UNIVERSITY OF CALIFORNIA,
IRVINE

Reusable Method Summaries for Improving Performance of Dynamic Dependence Analysis

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Vijay Krishna Palepu

Dissertation Committee:
Associate Professor James A. Jones, Chair
Professor Crista Lopes
Associate Professor Guoqing Xu

2017

DEDICATION

To my Mother, who taught me how to read and count.
To my Father, who taught me how draw loops in flow charts.
Together, they set me on a path towards engineering software.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like to thank my advisor, James Jones for the years of guidance, for helping me refine my ideas, for challenging my biases, for teaching the value of taking ownership of one's work, and generally for being a great mentor. Aside from being a great mentor, he has always been a calming presence whenever I faced a seemingly insurmountable problem — thank you James.

I would like to thank my committee members Crista Lopes and Harry Xu, who not only helped me improve this work with their constructive feedback, but wholeheartedly supported my Ph.D. studies at UC Irvine. Harry Xu nurtured my early interest in researching program analysis and set me on a path that eventually led to this dissertation. Crista Lopes agreed to guide and co-author my very first research paper, which gave me the necessary confidence to pursue a Ph.D. in Software Engineering. Crista Lopes and Harry Xu have been mentors to me throughout my time here at UC Irvine, and I thank them both.

This endeavor, over six years, was possible only because of the unrelenting support from my parents, Lalita and Ravi Shankar Palepu. Since the moment I raised the prospect of graduate school with the idea of researching software, my parents did everything in their means to support me towards that goal. Their support was unconditional, and unwavering. And it is fair to say that they often expressed their support with critiques and inquiries of my day-to-day progress, which would make a seasoned researcher feel proud. However, in more ways than one, their constant reminders to focus on my work allowed me to make continuous progress that culminated in this dissertation. For all that, and so much more — thank you Mom, thank you Dad.

Sudha Palepu — an accomplished spatial and exhibition designer, and my sister — taught me how to courageously pursue a passion and how to be excellent. Her persistence and courage in studying and mastering design, at the highest levels of excellence, in the face of overwhelming skepticism towards the general field and idea of design was both breathtaking and instructive. Her experiences as a student of design taught me to pursue my own path in researching and engineering software, irrespective of popular opinion. Sudha, I thank you for your courage, and your support through out these years.

I want to thank my college sweetheart and soon-to-be-wife, Shruti Nayar. Shruti stood by me and supported my decision to pursue a Ph.D., far away from home. Irrespective of the vast, physical distance that separated us over the years, her love kept me strong throughout my studies. I can trace every major milestone in my Ph.D., to memorable moments in our time together. Shruti makes me want to be a better and stronger person everyday. It would have been hard to pursue this work without her love and support — thank you Shruti.

I thank my lab mates who read and reviewed my papers, helped me prepare for conferences, talks, and a whole host of other things and generally have been great friends these few years. I want to particularly thank my academic big brothers — Fransisco Servant and Nicholas DiGiuseppe — who took me under their wings since my early days at UC Irvine and made

me feel at home. It is fair to say that I might not have applied to the Ph.D. program had it not been for Nicholas. I also thank all my friends, from my days as an engineering student, who never ceased to be excited about every project I undertook during my Ph.D. studies. My friends — particularly Rajat, Swati, Anirban, and Paritosh — were my cheerleaders throughout my time as a student at UC Irvine.

I would finally like to acknowledge that this material is supported by the National Science Foundation, under grants CCF-1116943, CAREER CCF-1350837, CCR-0325197, CNS-1321179, CCF-140982, and CNS-1613023, and by ONR under grants N00014-14-1-0549 and N00014-16-1-2913.

CURRICULUM VITAE

Vijay Krishna Palepu

EDUCATION

Doctor of Philosophy in Software Engineering **2017**
University of California, Irvine *Irvine, California, USA*

Master of Science in Software Engineering **2017**
University of California, Irvine *Irvine, California, USA*

Bachelor of Engineering in Computer Engineering **2010**
University of Pune, India *Pune, Maharashtra, India*

RESEARCH EXPERIENCE

Graduate Research Assistant **2012–2017**
University of California, Irvine *Irvine, California, USA*

TEACHING EXPERIENCE

Teaching Assistant **2012–2013**
University of California, Irvine *Irvine, California, USA*

Reader **Spring 2012**
University of California, Irvine *Irvine, California, USA*

PROFESSIONAL EXPERIENCE

Software Engineering Intern **Summer 2016**
Microsoft, Redmond *Redmond, Washington, USA*

Software Engineering Intern **Summer 2015**
Microsoft, Redmond *Redmond, Washington, USA*

REFEREED JOURNAL PUBLICATIONS

Palepu, Xu and Jones, **Dynamic Dependence Summaries** **2017**
ACM Transactions on Software Engineering and Methodology

REFEREED CONFERENCE PUBLICATIONS

Palepu and Jones, **Revealing Runtime Features and Constituent Behaviors within Software** **2015**
3rd IEEE International Working Conference on Software Visualization

Reddy, Kim, Palepu and Jones, **SPIDER SENSE: Software-Engineering, Networked, System Evaluation** **2015**
3rd IEEE International Working Conference on Software Visualization

Palepu and Jones, **Discriminating Influences among Instructions in a Dynamic Slice** **2014**
29th IEEE International Conference on Automated Software Engineering

Palepu and Jones, **Visualizing Constituent Behaviors within Executions** **2013**
1st IEEE International Working Conference on Software Visualization

Palepu, Xu and Jones, **Improving Efficiency of Dynamic Analysis with Dynamic Dependence Summaries** **2013**
28th IEEE International Conference on Automated Software Engineering

Martie, Palepu, Sajnani and Lopes, **Trendy bugs: Topic trends in the Android bug reports** **2012**
9th IEEE Working Conference on Mining Software Repositories

SOFTWARE

Blinky `https://github.com/spideruci/blinky`
Java bytecode tool for runtime program analysis and instrumentation.

Spider SENSE `http://spideruci.github.io/spidersense/`
Real-time web-based software analysis dashboard and build infrastructure.

Cerebro `http://spideruci.github.io/cerebro/`
Interactive visualization of software program executions.

ABSTRACT OF THE DISSERTATION

Reusable Method Summaries for Improving Performance of Dynamic Dependence Analysis

By

Vijay Krishna Palepu

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2017

Associate Professor James A. Jones, Chair

Software engineers construct modern-day software applications by building on existing standard and third-party software libraries and components that they necessarily do not author themselves. Consequently, to dynamically analyze a contemporary software application, all transitively dependent external libraries must also be monitored and analyzed, at each layer of the software application's call stack. However, dynamically analyzing large and often numerous external libraries may prove prohibitively expensive. To address the expenses associated with analyzing dynamically observable behavior of software components, this work presents an approach to summarize and reuse the results of dynamic analysis. This work, specifically focuses on dynamic analysis that computes data- and control-dependencies between executing instructions, and the memory locations they access, by monitoring software-runs at a fine-grained instruction-level. As such, the approach in this work models and summarizes software component behavior as data- and control-dependencies between inputs and outputs of invocable methods within software components. Such behavior models for invocable methods are called method dependence summaries. Method dependence summaries may be created with static analysis, as has been done and studied by prior work. However, a novel contribution of this work is the creation and investigation of method summaries by runtime monitoring of executing instructions in software runs. Such dynamically inferred method summaries are called "dynamic dependence summaries." Additionally, the approach

to reuse a dependence summary — created statically or dynamically — for the purposes of dynamic dependence analysis is, in itself, an important contribution of this work. Software components, equipped with reusable dependence summaries, afford the omission of their exhaustive runtime monitoring and analysis during their future runs. Nonetheless, the reuse of dependence summaries would necessitate the abstraction of any concrete runtime information enclosed within the summary; thus potentially causing a loss in the information modeled by the dependence summary. As a result, benefits to the efficiency of dynamic analyses that use summarization may be afforded with losses of accuracy. This work investigates the resulting trade-offs between accuracy losses and performance gains with 83 system-wide executions across eight real-world software systems. The empirical results show that dynamic dependence summaries exhibit median precision scores of 1.0 when modeling individual method invocations, across multiple executions of the eight software systems — indicating acceptable levels of accuracy. Simultaneously, statically created dependence summaries exhibit median precision scores ranging from 0.5 to 1.0, when modeling method invocations. The results also show notable degrees of cost savings while using summarized analysis, with median trace size reductions of 61% across the 83 executions, when compared to traditional exhaustive dynamic dependence analysis.

Chapter 1

Introduction

As the needs of society are increasingly accomplished with software systems, and those software systems become more complex and interrelated, software developers are, to an increasing extent, building components of software that interact with and build upon existing software components. Rather than writing all needed functionality from the low-level operating system to the high-level client interfaces, developers regularly use features that were developed by others, provided by components such as APIs, libraries, middleware, and infrastructures. Today's reality was predicted nearly fifty years ago by McIlroy (1968).

Prior research has identified some of the challenges that can be faced when depending upon and assembling existing components, often referred to as "components, off-the-shelf" or *COTS*. One common challenge to reusing third-party components is that analysis tasks become increasingly expensive as the extent and depth of component reuse increases (*e.g.,* layer upon layer of transitive component reuse). To properly analyze the program, the effects of the underlying infrastructure, and all of its layered components, must be understood. As such, an analysis that is complete and exhaustive must analyze all transitively underlying components to determine how they affect the program under test.

Orso *et al.* (2001) discussed some of the challenges of performing analysis in the presence of external components and proposed abstract representations, *i.e., meta-data*, to provide information about component functionality. Later, Orso *et al.* (2007) extended these ideas for component meta-data by specifying a concrete meta-data scheme to enable regression-test selection in the presence of components. Although Orso's solution for regression-test selection provides a powerful solution for the specific task of regression-test selection, the challenges of performing analysis in the presence of external components extend to many other (more heavyweight) dynamic software-analysis tasks. For example, dynamic dependence analysis necessitates the tracing of data and control flows through all encountered libraries and components during the whole execution. Frequent profiling of methods in these large, and often numerous, libraries and components contributes extensively to the already-heavy run-time costs, making the analysis often prohibitively expensive even for modestly sized software applications.

An important approach to reduce the costs of program analyses and provide such meta-data is to *summarize* the behaviors of these components. Once generated, component summaries can be reused, or *applied*, for future executions of those components, to improve the efficiency for a given program analysis.

This work, with dynamic dependence analysis as its background, looks to compute dependence summary meta-data that will characterize and capture external effects of reused components, for a modern object-oriented language, by treating each method as a component unto itself. Such dependence summary meta-data, or simply dependence summaries, can then be reused by a dynamic dependence analyzer to model the external effects of methods during their future invocations.

Indeed, the static program analysis community has extensively studied summary-based program analysis, with the development of various techniques that summarize procedural effects to achieve modular and efficient program analyses techniques (*e.g.,* Salcianu & Rinard (2005);

Rountev *et al.* (2008); Xu *et al.* (2009); Yorsh *et al.* (2008); Dillig *et al.* (2011)). As such, statically analyzing a method body to summarize the heap-data effects of executing that method would be one approach to creating such method dependence summaries.

However, such modular, summary-based approaches were designed for static program analyses, and have not been employed to dynamic dependence analysis. Moreover, as shown by prior research (*e.g.,* Korel & Laski (1990); Zhang & Gupta (2004a)), static analysis can lead to overly-conservative modeling of heap data effects. Thus, method summaries that rely only on static analysis, may render overly-conservative approximations of runtime heap data effects, due to issues like dynamic dispatch of polymorphic methods, access of individual array elements, or dynamically observed control flow. For example, statically built method summaries are often imprecise in distinguishing among objects of different subtypes that share a common super-type, making it particularly difficult for use in a dynamic analysis.

Additionally, statically built summaries are only possible with access to the component binaries that are actually executed, which may not always be possible, *e.g.,* in situations where the components are loaded dynamically, or in situations where the the binaries are rewritten on-the-fly, *i.e.,* during the execution of the binaries.

Given such potential shortcomings with static analysis, this work approaches the creation of dependence summaries by dynamically observing data and control dependencies. Such method summaries, created with dynamically observed dependencies, are called "dynamic dependence summaries".

Dynamic dependence summaries, as presented in this work, are computed for a method by *dynamically* observing the control and data dependencies within the method's representative executions, where each execution of the method results in a distinct dynamic dependence summary. Such a set of dynamic dependence summaries, for a given method, are then *abstracted*, to remove any runtime data that is specific to actual executions. Abstraction

of dynamic dependence summaries enables the reuse of the summaries for modeling the external-effects of subsequent executions of the method.

