3759	-81910	-89354	-2567105	-1927
E+SD+F	74	3683	-114394	-112157	-3725984	-2128

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	17608612	14401228	7722382	9886230	6678846	0.85	0	0	0
SD	5113093	4023061	2281952	2831141	1741109	0.80	79471699	96253	0.00
SC+E	5118320	4613372	2429884	2688436	2183488	0.93	0	0	0
E+SD	2543126	2501487	1377355	1165771	1124132	0.99	16245521	26675	0.00
SC+F	2970141	2597531	1364863	1605278	1232668	0.91	0	0	0
SD+F	1955955	1701687	926715	1029240	774972	0.90	16274744	24943	0.00
SC+F+E	2851382	2581851	1373181	1478201	1208670	0.93	0	0	0
E+SD+F	1428121	1405120	777314	650807	627806	0.99	8477691	15091	0.00

(c)

Table B.34: The raw and EPOR-relative exploration data and reduction data from disk scheduler DS7 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	36038	7325	2854899	1773213	9006579	0
SD	36054	9671	320514	157402	967201	0
SC+E	36044	7541	1757174	1028687	5700000	0
E+SD	36086	14613	636581	309869	2055708	0
E	36043	8189	1010626	2168202	7522067	0
SC+F	36148	14940	1849804	1143564	5880000	0
SD+F	36073	11424	316597	155417	955702	0
SC+F+E	36134	14495	1526923	893958	4980000	0
E+SD+F	36115	14735	571418	281616	1812125	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	-5	-864	1844273	-394989	1484512	0
SD	11	1482	-690112	-2010800	-6554866	0
SC+E	1	-648	746548	-1139515	-1822067	0
E+SD	43	6424	-374045	-1858333	-5466359	0
SC+F	105	6751	839178	-1024638	-1642067	0
SD+F	30	3235	-694029	-2012785	-6566365	0
SC+F+E	91	6306	516297	-1274244	-2542067	0
E+SD+F	72	6546	-439208	-1886586	-5709942	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	19592136	9011127	4251374	15340762	4759753	0.17	0	0	0
SD	2000755	830210	460272	1540483	369938	0.11	719140148	138584	0.00
SC+E	11982175	5771693	2736436	9245739	3035257	0.20	0	0	0
E+SD	2856111	1790584	1029497	1826614	761087	0.62	509945970	266895	0.00
SC+F	12578050	5883808	2783108	9794942	3100700	0.19	0	0	0
SD+F	1977207	820644	454545	1522662	366099	0.11	710342925	136605	0.00
SC+F+E	10448212	5038921	2396965	8051247	2641956	0.21	0	0	0
E+SD+F	2528376	1573384	897617	1630759	675767	0.61	433298735	240537	0.00

(c)

Table B.35: The raw and EPOR-relative exploration data and reduction data from replicated workers RP12 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	36029	6378	1496403	701784	5400914	0
SD	36086	9378	523174	207986	2018866	0
SC+E	36040	6990	916175	421679	3419295	0
E+SD	36081	9033	387088	153376	1599607	0
E	36046	7357	480884	961519	6417914	0
SC+F	36130	12882	1222803	566817	4373007	0
SD+F	36128	12178	503619	200552	1962390	0
SC+F+E	36097	11224	850187	389611	3205777	0
E+SD+F	36105	11003	377415	149814	1570605	0

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	-17	-979	1015519	-259735	-1017000	0
SD	40	2021	42290	-753533	-4399048	0
SC+E	-6	-367	435291	-539840	-2998619	0
E+SD	35	1676	-93796	-808143	-4818307	0
SC+F	84	5525	741919	-394702	-2044907	0
SD+F	82	4821	22735	-760967	-4455524	0
SC+F+E	51	3867	369303	-571908	-3212137	0
E+SD+F	59	3646	-103469	-811705	-4847309	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	9799819	5402924	2816427	6983392	2586497	0.33	0	0	0
SD	3433479	1821698	1099250	2334229	722448	0.28	141058275	197266	0.00
SC+E	5942178	3420262	1799928	4142250	1620334	0.36	0	0	0
E+SD	1909990	1454631	878457	1031533	576174	0.73	96085136	145076	0.00
SC+F	7893628	4375180	2282539	5611089	2092641	0.32	0	0	0
SD+F	3311365	1772376	1069730	2241635	702646	0.29	134878469	190117	0.00
SC+F+E	5509392	3206814	1692635	3816757	1514179	0.37	0	0	0
E+SD+F	1873030	1428991	863296	1009734	565695	0.73	93884999	141714	0.00

(c)

Table B.36: The raw and EPOR-relative exploration data and reduction data from replicated workers RP14 input program.

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	36068	7284	3567474	3870150	12208052	1
SD	36189	11667	1025894	1223108	3970000	1
SC+E	36086	8253	1458629	2250433	6286847	1
E+SD	36139	10583	599683	922882	2661671	1
E	36089	8723	722112	3246748	10869856	1
SC+F	36237	12376	2035248	2163348	6910000	1
SD+F	36278	14894	1001193	981585	3534082	1
SC+F+E	36195	12529	1167223	1825146	5009925	1
E+SD+F	36188	12682	568748	881842	2521013	1

(a)

Conf	Time (s)	Memory (MB)	States	Matched	Transitions	Errors
SC	-21	-1439	2845362	623402	1338196	0
SD	100	2944	303782	-2023640	-6899856	0
SC+E	-3	-470	736517	-996315	-4583009	0
E+SD	50	1860	-122429	-2323866	-8208185	0
SC+F	148	3653	1313136	-1083400	-3959856	0
SD+F	189	6171	279081	-2265163	-7335774	0
SC+F+E	106	3806	445111	-1421602	-5859931	0
E+SD+F	99	3959	-153364	-2364906	-8348843	0

(b)

Conf	E	A	I	E-I	A-I	A/E	S	D	D/S
SC	39104259	12200022	5092116	34012143	7107906	0.04	0	0	0
SD	12202031	3749493	1874935	10327096	1874558	0.04	1945181165	211956	0.00
SC+E	17748650	6636795	2592980	15155670	4043815	0.11	0	0	0
E+SD	3542841	2517959	1100345	2442496	1417614	0.65	-1515683079	126903	NaN
SC+F	20565527	6815904	2950767	17614760	3865137	0.05	0	0	0
SD+F	10650328	3277956	1690070	8960258	1587886	0.04	1599914993	217469	0.00
SC+F+E	13327809	5251111	2035187	11292622	3215924	0.12	0	0	0
E+SD+F	3356494	2385845	1040035	2316459	1345810	0.65	-1705230362	117697	NaN

(c)

Table B.37: The raw and EPOR-relative exploration data and reduction data from sleeping barbers SB2 input program.

Bibliography

[ACSE99] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan J. Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In *Proceedings of Static Analysis Symposium (SAS'99)*, pages 19–38, 1999. 3.2.2, 3.8

[AH90] Hiralal Agarwal and Joseph H. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256. ACM Press, 1990. 5

[AH03] Mark Allen and Susan Horwitz. Slicing Java Programs that Throw and Catch Exceptions. In *Procedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '03)*, pages 44–54. ACM, June 2003. 2.6

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. 3.6.4, 3.8

[BH93] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 206–222. Springer-Verlag, 1993. 2.1.1, 2.3, 2.6

[BH99] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. *ACM SIGPLAN Notices*, 34(10):35–46, 1999. 3.8

[Bla99] Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. *ACM SIGPLAN Notices*, 34(10):20–34, 1999. 3.8

[BP96] Giangranco Bilardi and Keshav Pingali. A Framework for Generalized Control Dependences. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 291–300, Philadelphia, Pennsylvania, United States, 1996. ACM, ACM Press New York, NY, USA. 2.2.1, 2.3, 2.6

[BR01] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Proceedings of 16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, Tampa Bay, FL, USA, Oct 2001. 3.8