As such, the core contribution of this work are two fold. First, this work uses dynamic data and control flow to create method dependence summaries that are abstract, *i.e.,* lacking in any runtime data and as such comparable with the summaries produced by static analysis. Second, this work uses abstract summaries, *i.e.,* summaries with no runtime context or data, towards the modeling of dynamic dependencies.

A summary-based approach to dynamic dependence analysis leads to improved efficiency, through reductions in execution-trace sizes, trace-recording times and dependence-analysis times. Dynamic dependence analysis forms the basis for a variety of dynamic techniques, such as dynamic program slicing (Agrawal & Horgan (1990)), bloat analysis (Xu *et al.* (2010)), information flow analysis (Newsome & Song (2005)), and potential parallelism detection (Holewinski *et al.* (2012)). Thus, a summary-based approach stands to improve the efficiency (in space and time) for all techniques that rely on dynamic dependence analysis.

Such potential gains in efficiency, due to using method dependence summaries, are met inaccuracies due to varying behavior across multiple invocations of the same method. Different invocations of the same method, may exhibit different external heap-data effects, due to (a) dynamic dispatch, and, (b) different control-flow paths; thus potentially resulting in differences in the different dynamic dependence summaries for the same method. First, differences in dependence summaries due to dynamic dispatch are handled by creating separate dynamic dependence summaries for each set of dynamically-observed argument types used to invoke the method. Second, in order to generically model the varying external-effects of a method due to varying control-flow, the dynamic summaries for different method invocations that share the same dynamically-observed argument types are abstracted and *aggregated* into a single dynamic dependence summary. The resulting aggregated, abstract dynamic dependence summary is used to generically model the behavior of any subsequent invocation of

the method, when invoked with a specific set of dynamically-observed argument types. An aggregated, abstract, dynamic dependence summary would resemble a statically created dependence summary, while presumably modeling fewer heap-data effects that were actually observed in specific invocations of the method.

However, aggregating different heap-data effects due to varying control flow may yet result in inaccuracies of two kinds. First, with dynamic dependence summaries, any reliance on dynamic data inherently renders the resulting dynamic dependence summaries unsound, since a given set of invocations of a method may not exhibit all possible control paths and the resultant external heap-data effects. Second, the aggregation of multiple dynamic dependence summaries for a given method leads to a generic model of the external effects of a method's invocation. Such a generic model of a method's external-effects, when used to describe the external-effects of a specific invocation of the method may introduce *spurious* dependence relationships. As such, an aggregate dynamic dependence summary, much like a static summary, introduces imprecisions within the dependence summary of a single invocation of a given method. Such imprecisions are likely to exist to a greater degree with statically created method dependence summaries, than with dynamically aggregated dependence summaries.

In light of the essential trade-off between accuracy and performance while using dependence summaries – created statically or dynamically – for dynamic dependence analysis, I propose the following statement as the thesis of this work:

> **Thesis:** Reusable dependence summaries for method invocations can reduce the computational costs involved in data- and control-flow based dynamic dependence analysis of software runs, while modeling such dynamic dependencies with moderate-to-high degrees of accuracy.

A reusable dependence summary for a method is expressed in terms of data- and control-flow based dependencies between the inputs and outputs of the method. An important

facet of reusable dependencies is the absence of execution-specific information which allows them to be applied, or reused for modeling the dependencies between inputs and outputs of future invocations of the summarized method. This absence of invocation-specific data in the summaries is the root cause of the information loss and as such the accuracy losses. As such, the central goal of evaluating the thesis statement is to measure the performance gains when using dependence summaries, while inspecting the resulting losses in accuracies while modeling dependencies.

The rest of this dissertation is organized as follows. Chapter 2 details the motivation for this work by providing the necessary context for this work in terms of the actual problem being solved. This is followed by Chapter 3 that provides basic definitions used through Chapters 4 and 5. Chapter 4 establishes the conceptual framework for describing how dependence summaries should be created and used for dynamic dependence analysis. Dynamic dependence summaries are defined and discussed in great detail in Chapter 5, with the discussion on the challenges in Chapter 4 serving as the background. Chapter 6 in concert with the thesis statement, provides an overview of the directions of empirical investigation for this work. The experimental setup and results of four empirical investigations are described in Chapters 7 to 10. The results of the different experiments are further discussed as a whole in Chapter 11. Finally, after brief overview of related work (Chapter 12), this dissertation concludes in Chapter 13 that enumerates the limitations and contributions of this work, while describing the future directions of research that this work stands to motivate.

Chapter 2

Motivation

The original motivation for dependence summaries stems from the overwhelming, and the potentially prohibitive costs of profiling the execution of standard and third-party libraries that support a software application's execution; in other words, methods that are typically ancillary to the development task at hand. As such, summarizing the effects of such methods would bring down profiling and analysis costs. However, it is important to note that this work does not provide any definition of what constitutes as ancillary for a development task. This work assumes that the demarcation of methods as ancillary (or not) for a development task can be calibrated as needed by the developer.

Software applications frequently execute methods from standard and third-party libraries, resulting in significant runtime spent in the execution of library code. To highlight this idea, Figure 2.1 pictorially illustrates three facets of an application's execution: (a) the chronology of executing instructions – progressing from left-to-right; (b) a colored distinction between the execution of application code (in purple) and library code (in orange); and (c) the "call-depth", *i.e.,* the depth in the method-call stack, at which the instructions are executing. The top-to-bottom progression of instructions shows method invocations and thus a deepening

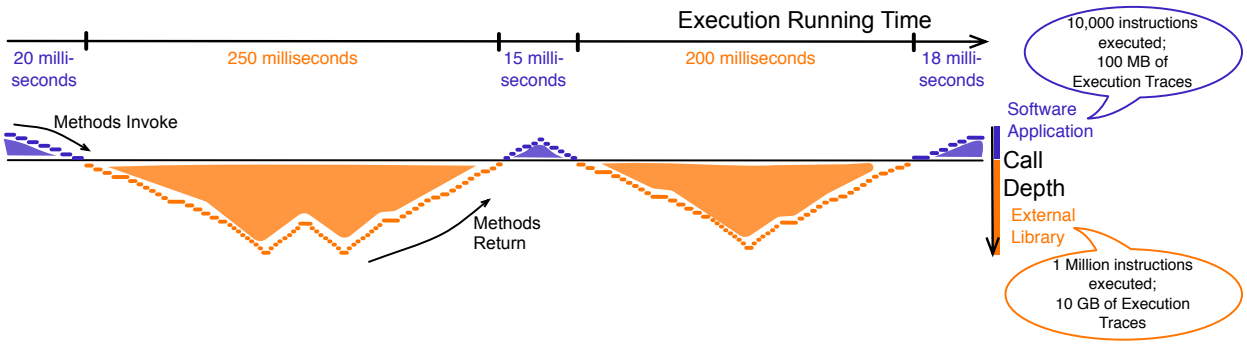


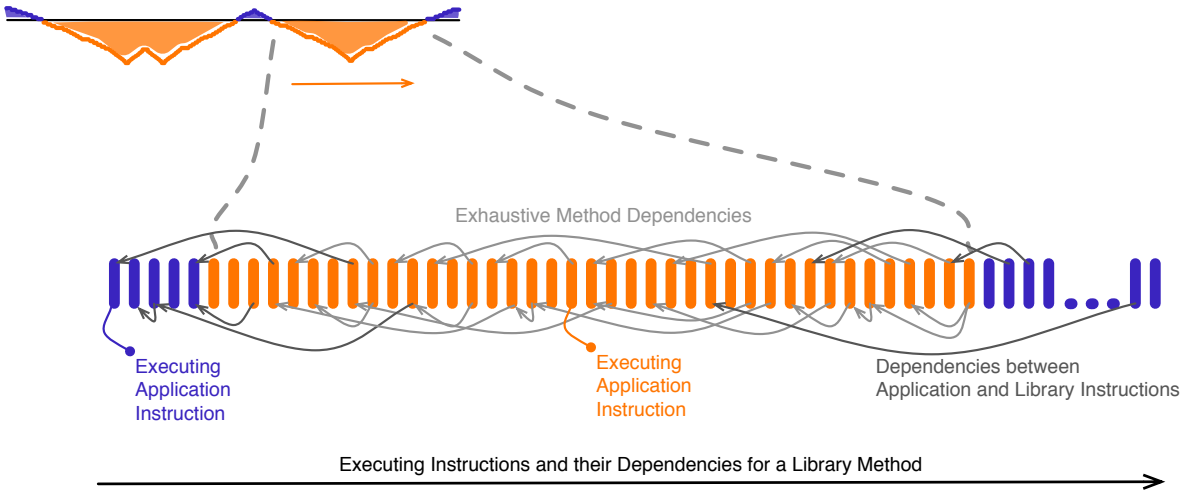
Figure 2.1: A pictorial illustration of the profiling costs of library and application code.

call-stack; while the bottom-to-top progression shows method returns.

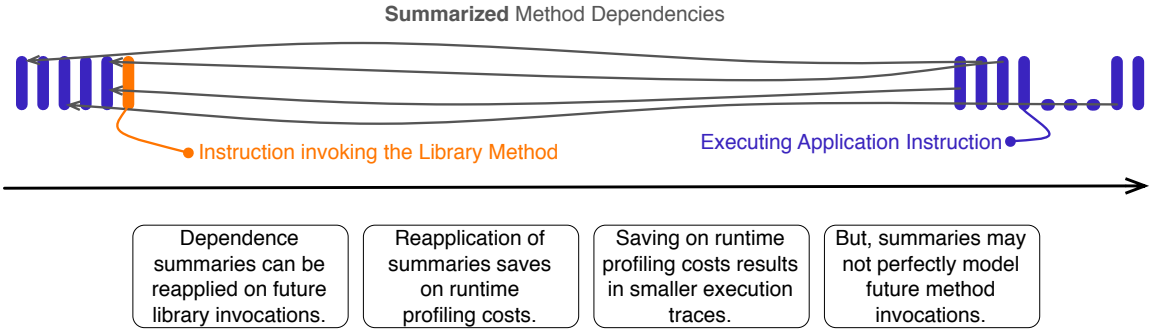
The wide and deep valleys of orange in Figure 2.1 that depict library code execution, pictorially show the significant execution-time spent in executing library code with extensive and deep call-stacks, in comparison with the brief sections of purple that depict execution of application code.

While such standard and third-party libraries are ancillary to the actual implementation and development of a software application, these external libraries exert substantial influence on the eventual executions of the software application. As such, profiling the execution of library code becomes an important, and expensive facet of runtime analysis of software systems. This notion of a runtime dependence on external libraries is illustrated in Figure 2.2a.

Figure 2.2a shows a "zoomed-in" illustration of the executing instructions within a library method call, invoked from within the application code. The illustration shows how executing instructions are data and control dependent on instructions executed *a priori*. In particular, the illustration depicts how executing library instructions depend on application instructions, and in turn executing application instructions are dependent on the execution of library instructions. Dependencies between the execution of library and application instructions necessitate the runtime analysis of library instructions, in order to comprehensively analyze the software application's executions. Moreover, such runtime analysis of library in-



(a) A pictorial illustration of the influence and dependence of executing instructions in a library method's invocation (in orange) on the execution of application code (in purple).



(b) Summarized Method Dependencies for a single execution of a Library method.

Figure 2.2: A pictorial illustration of the concept of Dynamic Dependence Summaries.

structions cannot be ignored even as they substantially contribute to an application's runtime as depicted in Figure 2.1.

The goal of exhaustively profiling library invocations is to monitor their *effects* on the execution of the software application. As such, a natural idea to reduce the cost for analyzing library method invocations would be to summarize their *effects*. The summaries of the *effects* of library method invocations can then be reused for future library method invocations, to model their influence on the software application's execution – without exhaustively profiling the library method invocations. This notion is illustrated in Figure 2.2b, where instead of depicting all executed instructions of the library method invocation and their dependencies, like in Figure 2.2a, only the summarized method dependencies and the execution of the

application's instructions are shown.

Unsoundness and Imprecision. It is important to note that the summary-based dynamic dependence analysis can introduce both unsoundness and imprecision, while modeling dependencies between the inputs and outputs of future method invocations. On one hand, when using dynamically created summaries the quality of a method's summary relies on the coverage of the tests used to train the summary. Thus, a summary may miss certain dependence relationships due to the lack of test cases. On the other hand, consider using static dependence summary, or a dynamic summary that aggregates information from multiple executions of the method. The application of a static or dynamically aggregated summary for a specific invocation may generate additional *spurious* dependence relationships that would not have been added in a regular dynamic dependence analysis. However, the original aim of this work was to bring down the overwhelming and often prohibitive costs of execution profiling for dynamic dependence analysis. As such, the goal of this work with dynamic dependence summaries, despite their obvious potential for introducing inaccuracies, is to make dynamic dependence analyses feasible, especially when the costs of runtime analysis are prohibitive.

Pilot Study. The discussion above was guided by conceptual illustrations to describe the potentially prohibitive expense associated with runtime profiling of external components for dynamic dependence analysis. In order to study this expense of analyzing software executions, I carried out a pilot study. As a part of this pilot study, I investigated the number of runtime-instructions[1] that get executed in a typical execution of real-world, large-scale software systems, and would need to be recorded and analyzed. Additionally, in order to investigate how summarizing effects of external components can assist with the expenses of dynamic analysis, I also studied the number of instructions executed from within the Standard

[1]A "runtime-instruction", or an "instruction-instance", is a dynamic instantiation of a static program instruction. A single static program instruction can be executed multiple times, and can lead to multiple runtime-instructions.

Java Library (*i.e.,* `rt.jar`), which is essentially an external library. The following five real world, large-scale subjects were used for this pilot study — ANTLR (> 35KLOCs), BLOAT (> 41KLOCS), JYTHON (> 245KLOCs), FOP (> 102KLOCs), PMD (> 60 KLOCs). These large scale subjects were obtained from the DACAPO performance benchmarks Blackburn *et al.* (2006).

The results of the pilot study suggest that the executions of the large-scale software systems that I selected, comprise of 59.9×10^9 runtime instructions on average; with the breakdown of runtime instructions for each software system as follows: ANTLR (4.2×10^9 runtime instructions), BLOAT (214.8×10^9 runtime instructions), JYTHON (51.9×10^9 runtime instructions), FOP (0.7×10^9 runtime instructions), PMD (28.4×10^9 runtime instructions). A reasonably efficient scheme to record the execution of each runtime instruction in an execution trace would approximately require 160 bits or 20 bytes of memory — 32 bits each to record the runtime-instruction's thread-ID, owner-class, owner-method, bytecode-offset and operand value.

Given this scheme and these subjects, an average execution of these benchmarks requires one terabyte (1TB) of disk storage to record an execution trace of nearly 60×10^9 runtime-instructions. Apart from storing executions at such scale, analyzing them would also be a significant challenge unto itself.

When counting only those runtime-instructions that were executed as a consequence of executing a method from the Java Standard Library (*i.e., rt.jar*), more than 32.5×10^9 runtime instructions were executed, with the breakdown for each software system as follows: ANTLR (1.6×10^9 runtime instructions), BLOAT (135.3×10^9 runtime instructions), JYTHON (13.1×10^9 runtime instructions), FOP (0.4×10^9 runtime instructions), PMD (12.3×10^9 runtime instructions).

As such by summarizing only the lowest level library (*rt.jar*), on average, more than 45% of

the costs associated with recording and analyzing runtime instructions maybe eliminated for the specific program executions used in this pilot study. Such reductions are significant, and can be made possible for dynamic dependence analyses by appropriately summarizing the effects of library-specific runtime-instructions. Further, such projected reductions in space requirements were achieved when the Standard Java library, was treated as the sole external component for all executions. Software applications often use several external libraries, thus presenting further avenues for summarization and potential cost savings.

Usage Scenario: Debugging. As discussed in Section 1, dynamic dependence analysis serves as the basis of several runtime-analysis techniques, many of which are used towards various software engineering tasks and applications. However as discussed above, dynamic dependence analysis is often expensive given the prohibitive nature of runtime software analysis. To further motivate the use and potential impact of method dependence summaries, a final portion of this chapter will consider the use of dynamic dependence analysis, and consequently dependence summaries, towards an actual software engineering task: debugging.

Agrawal & Horgan (1990) introduced the notion of using dynamic dependence analysis towards assisting software debugging. Agrawal & Horgan (1990) used the idea of tracing dependencies between different executing instructions within a the dynamic dependencies exhibited within a software execution, to identify the root cause of a developer-identified program error. They described such an approach of tracing dynamic dependencies as dynamic slicing — a dynamic version of static slicing that was originally introduced by Weiser (1981). Such early works in program slicing — static and dynamic — observed that programmers begin their investigation of an externally visible software error at the program instruction that first manifests the error. For instance, consider that a software engineer identifies an instruction where the program crashes due to a null pointer exception. The programmer would then identify the reference (*e.g.,* a local variable, or field) that points to the `null` value, *i.e.,* the immediate cause of the null pointer exception. The programmer

would then reason about and identify the last instruction where the reference was assigned the `null` value. Identifying such an instruction may result in a series of additional, similar, investigatory steps that may perhaps highlight the root cause for the null value, prompting the developer to fix such a fault in the program. In the absence of any automated analysis, the programmer would be left to debug the program by following by an iterative, and manual process of placing breakpoints at different instructions of the program, and stepping through the execution of individual instructions of the program to identify the root cause of the null pointer exception. Such an investigation is often based on a hit-and-trial approach.

With the aid of automated analyses, such as dynamic dependence analysis and slicing, works such as that by Agrawal & Horgan (1990) posited that a programmer will be able to identify the root cause of a program error by simply tracing the dependencies for individual, executing instructions (as shown in Figure 2.2a). Such a vision of debugging that used dynamic dependence analysis was further demonstrated by Ko & Myers (2008) with their work on Whyline. Whyline allowed the navigation of dependencies between executing instructions in a program run by allowing programmers to ask questions about the program state at any given point in the execution. For instance, the programmer may simply ask, "Why is the local variable `x` assigned to `null`?", at the exception-inducing instruction, and a tool such as Whyline would trace the dynamic dependencies and automatically navigate the developer to the last instruction that assigned the `null` value to the local variable `x`. This would not require the developer to manually identify such an instruction by placing breakpoints and stepping through lines of code — amounting to a hit-and-trial approach.

Such a vision of program debugging — guided by dynamic dependence analysis — is stymied by the potentially prohibitive costs of dependence profiling as discussed and illustrated in the previous sections of this chapter. Using method dependence summaries to bring down the profiling costs for dynamic dependence analysis is an underlying motivation of this work. However, the larger motivation behind bringing down the costs of dynamic dependence

analysis — with method summaries — is to make other client analyses that use dynamic dependencies, such as dynamic slicing, more feasible and practical. Making such dynamic analyses more practical may aid in the creation of automated software engineering tools that assist software programmers in their development tasks such as debugging.

Chapter 3

Definitions

The definitions of dependence summaries, their aggregation and reuse, are rooted in the notions of inputs and outputs to method-invocations; and the notion of dynamic dependence. As such, this chapter first defines what I mean by inputs and outputs for a method-invocation, followed by a brief overview of the definitions of dynamic dependence.

3.1 Method Invocations: Inputs and Outputs

I now introduce the concepts of inputs and outputs for a method under the framework of access-paths and object-graphs that I define as follows.

> **Definition 3.1** (Access Path (AP))**.** *For a member (f), i.e., field or array-element, an access-path is a sequence of memory locations, leading to the access of the member (f), starting from a reference-typed object (o), with each preceding location in the sequence being the owner of the succeeding location, and each succeeding location being the member of the preceding location.*

Notationally, an access-path from an object o to field f, with many intermediate locations $f_1, f_2 \ldots f_n$ is denoted as $[o \rightarrow f_1 \rightarrow f_2 \rightarrow \ldots \rightarrow f_n \rightarrow f]$, or simply as $[\![o \dashrightarrow f]\!]$. In addition, an access-path to an array-element within an array f at index l is denoted as, $[\![o \dashrightarrow f[l]]\!]$.

When stated differently, access-paths chain together a set of memory locations into a series of *owner-member* relations (between the set of memory locations), in turn informing how a certain field or array-element is accessed starting from a reference-typed object. This allows us to introduce the notion of an object graph that can be thought of as a set of access-paths that all start from a common reference-typed object.

Definition 3.2 (Object Graph (OG))**.** *An object graph (g), is a graph rooted at a reference-typed object (o); contains a set of memory locations as nodes that are accessible from the reference-typed object (o); via edges that represent* owner-member *relationships, and connect a owner memory location (reference-typed object) and a member memory location (field or array-element).*

Note that the only way to access a memory location in an object graph (g) is to perform a sequence of member (field or array-element) dereferences on the root object that in turn, is represented as the access-path of a member location in an object-graph. As such each member location can be expressed as an access-path within an object-graph. Figure 4.2 shows two example object-graphs reachable from parameter o^{02} for an invocation of the method `void add(int i)` before and after the execution of line 06. As an extension and generalization of the example in Figure 4.2 I now define the concept of a set of object-graphs that are accessible from the arguments of a method invocation.

In this work, a method-invocation refers to a specific instance in a series of runtime invocations of a single method, within a program execution; and is denoted as: $c : m(a_0, a_1, \ldots, a_n)$, where,

- c uniquely identifies a method invocation event;
- m is the method being invoked;
- $a_0, a_1, \ldots, a_n$ are actual arguments to the method;

Definition 3.3 (Accessible Object Graph Set (AOGS))**.** *For a method-invocation $c : m(a_0, a_1, \ldots, a_n)$ the accessible object graph set (AOGS), is a set of object graphs that are rooted at any reference-typed objects pointed-to by the arguments $(a_0, a_1, \ldots, a_n)$, and represented as $\mathcal{G}_c$.*

N.B., $\mathcal{G}_c^{pre}$ and $\mathcal{G}_c^{post}$ denote the accessible object-graph sets (AOGS's) immediately before and after the method-invocation $c : m(a_0, a_1, \ldots, a_n)$, respectively.

Such a conceptualization of how memory locations are related and accessed during method-invocations enables us to account for the situation discussed in Chapter 4 (Challenge 1), where a method-invocation potentially uses not only the arguments to the invocation, but any memory location reachable from those arguments. Using such concepts of access-paths, object-graphs and object-graph sets, I now define the inputs and outputs to a method-invocation.

Definition 3.4 (Input and Output Sets)**.** *For a method-invocation* $c : m(a_0, a_1, \ldots, a_n$*, the input set* $\mathcal{I}_c$ *and output set* $\mathcal{O}_c$ *are defined as,*

1. ***Input Set*** *(*$\mathcal{I}_c$*) is a set of memory locations that exist before the method-invocation, and can be read during the method-invocation, such that,*

   $$\mathcal{I}_c \subseteq \mathcal{G}_c^{pre} \cup \mathcal{P}_c$$ *where,*

   - $\mathcal{G}_c^{pre}$ *is the AOGS immediately before the execution of* m *at* c*;*
   - $\mathcal{P}_c$ *is the set of memory locations representing parameters passed into* c*.*

2. ***Output Set*** *(*$\mathcal{O}_c$*) is a set of memory locations that can be written-to during the method-invocation and are accessible after the completion of the method-invocation, such that,*

   $$\mathcal{O}_c \subseteq \mathcal{G}_c^{post} \cup \mathcal{L}_c \cup \{ret\} \cup \{g_{ret}\}$$ *where,*

   - $\mathcal{G}_c^{post}$ *is the AOGS immediately after the execution of* m *at* c*;*
   - ret *is the memory location containing the value to be returned;*
   - g_{ret} *is the object graph defined by* ret *(if* ret *is of reference type); and,*
   - $\mathcal{L}_c$ *is a set of integer indices used in the accesses of the arrays referenced by memory locations in* $\mathcal{G}_c^{pre}$*,* $\mathcal{G}_c^{post}$ *or* g_r*.*

Locations that are unreachable from a method argument, via such access paths, do not escape the method-invocation and have no impact beyond the scope of the method-invocation. Hence, such locations are discounted from a method-invocation's input and output sets and, eventually from the method's dependence summary. As such, the input and output sets can often be a subset of the accessible object-graph sets for a method-invocation, as portrayed

in Definition 3.4. Additionally, the indices used to access the arrays in the object graphs are included in the output set. Tracking these indices is necessary for the precise handling of array accesses, as described in Chapter 4. Note that this approach treats static fields, which are globally accessible and used in a method-invocation, as additional arguments to a method. Similarly, since Java is a call-by-value language, where the values of the actual parameters cannot be changed by the method, $\mathcal{P}_c$ — locations serving as arguments to a method-invocation — is not part of the output set.

3.2 Dynamic Dependencies

After defining the input and output sets for a method-invocation, the next step towards defining a dynamic dependence summary is to define dynamic dependencies between the inputs and outputs of a method-invocation. I now present the definitions of dynamic data and control dependence between memory locations in a program execution.

In this work, I refer to the execution of a static program instruction, during the program's execution, as an instruction-execution event; and it is denoted using the form: a^j *i.e.,* the j^{th} execution of the static program instruction a.