[BS96] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, 1996. 3.3.3, 3.6.4

[CB01] Zhengqiang Chen and Xu Baowen. Slicing Concurrent Java Programs. *SIGPLAN Notices*, 36(4):41–47, April 2001. 5, 5.8

[CCL98] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology – Special Issue on Program Slicing*, 40:595–607, 1998. 5

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-state Models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 439–448, June 2000. 5.4.4, 5.6.3

[CGS+99] J. D. Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape Analysis for Object Oriented Languages. Application to Java. In *Proceedings of Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 1–19, Denver, CO, USA, Oct 1999. ACM. 3.8

[CLL+02] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, June 2002. 3.8

[CLS01] J. D. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical report, IBM Research Division, Thomas J. Watson Research Centre, 2001. 3.8

[Das00] Manuvir Das. Unification-based Pointer Analysis with Directional Assignments. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 35–46, June 2000. 3.8

[DHH+06] Matthew B. Dwyer, John Hatcliff, Mattew Hoosier, Venkatesh Ranganath, Robby, and Todd Wallentine. Evaluating the Effectiveness of Slicing for Model Reduction of Concurrent Object-Oriented Programs. In *Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, pages 73–89, 2006. 5.4.4

[DHRR04] Matthew B. Dwyer, John Hatcliff, Robby, and Venkatesh Prasad Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Formal Methods in System Design*, 25(2-3):199–240, 2004. 6, 6.1, 2, 6.4.2, 6.5

[EMCGP99] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999. 2.3, 6.1

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000. 3.8

[FF01] Cormac Flanagan and Stephen N. Freund. Detecting Race Conditions in Large Programs. In *Proceedings ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, 2001. 3.8

[FF04] Cormac Flanagan and Stephen N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of Symposium on Principles of programming languages (POPL)*, 2004. 6.5

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of Symposium on Principles of programming languages (POPL)*, 2005. 6, 6.3

[FQ03a] Cormac Flanagan and Shaz Qadeer. Transactions for Software Model Checking. *Electronic Notes in Theoretical Computer Science*, 89, 2003. 6.5

[FQ03b] Cormac Flanagan and Shaz Qadeer. Types For Atomicity. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in Languages Design and Implementation (TLDI)*, pages 1–12, 2003. 6.5

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996. 4.2

[GJS00] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, second edition, 2000. 3.2.1, 4

[God95] Patrice Godefroid. *Partial Order Methods for the Verification of Concurrent Systems*. PhD thesis, Facultè des Sciences Appliquèes, Universite De Liege, 1995. 6, 6.1, 6.2.1, 6.5

[HCD+99] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In *Proceedings of the 1999 International Symposium on Static Analysis (SAS'99)*, Lecture Notes in Computer Science, pages 1–18, Sept 1999. 2.6, 3.1.2, 3.2.2, 3.2.3, 3.8, 5, 5.8

[HDD+03] John Hatcliff, William Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *Proceedings of the 2003 International Conference on Software Engineering (ICSE'03)*, May 2003. 1.2

[HDZ00] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing Software for Model Construction. *Journal of Higher-order and Symbolic Computation*, 13(4):315–353, 2000. A special issue containing selected papers from the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. 2.2.1

[HPR89] Susan Horwitz, Phil Pfeiffer, and Thomas W. Reps. Dependence Analysis for Pointer Variables. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI'89)*, pages 28–40. ACM, 1989. 2.6, 3.8

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using Dependence Graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI'88)*, volume 23, pages 35–46, 1988. 3.8

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Language and Systems*, 12(1):26–60, 1990. 2.6, 5, 5.1.2

[HS04] Christian Hammer and Gergor Snelting. An Improved Slicer for Java. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, pages 17–22, 2004. 3.8, 5.8

[HU74] M. S. Hecht and J. D. Ullman. Characterizations of Reducible Flow Graphs. *Journale of ACM*, 21(3):367–375, 1974. 5

[IT98] Lynette I.Millett and Tim Teitelbaum. Slicing Promela and its applications to model checking, simulation, and protocol understanding. In *Proceedings of the 4th International SPIN Workshop*, pages 75–83, 1998. 2.6