Definition 3.5 (Dynamic Dependence)**.** *Given two memory locations, l_1 and l_2; memory location l_2 is said to be directly and dynamically dependent on memory location l_1 under the following two conditions:*

1. ***Dynamic Data Dependence.*** *An instruction-execution event (a^j) writes a value to the memory location l_2 that is computed using the value read from memory location l_1;*

2. ***Dynamic Control Dependence.*** *An instruction-execution event a^j that writes a value to the memory location l_2 and the occurrence of a^j was predicated on the value at memory location l_1.*

Using Definition 3.5 that defines relations of direct dependence between two memory locations, transitive dependence between two memory locations (l_1 and l_2) can be easily established by following a sequence of direct dependencies from one location (l_2) to the other (l_1). Based on this definition of dynamic dependence, relations of direct or transitive dependencies between memory locations in the input and output sets of a method-invocation can now be established. Such a set of dynamic dependencies between the input and output locations of a method-invocation is essentially the dynamic dependence summary of a single method-invocation.

Chapter 4

Challenges in Summarizing Dynamic Dependence Analysis

This chapter discusses the principle challenges involved in summarizing dependencies for the purposes of dynamic dependence analysis, as such acting as the foundation of this work. The discussion also includes an overview of a resolution for each challenge. The discussions of these challenges, is carried out in the context of a simple software program and it's execution that is used as a running example henceforth in this document. It is worth noting that while these challenges are described with a partial focus on creating dependence summaries dynamically, they provide the necessary foundations to apply statically created summaries for dynamic dependence analysis.

I start with describing the running example, followed by discussions for the following principle challenges: (1) defining dependence summaries with objects; (2) abstracting any concrete information; (3) accounting for varying method behavior; and (4) reusing abstract dependence summaries.

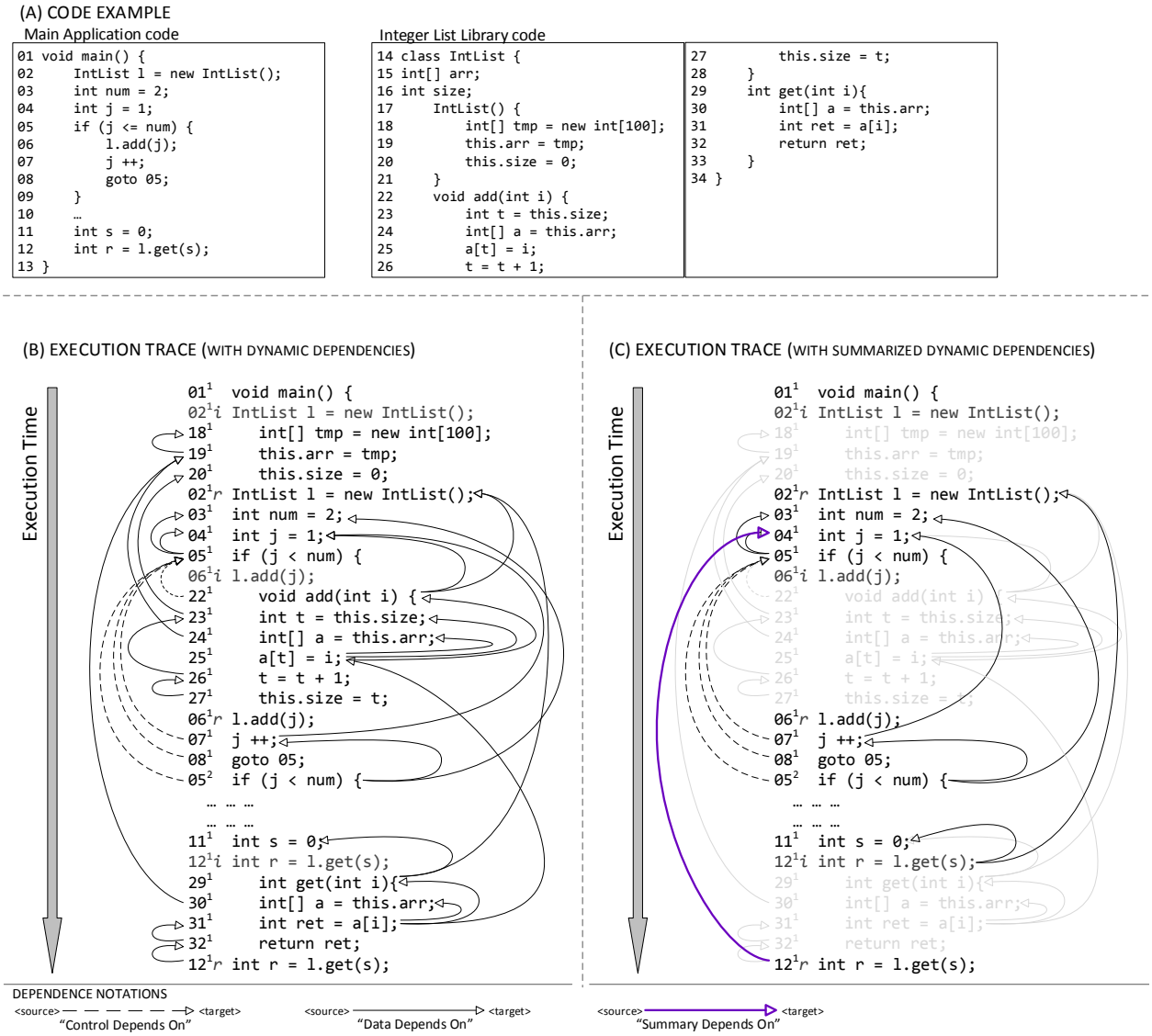


Figure 4.1: Example Program: Code & Execution Trace (with dynamic dependencies).

Running Example

Figure 4.1(a) shows the implementation of the main application (lines 01–13) and the library `IntList` that supports the storage and retrieval of integer values (lines 14–34) in the form of a list. The main application iteratively adds integer values (lines 03–09), into the `IntList` object created in line 02. After the addition of integer values, Lines 11 and 12 retrieve an integer value from the `IntList` object. The `IntList` library supports the creation of a new list of integers (lines 17–21); the addition of an integer value at the end of the integer-list

(lines 22–28); and the retrieval of an integer value at a designated position in the integer-list (lines 29–33).

Example Execution Trace

Figure 4.1(b) shows the execution trace of a single run of the application, along with the usage of the `IntList` library. Each line in the trace is an event depicting the execution of a single source code statement, and is represented by the source code statement itself along with the line number of the source code statement, annotated with an integer i representing the i^{th} execution of the statement. *N.B.,* a single statement can be executed multiple times in a single program execution. The execution starts with the invocation of the `main` method as shows in the first line of the execution-trace and progresses "downward" with every succeeding line in the trace, as shown in Figure 4.1. Execution events in the trace that depict invocation[1] of library methods are annotated with the alphabet 'i', *e.g.,* `06`^{1}*i* `l.add(j);`. Similarly, the completion of library methods' execution, after their return, is depicted with the alphabetical annotation 'r', *e.g.,* `06`^{1}*r* `l.add(j);`. Additionally, execution of methods and instructions within the `IntList` library are portrayed with textual indentations in the execution trace, to better represent the execution of library instructions.

Furthermore, dotted and solid edges, with triangular arrowheads are used to portray control and data dependencies, respectively, between different execution events in Figure 4.1(b). The dependence edges are drawn starting from the *dependent* execution event; with the edges ending (with the arrowhead) at the *dependee* execution event. For instance, the solid edge: `19`^{1} —▷ `18`^{1} implies that execution event `19`^{1} is data-dependent on execution event `18`^{1}.

[1]In this work, unless mentioned otherwise, a "method-invocation" or "invocation", refers to a specific instance in a series of runtime invocations of a single method, within a program execution.

4.1 Challenge 1: Defining Dependence Summaries with Objects

Horwitz *et al.* (1990) pioneered summary-based dependence analysis — a summary edge of a procedure relates an input parameter i with an output parameter o; depicting a possible (direct or transitive) dependence of the computation of the value in output o on the value in input i. Such summary edges between a procedure's inputs and outputs abstracts away intermediate dependence relationships within the procedure.

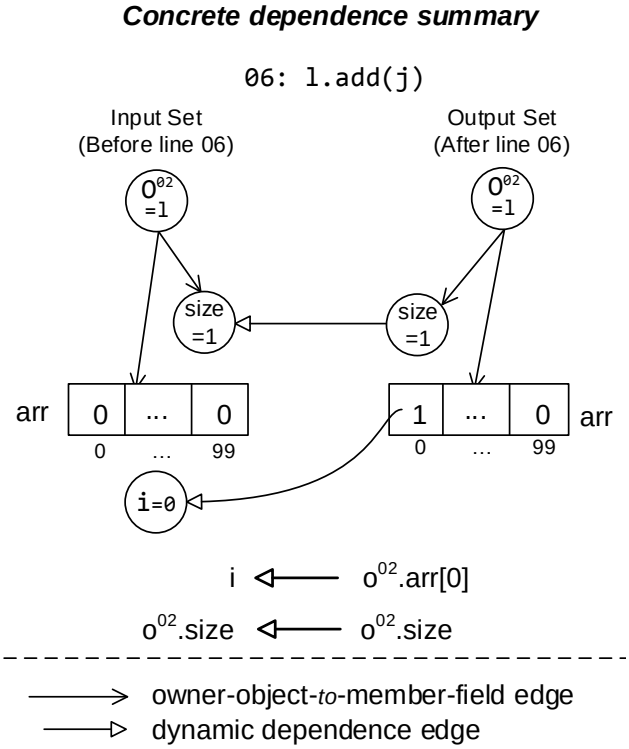


Figure 4.2: Concrete Dynamic Dependence Summary for the runtime invocation of the `add` method, as shown in the Example Program Execution Trace in Figure 4.1.

However, modern object-oriented languages, such as Java, impose new challenges in modeling such procedure inputs and outputs and the relationships between them, that existing techniques do not address. The notions of inputs and outputs for a method in an object oriented language, like Java, is much broader than those discussed in Horwitz *et al.* (1990),

because a method can access not only its arguments, but potentially all objects reachable from the arguments.

In this work specifically, for a single instance of a method-invocation, inputs and outputs are considered as a set of (heap or stack) locations where,

1. *input* locations exist before a given method-invocation, and can be read during the method-invocation, and
2. *output* locations can be written to during the method's invocation and are accessible after the completion of the method invocation.

Example

Figure 4.2 illustrates the dynamic dependence summary for the invocation `06[1]i 1.add(j);` — *i.e.,* the invocation's inputs, outputs and the dynamic dependencies between them. Figure 4.2 shows the input and output sets for the method invocation, on the left-hand and right-hand sides respectively. The input set contains the heap or stack locations that served as the method invocation's inputs. Similarly, the output set contains heap or stack locations that were modified during the method invocation and are available as its outputs. Arrows going from the output-set to input-set depict the dependencies (direct or transitive) between the inputs and outputs. Further, the *owner*-to-*member* relations between the different locations within the input or output sets are also depicted graphically with an arrow going from the owner object to the member. For instance, the relation $o^{02} \rightarrow size$ shows that $size$ is a member of the owner object o^{02}. Such *owner*-to-*member* relations are also represented textually with a dot-separator (.) between the owner and member (*e.g.,* $o^{02}.size$).

Now, the invocation `06[1]i 1.add(j);` for the method `void add(int i)` accepts as inputs the integer value j and receiver object l; and does not explicitly return any value. However,

a closer inspection suggests that the invocation `06[1] i 1.add(j);` results in the modification of the first element in the `arr` data structure (denoted as $o^{02}.arr[0]$ in Figure 4.2); and it updates the value in the field, $o^{02}.size$. As such, the array-element ($o^{02}.arr[0]$) and the updated field ($o^{02}.size$) are available as outputs after the return of the method. Moreover, the initial value of the field $o^{02}.size$ served as an input towards the field's update during the method-invocation, even though it was not passed as an actual argument to the method. Essentially, the inputs and outputs, for a specific method-invocation are not restricted to the method's actual arguments or a possible return value. A method's dependence summary should model such method inputs and outputs, along with the dependencies between them using the method-invocation's constituent data and control flow.

Critically, each location in the input and output sets (*e.g., size*), where necessary, is modeled using a combination of a root object (*e.g.,* o^{02}) and an *access path* that specifies how the location is accessed, from the parameter (*e.g.,* $o^{02}.size$), through a series of member dereferences. Such modeling of inputs and outputs, and the dynamic dependencies therein, would yield effective applications of dependence summaries for downstream client analyses. The concepts of inputs, outputs and access paths are formally defined in Chapter 3.

4.2 Challenge 2: Abstracting Concrete Summaries

Figure 4.2 depicts the inputs, outputs and the dependencies between them, for a specific method-invocation within a program execution. I call such a model of dependencies that is tied to a specific method-invocation, as a concrete dynamic dependence summary, or simply, concrete summary, as formally defined in Section 5.1. For example, the object o^{02} is assigned to the concrete value l in the concrete dependence summary, as shown in Figure 4.2. Since the value l is tied to the specific method-invocation `06[1] i 1.add(j);` it might not be valid for a different invocation of the method `void add(int i)`. Hence, it follows that such

concrete information in a dependence summary that is specific to a single invocation of a given method, prevents us from reusing, or *applying* the summary for other invocations of the method in question.

To enable the reuse of a concrete summary, the information tied to a specific method-invocation should be replaced with abstract information that may be applicable to all possible invocations of the method. In turn, the substitute abstract information should allow translation back to the concrete information that is specific to other invocations of the given method. In other words, the abstraction of concrete summaries should, in part, allow application of a method summary for all invocations of the method, instead of specific invocations.

Abstraction of method summaries is performed in this work by using symbolic names for method arguments. As illustrated in Figure 4.2, the actual arguments for the method-invocation `06[1]i 1.add(j);`, at line 02, are the concrete object l and the concrete variable j. Upon abstraction of the concrete dynamic dependence summary for the given method-invocation, the concrete object l and the concrete variable j are replaced by symbolic names p_0 and p_1, respectively. As depicted in Figure 4.3(a), the final set of *abstract summary edges* use the symbolic names p_0 and p_1, instead of their counterparts in the concrete summary.

4.2.1 Challenge 2.1: Abstracting Concrete Array Accesses

Precise handling of array accesses can be critically important in the dependence analysis of method-invocations within software systems. When an abstract summary involving an array access is applied at a method-invocation, one wishes to understand precisely which array element is used or defined inside the method execution. Without such information, spurious dependence relationships may be generated — any data retrieved from an array would depend on any data added into the array. Precise handling of array accesses is challenging because the index used to access the array is not projected as an output of the method, and thus,

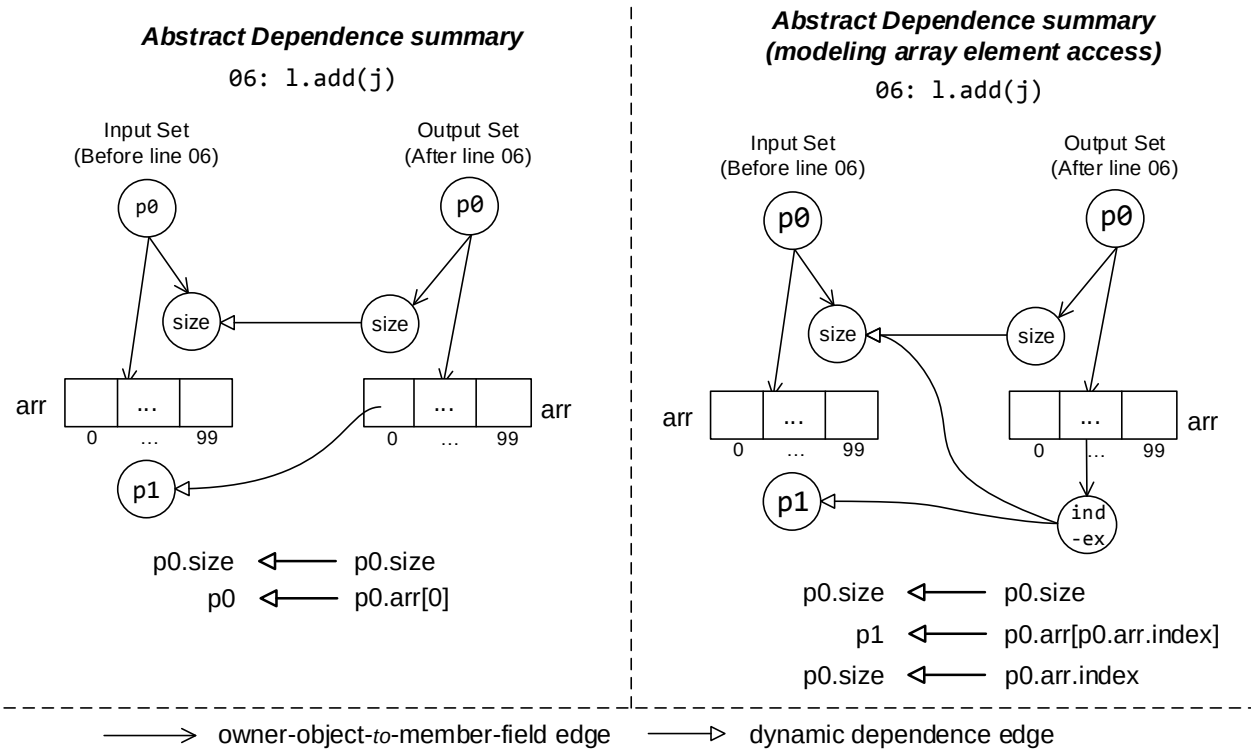


Figure 4.3: Abstract Dynamic Dependence Summaries, with and without modeling array element access, for the runtime invocation of `add` method at Line 06, as shown in the Example Program Execution in Figure 4.1.

no summary will contain its dependence information.

To address this problem, the approach in this work creates a special symbolic name for each array index, as shown in Figure 4.3(b). If the accessed array is an input or output of the method, the index used to access the array is considered as a (special) output (as shown in Figure 4.3(b)), and thus the transitive dependence relationships leading to the computation of the index would be included in the summary. If the index is a constant value (*i.e.,* its computation does not depend on any method input), this constant value is recorded in the summary. In our example, index t used in line 23 of Figure 4.1(a) is abstracted by a symbolic name $p_0.arr.index$, which is dependent on the symbolic location $p_0.size$, as portrayed in Figure 4.3(b). When the abstract summary is applied, or reused, for a future invocation of the same method, we will be able to obtain the run-time value of $p_0.size$ (before an invocation to `void add(int i)`), and identify the array element that is accessed during

the invocation.

4.3 Challenge 3: Accounting for Varying Method Behavior

A set of concrete summaries, for a given method (*e.g.,* `void add(int i)`) essentially represents the behavior of as many individual invocations of the method in question. As such, the abstraction of such a set of concrete summaries will result in an equal number of abstract dynamic dependence summaries — one abstract dynamic dependence summary for one specific invocation of the method in question. If all abstract dynamic dependence summaries, for the given method, model the same set of abstract inputs, abstract outputs, and the dynamic dependencies between those abstract inputs and outputs, then it is safe to say that the method exhibits similar external heap-data effects, or what we call method behavior, for each of its different invocations. In other words, if all invocations of a given method result in the same abstract dynamic dependence summary, then the behavior of any invocation of the method can be represented with that single abstract dynamic dependence summary.

The challenge arises when different invocations of a given method, exhibit different external heap-data effects, thus, resulting in different abstract summaries, *i.e.,* differing sets of abstract inputs, abstract outputs and dependence relations between such inputs and outputs. The challenge is to accurately model the behavior of any subsequent invocation of the given method, by selecting a set of abstract inputs, abstract outputs and dependencies between the inputs and outputs, from a divergent set of abstract summaries that represent varying heap-data effects of dynamically observed method-invocations. In other words, the resultant dynamic dependence summary for a method should be able to model the correct set of heap data effects (as dynamic dependencies between inputs and outputs) of any invocation of the

method, in general.

Dynamic dependence summaries should account for variations in method-invocations, if they are to be amenable for reuse, for any subsequent invocation of the given method. Variations in external heap-data effects for different invocations are observed under two broad circumstances that I discuss below.

4.3.1 Challenge 3.1: Accounting for Polymorphic Methods

Different method-invocations, for a given method, may accept objects of different types that share a common super-type, as arguments. Although these objects share a common super-type, their fields can differ significantly, or they might result in the execution of entirely different method implementations — like in the event of polymorphism — resulting in different external heap-data effects, and thus abstract summaries between different invocations.

The approach in this work overcomes this problem by additionally recording the dynamically-observed type information of each input argument with the symbolic name representing the parameter. This enables the accurate modeling of the variances in method behavior due to differing input argument types. Before a summary edge is made concrete (during the dependence analysis), the approach first checks whether the recorded type in the edge matches the type of its corresponding actual parameter in the current execution, and only apply those summary-edges that match.

4.3.2 Challenge 3.2: Accounting for Divergent Control Flow

A similar problem arises when different control-flow paths are followed across different invocations of the same method, even with the same type(s) of input argument(s). It is conceivable that the resulting external heap-data effects also vary from one invocation to another, for

such methods, thus resulting in different abstract summaries for the same method.

In this work, abstract summaries from different method invocations that share the common set of dynamically-observed argument-types, for a given method, are *aggregated* into a single over-arching summary that I refer to as an aggregated abstract dynamic dependence summary, or simply, an aggregated summary. An aggregated summary is created by simply performing set-union operations on the sets of abstract inputs, abstract outputs and the dependencies (between such inputs and outputs) from the different abstract summaries that are being aggregated. Such an aggregated summary serves as a generic model of the external heap-data effects for any of the method's invocations that share the same dynamically-observed argument types.

4.4 Challenge 4: Reusing Abstract Dependence Summaries

An abstract dynamic dependence summary represents a symbolic model of the heap-data effects that are a result of invoking a particular method in question. However, to be able to save subsequent costs of analyzing the effects of such methods dynamically, the abstract summary must be made concrete. In other words, the symbolic information in an abstract summary must be substituted with runtime or concrete information to enable the modeling of specific invocations of the given method, thus going beyond a generic model of the method's behavior. However, even such a concrete model of heap-data effects of a specific method-invocation, does not represent the dependencies between those instructions that influence or depend upon the method-invocation. To be able to use dynamic dependence summaries (concrete or abstract) towards modeling dependencies between actual runtime instructions, it is important to transform the heap-data effects, which are essentially dependencies be-

tween the inputs and outputs of a method invocation, into dependencies between runtime instructions that define and use a method-invocation's inputs and outputs, respectively. I refer to such a reuse of method's dynamic dependence summary as summary *application*.

Summary application is carried out in two steps. First, symbolic names in a method summary's input and output sets are substituted or *concretized* with their respective concrete (heap or stack) locations at a method-invocation. Such concrete information in the summary forms the actual (transitive or direct) dependence relationships between the method-invocation's actual inputs and outputs. Second, the dependencies between the inputs and outputs of a method invocation are then used to derive runtime dependencies between instructions that define or use those inputs and outputs. If a specific output (location) of a method-invocation is dependent on an input (location) to the method-invocation; then we determine that any runtime instruction that uses the value from the output, is dynamically dependent on any instruction that defines the value in input to the method invocation.

Example

To better illustrate the idea of deriving dependencies between runtime instructions from a method-invocation's dependence summary, consider Figure 4.4. Figure 4.4 depicts an execution trace for a sample program, both of which are presented earlier in Figures 4.1(a) and 4.1(b). Each line in the execution trace shows the execution of an individual program instruction that we call a runtime instruction or an instruction-execution event.[2] The progress of the execution is denoted by a downward-arrow on the right that depicts execution-time. In addition, Figure 4.4 also portrays snapshots of the execution's memory on the right-hand-side of the execution-trace, taken at four different moments during the progress of the execution.

[2] *N.B.,* a single program instruction can be executed multiple times, and can lead to multiple instruction-execution events; *e.g.,* program instruction `05: if (j < num)` results in two events: `05`^{1} `if (j < num)` and `05`^{2} `if (j < num)`.