[JP93] Richard Johnson and Keshav Pingali. Dependence-based Program Analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–89, 1993. 2.6

[KLM+98] Robert P Kurshan, Vladdimir Levin, Marius Minea, Doron Peled, and Husnu Yenigun. Static Partial Order Reduction. In *Proceedings of Tools and Algorithms for the Construction and Analysis of System (TACAS'98)*, pages 345–357, 1998. 6.5

[KP92] Shmuel Katz and Doron Peled. Defining Conditional Independence Using Collapses. *Theoretical Computer Science*, 101(2):337–359, 1992. Selected papers of the International BCS-FACS Workshop on Semantics for Concurrency. 6.1, 3

[Kri98] Jens Krinke. Static Slicing of Threaded Programs. In *Proceedings ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada, June 1998. ACM SIGPLAN Notices 33(7). 2.6, 3.1.2, 3.8, 5.8

[Kri02] Jens Krinke. Evaluating Context-Sensitive Slicing and Chopping. In *Proceedings of Internationational on Software Maintainance (ICSM'02)*, 2002. 5.1.2, 13, 5.8

[Kri03a] Jens Krinke. *Advanced Slicing of Sequntial and Concurrent Programs*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2003. 5

[Kri03b] Jens Krinke. Barrier Slicing and Chopping. In *Proceedings of Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 81–87, 2003. 5.6.1

[Kri03c] Jens Krinke. Context-Sensitive Slicing of Concurrent Programs. In *Proceedings of ESEC/SIGSOFT FSE'03*, pages 178–187, 2003. 5.1.3, 5.3.5, 5.8

[Kri04] Jens Krinke. Context-Sensitivity Matters, But Context Does Not. In *Proceedings of Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 29–35, September 2004. 5.6.2, 5.8

[LH96] Loren Larsen and Mary Jean Harrold. Slicing Object-Oriented Software. In *Proceedings of International Conference on Software Engineering (ICSE'96)*, pages 495–505, 1996. 3.8

[LH98] Donglin Liang and Mary Jean Harrold. Slicing Objects Using System Dependence Graphs. In *Proceedings of International Conference on Software Maintenance (ICSM'98)*, pages 358–367, 1998. 3.8

[LH99] Donglin Liang and Mary Jean Harrold. Efficient Points-to Analysis for Whole-Program Analysis. In *Proceedings of ESEC/SIGSOFT FSE'99*, pages 199–215, 1999. 3.8

[LH01] Donglin Liang and Mary Jean Harrold. Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses. In *Proceedings 8th International Static Analysis Symposium (SAS 2001)*, volume 2126 of *Lecture Notes in Computer Science*, pages 279–298. Springer, July 2001. 3.8

[LH03] Ondej Lhótak and Laurie J. Hendren. Scaling Java Points-to Analysis Using SPARK. In Görel Hedin, editor, *Proceedings of Compiler Construction (CC'03)*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169. Springer, April 2003. 3.4.1, 3.6.4, 3.8

[Lho02] Ondrej Lhotak. Spark: A Flexible Points-to Analysis Framework for Java. Master's thesis, School of Computer Science, McGill University, December 2002. 5.4.4

[Lho06] Ondrej Lhotak. *Program Analysis using Binary Decision Diagrams*. PhD thesis, School of Computer Science, McGill University, January 2006. 14

[LR94] P. Livadas and A. Rosenstein. Slicing in the presence of pointer variables, 1994. 3.8

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996. 4.2

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999. 4, 4.2, 5.6.3

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, Prentice Hall Europe, Campus 400, Marylands Avenue, Hemel Hempstead, Hertfordshire, HP2 7EZ, 1989. ISBN: 0-13-115007-3. 2.4.1

[MRR02] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA'02)*, pages 1–11, 2002. 3.8

[Muc97] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers. Inc., San Francisco, California, USA, 1997. 2.1.1, 2.1.3, 3.1.1

[NAC98] G. Naumovich, G. Avrunin, and L. Clarke. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. Technical Report UM-CS-1998-044, University of Massachusetts, Amherst, October 1998. 3.2.2, 3.8