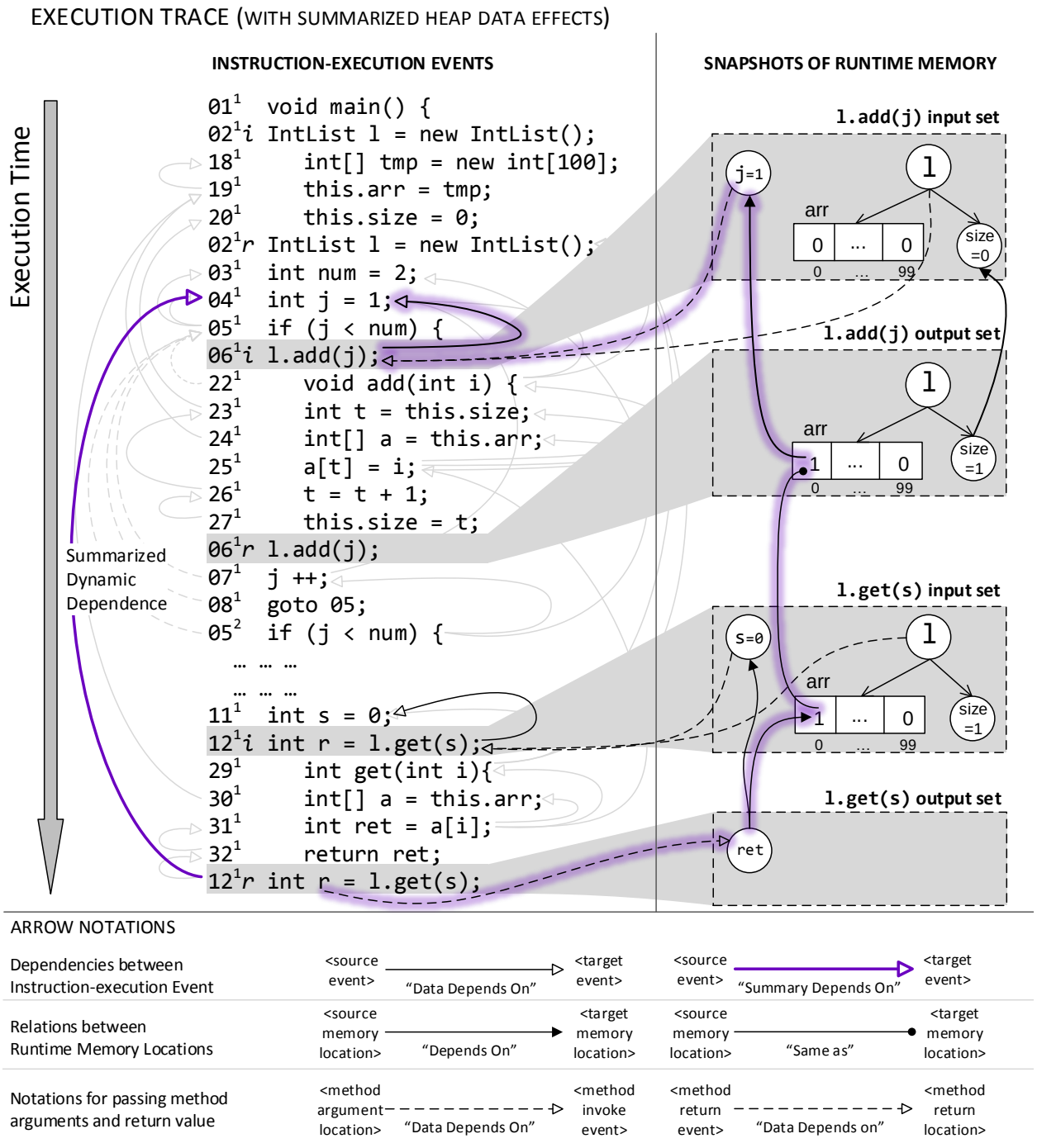


Figure 4.4: Example Program Execution Trace with summarized heap data effects *i.e.*, Dynamic Dependence Summaries.

These four snapshots of memory depict the inputs and outputs of the individual invocation of two methods: `l.add(j)` and `l.get(s)`. As such these snapshot show the inputs to the invocation events, *i.e.*, `06¹i l.add(j);` and `12¹i int r = l.get(s);`; and the outputs of the

invocations, just after their return events, *i.e.,* `06[1]r l.add(j);` and `12[1]r int r = l.get(s);`. For instance, the snapshot showing the inputs to the invocation event `06[1]i l.add(j);`, shows the input arguments `j` and (the receiver object) `l`. In addition, the memory locations accessible from `l`, *i.e.,* the field `size`, and the field array `arr` (along with `arr`'s elements), are also depicted with a simple arrow going from `l` to its members, *e.g.,* `l` $\rightarrow$ `size`. Aside, from showing the inputs and outputs to a method-invocation, the figure also shows how its outputs are dependent on the inputs with an arrow going from a location in the output set to a location in the input set. The arrow used to show such a dependency has a black-color-filled triangular arrowhead, *e.g.,* `arr[0]` —▶ `j`. The execution-trace and snapshots together also depict the method arguments' dependence on the invocation events and the dependence of the return event on a return value, one exists. Such relations between the invocation and return events and the actual memory values are shown simply as data dependencies, and are depicted with dashed arrows.

In Figure 4.4, consider the return event `12[1]r int r = l.get(s);` that is dependent on, or *uses*, the return value `ret`. The value `ret`, in turn, is part of the output set to the method-invocation event `12[1]i`. When we follow the chain of summarized dependencies between inputs and outputs, starting from `ret` that are highlighted in Figure 4.4, we obtain the following sequence of memory locations that ends with the input `j` within the input set for the method-invocation `06[1]i l.add(j);`:

`ret` —▶ `l`→`arr[0]` —▶ `j`

The input value `j` is in turn dependent on the invocation event `06[1]i l.add(j);`, as depicted in the figure. Upon tracing the dependency for the event `06[1]i`, with respect to the value `j`, we arrive at the runtime instruction `04[1]i int j = 1;` that actually defines the value `j`. In other words, the runtime instruction `04[1]i int j = 1;` defined the value of `j`, which was used to define the value at `l`→`arr[0]`, that was finally used by the runtime instruction `12[1]r`

as return value. Such a long chain of dependencies between the memory locations can be simply translated to a summarized dependency that the runtime instruction $\boxed{12^1 r}$ has on the runtime instruction $\boxed{04^1}$. Such a *summarized dynamic dependence* is depicted on the left-hand-side of the execution trace, with an arrow going from $\boxed{12^1 r \texttt{ int r = l.get(s);}}$ to $\boxed{04^1 \texttt{ int j = 1;}}$.

Chapter 5

Dynamic Dependence Summaries

With the discussion of the challenges involved in summarizing for dynamic dependence analysis as background, I present the formal concepts and algorithms that define dynamic dependence summaries in Section 5.1. In addition, I devote Section 5.2 to describe how such dynamic dependence summaries for method-invocations can be used for a summary-based dynamic dependence analysis.

5.1 Concepts and Algorithms

This section formally defines concrete and abstract dependence summaries, the aggregation of abstract summaries and finally the concepts behind the application (or reuse) of dynamic dependence summaries. These concepts as a whole present the core technique that computes and uses summaries to improve the efficiency of dynamic dependence analysis. The discussion in this section is in keeping with the conventions of object oriented programming, as available in the Java programing language.

Concrete Summaries. Informally, a concrete dynamic dependence summary, or simply,

a concrete summary, is a set of (transitive or direct) dynamically observed, data or control dependencies between the inputs and outputs of a specific invocation of a given method. Using the definitions for inputs and outputs for a method-invocation and dynamic data and control dependencies between memory locations, I formally define concrete summaries as follows.

Definition 5.1 (Concrete Dynamic Dependence Summary)**.** *For a method invocation event of the form* $c : m(a_0, a_1, \ldots, a_n)$*, the concrete dynamic dependence summary* $\mathcal{S}_c$ *is a cartesian set* $\mathcal{I}_c \times \mathcal{O}_c$*, where each element in the set is a transitive (or direct) dynamic dependence of the locations in the output set (*$\mathcal{O}_c$*), upon the locations in the input set (*$\mathcal{I}_c$*).*

Notationally, an access-path-based concrete dynamic dependence summary is expressed in the form: $\bigcup\{\ [\![o_i \dashrightarrow f]\!] \Leftarrow [\![o_j \dashrightarrow g]\!]\ \}$ where,

- $[\![o_i \dashrightarrow f]\!] \in \mathcal{I}_c$;
- $[\![o_j \dashrightarrow g]\!] \in \mathcal{O}_c$;
- o_i and o_j are the objects pointed to by parameters a_i and a_j;
- $[\![o_i \dashrightarrow f]\!]$ and $[\![o_j \dashrightarrow g]\!]$ are access-paths for heap locations f and g respectively.

Abstract Summaries. As described in Chapter 4 (Challenge 2), concrete summaries contain invocation-specific information and cannot be reused. Abstraction needs to be performed to replace concrete information with suitable abstract information, so that the abstracted summaries are applicable to all other executions. The abstraction process has two steps. In the first step, each node in an object graph that is part the concrete summary is expressed with the corresponding root object and the access-path through which the node can

be reached. The result of this step is a set of access-path-based concrete summary edges, as shown bottom part in Figure 4.2. In the second step, each concrete parameter object or variable is replaced with a symbolic name, resulting in the final abstract summary that can be applied in other executions of the method (as shown in the bottom parts of Figure 4.3). Note that in this approach, these two steps are combined in one single summary generation phase. They are discussed separately in the paper for the clarity of presentation.

Definition 5.2 (Abstract Dynamic Dependence Summary)**.** *Given a concrete dynamic dependence summary of the form* $\boxed{\bigcup\{\ [\![o_i \dashrightarrow f]\!] \Leftarrow [\![o_j \dashrightarrow g]\!]\ \}}$*, for a method-invocation of the form* $c : m(a_0, a_1, \ldots, a_n)$*, the abstract dynamic dependence summary is the symbolic representation of the method-invocation specific runtime-information in the concrete summary; and the resulting access-path based abstract dynamic dependence summary is of the form* $\boxed{\bigcup\{\ [\![p_i \dashrightarrow f]\!] \Leftarrow [\![p_j \dashrightarrow g]\!]\ \}}$ *where,*

p_i *and* p_j *are the symbolic names for the* i*-th and* j*-th memory locations* o_i *and* o_j *that are pointed to by parameters* a_i *and* a_j*, respectively.*

N.B., for concrete summary edges that model array element accesses as follows,

$[\![o_i \dashrightarrow f[l]]\!] \Leftarrow [\![o_j \dashrightarrow g]\!]$

$[\![o_k \dashrightarrow h]\!] \Leftarrow l$, where, l is an actual index value within an array;

the corresponding abstract summary edges are of the form,

$[\![p_i \dashrightarrow f[f.index]]\!] \Leftarrow [\![p_j \dashrightarrow g]\!]$

$[\![p_k \dashrightarrow h]\!] \Leftarrow f.index$, where $f.index$ is the symbolic name for the actual index value l and f is the array.

The algorithm for summary abstraction starts with computing a regular dependence graph (line 3) and the transitive dependence relationships for each node on it (line 4). Initially, the input set *is* contains all incoming objects (line 5) and the output set *os* contains only the returned objects (line 6). *sn* contains the symbolic name for each incoming and outgoing object. The worklist-based trace processing (lines 9–44) iteratively identifies and adds heap and stack locations into the input and output sets, and computes symbolic names for them (stored in set *sn*). After this processing is done, each transitive dependence edge, between statement executions, $s_1 \dashrightarrow s_2$ is retrieved from map *td*, (lines 45–52). If s_1 reads a variable/object in the input set and s_2 writes a variable/object in the output set (line 46), we find each symbolic name p_a for a and each symbolic name p_b for b (line 48), and add an abstract summary edge $p_a \dashrightarrow p_b$ into the abstract edge set *as*. Eventually sets *as* for all m's executions (in the training phase) are combined and used as m's abstract summary. Algorithm 1 shows the handling only for the two most complicated cases (i.e., array reads and array writes); the handling for all other cases is simpler and can be easily derived from the two cases shown.

Aggregate Dependence Summary.

Eventually, abstract summaries computed for all invocation events that invoke method m are combined, or aggregated, and used as m's summary for the future dependence analysis. This is done in the event where different method-invocations, for the given method, result in different or varying abstract summaries as discussed in Chapter 4 (Challenge 3). The aggregation of different abstract summaries performs two essential functions, as showcased in Algorithm 2. First, the aggregation step creates and maintains separate aggregate summaries for method-invocations with different dynamically-observed input argument types. Lines 6—14 in Algorithm 2 are used to create a signature for the method-invocation corresponding to a given abstract summary. The signature of a method-invocation is simply the string concatenation of the name of the invoked method and the types of the runtime arguments to

Algorithm 1 Abstract Summary Generation from an instruction-level execution trace for a method.

Require:
An execution trace t_m for method m
Objects $o_1, o_2, \ldots, o_n$ passed into m from the caller, and o_r returned from m

Set $as \leftarrow \emptyset$ // a set of abstract summary edges
Edge Set $td : \langle s_i \dashrightarrow s_j \rangle$ //transitive dependence relationships
Dependence graph $g = computeRegularDependenceGraph(t_m)$
$td = computeTransitiveClosureForEachNode(g)$
Set $is \leftarrow \{o_1, o_2, \ldots, o_n\}$ //input set
Set $os \leftarrow \{o_r\}$ //output set
Map $sn \leftarrow \{(o_1, \{p_0\}), (o_1, \{p_1\}), \ldots, (o_n, \{p_n\}), (o_r, \{p_r\})\}$ //initial symbolic names
List $wl \leftarrow \{o_1, o_2, \ldots, o_n, o_r\}$ //initial worklist
while $wl \neq \emptyset$ **do**
    Object $o \leftarrow wl.pop()$
    for each statement execution s in t_m **do**
        switch (s)
        case "a = b[i]":
            if $o_b = o$ **then**
                for each symbolic name p in $sn(o)$ **do**
                    String $p_i \leftarrow append(p, ".index")$
                    String $p_a \leftarrow append(p, ".[", p_i, "]")$
                    if $p_a \notin sn(o_a)$ **then**
                        $sn(o_a) \leftarrow sn(o_a) \cup \{p_a\}$
                        $sn(i) \leftarrow sn(i) \cup \{p_i\}$
                        $wl \leftarrow wl \cup \{o_a\}$
                    end if
                    if $o \in is$ **then**
                        $is \leftarrow is \cup \{o_a\}$
                    end if
                    $os \leftarrow os \cup \{o_a\} \cup \{i\}$
                end for
            end if
        case "a[i] = b":
            if $o_a = o$ **then**
                for each symbolic name p in $sn(o)$ **do**
                    String $p_i \leftarrow append(p, ".index")$
                    String $p_b \leftarrow append(p, ".[", p_i, "]")$
                    if $p_b \notin sn(o_b)$ **then**
                        $sn(o_b) \leftarrow sn(o_b) \cup \{p_b\}$
                        $sn(i) \leftarrow sn(i) \cup \{p_i\}$
                        $wl \leftarrow wl \cup \{o_b\}$
                    end if
                    $os \leftarrow os \cup \{o_b\} \cup \{i\}$
                end for
            end if
        end switch
    end for
end while
for each transitive dependence edge $s_1 \dashrightarrow s_2 \in td$ **do**
    if s_1 reads from a location a AND s_2 writes into a location b AND $o_a \in is$ AND $o_b \in os$ **then**
        // a and b can be both variables and field locations
        for each $p_a \in sn(a)$, each $p_b \in sn(b)$ **do**
            $as \leftarrow as \cup \{p_a \dashrightarrow p_b\}$
        end for
    end if
end for
return as

Algorithm 2 Aggregation of Abstract Summaries for Method-invocations.

Require:

1: Map *aggregate_summaries* $\leftarrow \emptyset$ // aggregate abstract summaries mapped by method-invocation argument types
2: Set *abstract_summaries* // a set of abstract summaries as input
3: **for** each abstract summary $\bigcup\{[\![p_i \dashrightarrow f]\!] \Leftarrow [\![p_j \dashrightarrow g]\!]\} \in$ *abstract_summaries* **do**
4: Object *summary* $\leftarrow \bigcup\{[\![p_i \dashrightarrow f]\!] \Leftarrow [\![p_j \dashrightarrow g]\!]\}$
5:
6: // a) get method-invocation information associated with the abstract summary.
7: String *inovked_method_name* $\leftarrow$ getMethodName(*summary*)
8: List *argument_types* $\leftarrow$ getMethodInvokeArgumentTypes(*summary*)
9:
10: // b) begin extraction of dynamically-observed method-invocation signature.
11: String *signature* $\leftarrow$ *inovked_method_name* // signature starts with invoked-method's name
12: **for** each argument type *type* $\in$ *argument_types* **do**
13: *signature* $\leftarrow$ *append*(*signature*, *type*)
14: **end for**
15:
16: // c) begin aggregation of abstract summary.
17: Object *aggr_sumr* $\leftarrow$ *aggregate_summaries*[*signature*] // aggregate summary for method invocation signature
18: **for** each access-path-based dependency $\{[\![p_i \dashrightarrow f]\!] \Leftarrow [\![p_j \dashrightarrow g]\!]\} \in$ *summary* **do**
19: *aggr_sumr.inputs* $\leftarrow$ *aggr_sumr.inputs* $\cup \{[\![p_i \dashrightarrow f]\!]\}$ // union of abstract inputs.
20: *aggr_sumr.outputs* $\leftarrow$ *aggr_sumr.outputs* $\cup \{[\![p_j \dashrightarrow g]\!]\}$ // union of abstract outputs.
21: *aggr_sumr.dependencies* $\leftarrow$ *aggr_sumr.dependencies* $\cup \{[\![p_i \dashrightarrow f]\!] \Leftarrow [\![p_j \dashrightarrow g]\!]\}$ // union of dependencies.
22: **end for**
23: *aggregate_summaries*[*signature*] $\leftarrow$ *aggr_sumr* // update aggregate summary for method invocation's signature
24: **end for**
25: return *aggregate_summaries*

the method-invocation. This enables proper modeling of method-behavior of polymorphic methods and their invocations as discussed in Chapter 4 (Challenge 3.1).

Second, for method-invocations of a given method that share the same dynamically-observed input arguments types, an *aggregate* summary is created that results in the generic modeling of the method's varying control flow as discussed in Chapter 4 (Challenge 3.2). Lines 16—23 in Algorithm 2 handle the aggregation of different abstract summaries that share the same input parameter types. Once the signature of the method invocation is computed, it is used retrieve an existing aggregate summary (Algorithm 2, Line 17). This is followed by Lines 18—21 in Algorithm 2 that a perform a union of all inputs, outputs and dependence-edges of the abstract summary with the existing aggregate summary, which is finally stored in Line 23. It is worth noting that a node (memory location) within an object-graph may have multiple access paths from the root object. For each such access path used in the method invocation, a corresponding summary edge is generated. Note that this treatment can potentially introduce both unsoundness and imprecision, and is further discussed in Chapter 1

Summary Application. Aggregated abstract dynamic dependence summaries are finally used to model the behavior of any subsequent invocations of their respective methods, thus enabling the reuse of recorded method behavior. I refer to such a process of reusing method behavior as the application of a dynamic dependence summary for a method invocation, or simply summary application. Summary application for a given method invocation is composed of two steps. First the symbolic information in the aggregated abstract summary is concretized. The dependencies within abstract dependence summaries are made concrete by substituting the abstract symbolic information that represent the method's arguments, in both the input and output sets, with concrete runtime objects. The concrete runtime objects for the substitutions in the input and output sets are collected just before and just after a method-invocation respectively. This allows us to re-create a specific concrete model

of the heap-data effects between the inputs and outputs of a specific method-invocation. To illustrate, consider the example of applying summaries at the invocation of the method `add` (line 06 in Figure 4.1 (a)) in a future execution of the program. As shown below the following three abstract summary edges are concretized into their respective concrete summary edges before being recorded into the trace –

1. Substituting p_0 with o^{02}; and p_1 with i;
2. Abstract Dependencies

   - $[\![p_0 \dashrightarrow size]\!] \Leftarrow [\![p_0 \dashrightarrow size]\!]$
   - $p_1 \Leftarrow [\![p_0 \dashrightarrow arr[arr.index]]\!]$
   - $[\![p_0 \dashrightarrow size]\!] \Leftarrow arr.index$

   Concrete Dependencies

   - $[\![o^{02} \dashrightarrow size]\!] \Leftarrow [\![o^{02} \dashrightarrow size]\!]$
   - $i \Leftarrow [\![o^{02} \dashrightarrow arr[arr.index]]\!]$
   - $[\![o^{02} \dashrightarrow size]\!] \Leftarrow arr.index$

Second, the dynamic dependencies between the concrete summary's inputs and outputs, for a given method-invocation, are translated to summarized dynamic dependencies between instruction execution events. It is important to remember that the reuse of dynamic dependence summaries is done with goal of efficient computation and modeling of dynamic dependencies between actual instruction-execution events. As such, I extend the dynamic dependencies between memory locations in the input and output sets of a method-invocation, to the resulting dynamic dependencies between the actual runtime instruction-execution events that define and use the memory locations in the input and output sets respectively. Such dependencies between runtime instruction-execution events derived from dynamic dependence summaries are called summarized dynamic dependencies and are defined as follows.

Definition 5.3 (Summarized Dynamic Dependence)**.** *Given two instruction execution events a^j and b^k, and a method-invocation of the form $c : m(a_0, a_1, \ldots, a_n)$, there exists a summarized dynamic dependence on event a^j for event b^k (represented as: $a^j \Leftarrow b^k$), under the following conditions:*

- *a^j writes a memory location l_1 that belongs to the input set of a method-invocation and b^k reads a memory location l_2 that belongs to the output set of the same method-invocation; and*
- *there exists a relationship $[\![p_i \dashrightarrow f]\!] \Leftarrow [\![p_j \dashrightarrow g]\!]$ in the abstract summary of the invoked method, such that,*

  – the location $[\![o_i \dashrightarrow f]\!]$ (before the call) is the same location as l_1;

  – and the location $[\![o_j \dashrightarrow g]\!]$ (after the call) is the same location as l_2;

  where o_i and o_j are the concrete run-time objects for the symbolic values p_i and p_j, respectively.

N.B., there also exists a summary dependence edge of the form $a_j \Leftarrow b_k$ if two pairs of relationships in the abstract summary that model an array element access,

- $[\![p_j \dashrightarrow g]\!] \Leftarrow [\![p_i \dashrightarrow f[f.index]]\!]$; $[\![p_k \dashrightarrow h]\!] \Leftarrow f.index$, and
- $[\![p_r \dashrightarrow v[v.index]]\!] \Leftarrow [\![p_m \dashrightarrow q]\!]$; $[\![p_t \dashrightarrow u]\!] \Leftarrow v.index$, such that

- $[\![o_m \dashrightarrow q]\!]$ is the same location as l_2,
- $[\![o_j \dashrightarrow g]\!]$ is the same location as l_1,
- $[\![o_i \dashrightarrow f]\!]$ and $[\![o_r \dashrightarrow v]\!]$ refer to the same array object, and
- values in $[\![o_k \dashrightarrow h]\!]$ and $[\![o_t \dashrightarrow u]\!]$, which represent indicies, are equal.

As discussed earlier in Section 2, two (ab-stract) array slots of the form $[\![p_1 \dashrightarrow f[f.index]]\!]$ and $v.index$ are considered to be the same location in an execution, if (1) the two concrete

array objects in locations $[\![o_i \dashrightarrow f]\!]$ and $[\![o_r \dashrightarrow v]\!]$ are the same, and (2) the values of the inputs that the two indices depend on (*i.e.,* $[\![o_k \dashrightarrow h]\!]$ and $[\![o_t \dashrightarrow u]\!]$ in the definition) are the same. A transitive edge is *not* added if one or both of the indices depend on multiple inputs, because it is unclear how the indices are computed from these inputs and how to compare their values.

5.2 Summary-Based Dynamic Dependence Analysis

As a part of the approach, I compute dynamic dependence information and dynamic dependence summaries by analyzing program executions. Dynamic information for program exections are generated and recorded in the form of execution traces during a summary generation phase, using a representative test exectuions. The dynamic dependencies and consequently the dynamic dependence summaries are computed from the execution traces and stored to a disk file for use in a future dynamic dependence analysis. The computation and use of the dynamic dependence summaries are performed in the following phases.

Phase I. Summary Generation using Representative Executions. I produce the abstract summaries for the methods being summarized (these can be user-specified in a configuration file, by method, class or package) by analyzing the execution traces. For each method m, I find all instances of m's execution, and for each instance in the trace, I use a worklist-based algorithm to compute an abstract summary according to Algorithm 1. The result of the summary generation is a mapping of inputs (formal method parameters or accessible fields) to outputs (formal method parameters or accessible fields) that they influenced, expressed as abstract summary edges, within aggregated dynamic dependence summaries.

Phase II. Summary Application in Dependence Analysis. To apply the summaries

to dynamic dependence analyses, the developer would choose to instrument her test case (*i.e.,* program execution) supported by the dynamic dependence summaries generated in the first phase. In the program's execution, the instrumenter would inspect each method-invocation to determine the existence of a corresponding dynamic dependence summary. If the summary is not provided for a method-invocation, the program execution and its instrumentation proceed with exhaustive dependence profiling. However, if a summary does exist for the method: (1) the abstract summary edges are obtained; (2) the runtime concrete inputs and outputs are matched with the symbolic names in the corresponding summary; and (3) the concretized summary edges are recorded to the trace.

Phase III. Dependence Graph Computation and Use. A summary-based dependence analysis profiles the execution of all methods except those that have abstract dependence summaries. These dependence summaries are then used to carry out dependence analysis by building a summary-based dynamic dependence graph. Before presenting the summary-based dynamic dependence graph, I first define regular dynamic dependence graph.