[Nan01] Mangala Gowri Nanda. *Slicing Concurrent Java Programs: Issues and Solutions*. PhD thesis, Indian Institute of Technology, Bombay, November 2001. 3.8, 5, 5.1.3, 5.8

[NR00] Mangala Gowri Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA'00)*, pages 180–190, 2000. 5.8

[PC90] A Podgurski and L Clarke. A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(8):965–979, 1990. 2.1.1, 2.2.1, 2.3, 2.6

[RAB+04] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer, and John Hatcliff. A New Foundation For Control-Dependence and Slicing for Modern Program Structures - Technical Report #8. Technical Report 8, Kansas State University, 2004. This is available at http://projects.cis.ksu.edu/docman/?group_id=12. 2

[RAB+05] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B.Dwyer, and John Hatcliff. A New Foundation For Control-Dependence and Slicing for Modern Program Structures. In *Programming Languages and Systems, Proceedings of 14th European Symposium on Programming, ESOP 2005*. Springer-Verlag, April 2005. Extended version is available at http://projects.cis.ksu.edu/docman/?group_id=12. 1.2, 2

[Ran02] Venkatesh Prasad Ranganath. Object-Flow Analysis for Optimizing Finite-State Models of Java Software. Master's thesis, Department of Computing and Information Science, Kansas State University, 2002. (document), 3.3.3, 3.4.1, 3.6.4, 3.7.2, 3.7, 3.8, 5.4.4

[RDHI03] Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. Space-Reduction Strategies for Model Checking Dynamic Systems. In *In Proceedings of the 2003 Workshop on Software Model Checking (SMC 2003)*, July 2003. 6.4.2

[RH04] Venkatesh Prasad Ranganath and John Hatcliff. Pruning Interference and Ready Dependences for Slicing Concurrent Java Programs. In Evelyn Duesterwald, editor, *Proceedings of Compiler Construction (CC'04)*, volume 2985 of *Lecture Notes in Computer Science*, pages 39–56. Springer-Verlag, March 2004. 3

[Ric75] Richard J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717–721, December 1975. 6.5

[RR95] Thomas Reps and Genevieve Rosay. Precise Interprocedural Chopping. In *SIGSOFT '95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 20, pages 41–52. ACM, ACM, 1995. 5.1.2, 5.3

[Ruf00] Erik Ruf. Effective Synchronization Removal for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI'00)*, pages 208–218, June 2000. 3.2.2, 3.3, 3.8

[SR01] Alexandru Sălcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. *ACM SIGPLAN Notices*, 36(7):12–23, 2001. 3.8

[Ste96] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Conference Record of the 23th Annual ACM Symposium on Principles of Programming Languages(POPL'96)*, pages 32–41. ACM Press, Jan 1996. 3.8

[Sto02] Scott Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, 2002. 6, 6.1, 6.5

[SUL00] Scott D. Stoller, Leena Unnikrishnan, and Yanhong A. Liu. Efficeint Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In *Computer Aided Verification (CAV 2000)*, pages 264–279, 2000. 6

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995. 2.1.3

[Ven91] G. A. Venkatesh. The Semantic Approach to Program Slicing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 107–119. ACM Press, 1991. 5

[VR00] Raja Vallée-Rai. SOOT: A Java Bytecode Optimization Framework. Master's thesis, School of Computer Science, McGill University, Oct 2000. 3.7.1

[VSD86] Alfred V.Aho, Ravi Sethi, and Jeffrey D.Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley International, 1986. 3.3.2

[Wei84] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984. 2.1.3, 2.4.1, 3.8, 5

[WL04] John Whaley and Monica S. Lam. Cloning-based Context-sensitive Pointer Alias Analysis using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144. ACM Press, 2004. 14

[Zha99] Jianjun Zhao. Slicing Concurrent Java Programs. In *Proceedings of the 7th IEEE International Workshop on Program Comprehension (IWPC'99)*, pages 126–133, May 1999. 3.2.2, 5