> **Definition 5.4** (Dynamic Dependence Graph)**.** *A dynamic dependence graph $(\mathcal{V}, \mathcal{E})$ has node set $\mathcal{V} \subseteq \mathcal{D} \times \mathcal{N}$, where each node is a static statement ($\in \mathcal{D}$) annotated with an integer i ($\in \mathcal{N}$), representing the i-th execution of this statement. An edge $e \in \mathcal{E}$ of the form $a^j \leftarrow b^k$ ($a, b \in \mathcal{D}$; $j, k \in \mathcal{N}$) denotes that the j-th execution of statement a writes a (heap or stack) location that is then used by the k-th execution of b, without an intervening write to that location.*

A dynamic dependence graph essentially represents, or models, a program execution; and is shown for an example program execution in Figure 4.1 (b). Based on the definitions of summary edges and dynamic dependence graph, I give the definition of summary-based dependence graph.

Definition 5.5 (Summary-Based Dynamic Dependence Graph)**.** *A summary-based dynamic data dependence graph $(\mathcal{V}, \mathcal{E} \cup \mathcal{T})$ is a regular dynamic data dependence graph augmented with an additional set of dependence edges $\mathcal{T}$ that denote summary dependence edges (refer Definition 5.3) between any two nodes a^j and b^k $(a^j, b^k \in \mathcal{V})$.*

A dynamic dependence graph is computed by recovering dependence relationships from the trace, as described in Definition 5.4. When (concretized) summary edges are encountered, the location matching approach described in Definition 5.3 is used to recover the missing relationships, to build a summarized dynamic dependence graph. Using this summarized dynamic dependence graph, developers and automated techniques can perform dynamic analyses, such as interrogative software debugging, bloat and change impact analysis and dynamic slicing, as discussed in Chapter 1.

Program Instrumentation. The implementation of the summary-based dependence analysis is based on the instrumentation and analysis of executable Java class files. The goal of the instrumentation is to enable the generation of a detailed trace that records the execution of each instruction in the program and the heap/stack location it accesses. I assign a unique ID to each run-time object that is used to identify the object and its fields in the execution trace. Program instrumentation involves the addition of probe instructions within the executable code for instructions that require runtime monitoring; thus enabling the requisite analysis of the executing instructions. With the execution trace, I construct the dynamic data dependence graph, which then enables client dynamic analysis techniques such as dynamic slicing. I perform load-time bytecode instrumentation for classes that do not belong to the Java standard library using ASM Bruneton *et al.* (2002), a Java bytecode manipulation framework. In contrast, I instrument the classes in the standard Java library prior to execution—several of these classes cannot be instrumented during load time due to the technical requirements of the JVM to load them prior to the instrumenter.

Chapter 6

Evaluation Overview

In order to devise an appropriate evaluation plan for this work it is important to revisit the thesis statement, as originally stated in Chapter 1:

> **Thesis:** Reusable dependence summaries for method invocations can reduce the computational costs involved in data- and control-flow based dynamic dependence analysis of software runs, while modeling such dynamic dependencies with moderate-to-high degrees of accuracy.

To support the above thesis, this evaluation will investigate the costs and accuracy of creating and using method dependence summaries along four directions:

1. the extent of performance gains in runtime-profiling for dependence analysis, when using method summaries;
2. the exent of accuracy losses when using statically and dynamically created method summaries to model heap-data effects of actual method invocations;
3. the extent of similarity between dynamic dependence summaries within and across

program executions; and,

4. the usage of dependence summaries for an actual software engineering technique that relies on dynamic dependence analysis.

Each subsequent section of this chapter will elaborate on the details of the different directions of investigation. The rest of this chapter frames each direction of investigation with specific research questions. Each of those research questions are answered in subsequent chapters of this dissertation. Additionally, the last section of this chapter will also elucidate the specific software subjects that will be used to perform the empirical investigation to answer the research questions.

6.1 Investigating Performance Gains

The usage of dependence summaries will afford the omission of exhaustive profiling of the method invocations being summarized. As such, the hypothesis behind this study is an obvious and clear reduction in the runtime profiling costs, when using dependence summaries, as against exhaustively profiling method invocations. This can be framed as the following research question.

RQ1 What are the gains in performance cost savings with usage of dependence summaries for dynamic analyses?

The usage of dependence summaries will bring down profiling costs in an obvious way. However, the motivation and goal behind this question is to qualitatively examine the extent of such reductions, which in turn will motivate the subsequent investigations into the accuracy of dependence summaries. Chapter 7 presents the experimental design and results for research question RQ1.

6.2 Investigating Accuracy Losses

The investigation into the extent of accuracy losses is carried out by considering multiple facets of summary creation and usage – each facet guided by its own research question.

Static vs. Dynamic Summaries. First, the investigation considers the two approaches used to create the dependence summaries: (a) statically and (b) dynamically. The static approach will model the dependencies by statically analyzing the binaries of the methods being summarized. The dynamic approach will involve analyzing exhaustive execution (data- and control-flow) profiles for representative invocations of the methods being summarized.

A primary hypothesis for this study is that both statically and dynamically modeled dependence summaries will introduce inaccuracies when compared to the actual heap-data effects of a specific method invocations. That said, this study anticipates that statically modeled dependence summaries will introduce inaccuracies to a greater degree than its dynamic counterpart. The study considers a dependence summary as inaccurate if it models dependencies between the method's inputs and outputs that are different from the actual "inputs/outputs" dependencies of specific method invocation, with greater differences implying higher inaccuracies.

In this study, all library methods for a software subject are targets for summarization, keeping with the original motivation of this work. As such, this study is framed with the following two research questions.

RQ2a How does the use of aggregated dynamic dependence summaries affect the accuracy of dynamic analysis for designated library methods?

RQ2b How does the use of static dependence summaries affect the accuracy of dynamic analysis for designated library methods?

RQ2c To what degree are static and aggregated dynamic dependence summaries similar, across different client programs for designated library methods?

Chapter 8 presents the experimental design and results all three research questions: RQ2a–c.

Suitability for Summarization. Next, the suitability for a method to be summarized is studied. All summarized library methods are studied for their accuracy in modeling dependence summaries, with respect to their positions in a calling hierarchy, specifically the call-graph.

Methods lower in a call-graph (*e.g.,* leaf methods that invoke no other methods) will be limited in their scope of operation, than their counterparts higher in the calling hierarchy. Thus, methods that are invoked lower in a calling hierarchy will afford the creation of accurate dependence summaries, in comparison to methods invoked higher in the calling hierarchy.

With such an intuition, I anticipate that the accuracy of method dependence summaries, created dynamically or statically, will increase with increasing depth of the methods in a statically determined call graph. In particular, the results of such a study may provide some explanation for the inaccuracies in method dependence summaries. Moreover, such a study may potentially guide in answering the question: "Is a given method suitable target for summarization?"

The resulting research questions are as follows.

RQ3a For methods at varying positions in a call-graph, how does the use of aggregated dynamic dependence summaries affect the accuracy of dynamic analysis?

RQ3b For methods at varying positions in a call-graph, how does the use of static dependence summaries affect the accuracy of dynamic analysis?

Section 8.3 details the experimental design and results that look to answer research questions RQ3a and RQ3b.

6.3 Investigating Similarity in Method Behavior

Unlike with static dependence summaries, dynamically creating dependence summaries requires the exhaustive profiling of method invocations that are representative of the method's behavior. And thus, the creation of dynamic dependence summaries itself warrants an investigation.

Specifically, it is important to understand how similar are the summaries across method invocations. When sampling method invocations, the goal is to create a generic dependence summary for the method that models the potential variety in the method's behavior. As such, understanding how to sample such method invocation, is to understand the similarity in the method's behavior: (a) during different invocations within a single test run; (b) across multiple test runs for a program; and (c) during invocations across subject programs.

Consider that a method exhibits similar summaries across its multiple invocations. Such method behavior similarity practically implies that it may be sufficient to profile a limited few invocations for a method, in order to build its dynamic dependence summary. As such, avoiding the profiling of all of the method's invocations. If however, the study reveals that a method exhibits varying behavior across its invocations, then building dynamic dependence summaries becomes a potentially expensive process due to the necessity of exhaustively profiling such method invocations.

This study anticipates that a majority of a method's behavior will be discovered by sampling a limited number of invocations during an individual software run. As a secondary hypothesis, this study expects greater similarity in method behavior across executions for a

single software subject, as against, across multiple software subjects. And thus, this line of investigation can be framed with the following research questions. The experimental designs and results for all three research questions are in Chapter 9.

RQ4a How does the behavior of a method vary with successive invocations in a given software execution?

RQ4b To what degree is the behavior of a method, as modeled by dynamic dependence summaries, similar across different executions of a client software subjects?

RQ4c To what degree is the behavior of a method, as modeled by dynamic dependence summaries, similar across different client software subjects?

6.4 Studying Dynamic Slicing with Dependence Summaries

This investigation will be carried out as a case study to qualitatively understand how an actual client analysis — dynamic slicing — is effected with the usage of dependence summaries. I anticipate that the results of this investigation to be in congruence with the outcomes of the preceding research questions. That said, the value of such a study lies in its ability to demonstrate the usage of summarizing dynamic dependence analysis in the context of a real world software analysis. Chapter 10 details a case study that uses dynamic dependence summaries for dynamic slicing, in an effort to answer the following research question.

RQ5 How does the use of dynamic dependence summaries affect the efficiency and effectiveness of a runtime client analysis?

6.5 Experimental Subjects

This evaluation employs eight client programs — NANOXML (> 2.6 KSLOCs), PL241 (> 6.9 KSLOCS), JTOPAS (> 10 KSLOCs), ANTLR (> 35 KSLOC), BLOAT (> 41 KSLOC), PMD (> 60 KSLOC), FOP (> 102 KSLOC), and JYTHON (> 245 KSLOC). The results of this empirical evaluation are based on investigating the summaries of all methods in the Java Standard library (*i.e.,* `rt.jar`) that were executed by the different runs of the eight client programs. Further, for each subject program, the evaluation considers the client code to be parts of the application that were not contained within the Java Standard library (`rt.jar`).

Multiple executions, with different test inputs, were monitored for the eight programs: 20 executions each for NANOXML, PL241, and JTOPAS; 5 executions each for ANTLR, BLOAT, PMD, FOP, and 3 executions for JYTHON. Monitoring the 83 executions, across the 8 subjects, involved the generation and storage (to disk) of the resulting execution traces. As such, the evaluation monitors fewer test executions for a client program with the increasing scale of the program's execution, in terms of the resulting trace-size and profiling time.

ANTLR, BLOAT, PMD, FOP, and JYTHON, along with their test inputs, were obtained from the DACAPO benchmarking suite by Blackburn & *et al.* (2006). NANOXML and JTOPAS and its test inputs, are obtained by the Subject-artifact Infrastructure Repository by Do *et al.* (2005). PL241 is an SSA-based optimizing compiler, and its development and test inputs are a product of a graduate-level course on advanced compiler construction at the University of California, Irvine.

The client programs were chosen such that they were real-world programs that carried out non-trivial computations, across system-wide test runs. It was important to select subjects with system-wide tests to enable the execution of significant portions of the subjects, thus, resulting in the execution of a wide range of methods from the Java Standard Library, through real world software systems.

Chapter 7

Investigation of Performance Gains

The investigation of performance gains due to a summary-based analysis provides the concrete motivation to use summaries while performing dynamic dependence analysis. The specific performance results presented in this chapter show the extent of cost savings for multiple test runs of complex, real world software subjects, and are driven with the following research question.

> **RQ1** What are the gains in performance cost savings with usage of dependence summaries for dynamic analyses?

As such, the rest of this chapter is devoted to describing the experimental setup and reporting the experimental results for answering the above research question.

Independent Variable. This experiment uses the following independent variable: the dynamic analysis used to detect data and control flows in program executions. The experiment performed dynamic analysis in two ways:

Treatment 1: Exhaustive Analysis. All components are instrumented, monitored and analyzed for data and control flow.

Treatment 2: Summary-based Analysis. Only the program under test is instrumented, monitored and analyzed for data and control flow, and reusable summaries are used for external components.

This experiment collected whole execution traces that recorded all executed instructions for all 83 executions across the eight client programs (NANOXML, PL241, JTOPAS, ANTLR, BLOAT, PMD, FOP, and JYTHON).

Such execution traces contain the data and control flows associated with each executed instruction, and are stored on disk. A single whole execution trace corresponds to a single run of any given subject program. The exhaustive analysis to produce such traces carries out the instrumentation, analysis and recording of instructions from all components.

As explained earlier in the final section of Chapter 6, this experiment treats the Java Standard Library (*rt.jar*) as the external library that is designated for summarization. As such the summary-based analysis, carries out such instrumentation and recording for all instructions, expect for instructions that are executed in the designated external library, *i.e.,* the Java Standard Library (*rt.jar*). In the summary-based analysis, reusable execution summaries are used to model the data- and control-flows at method invocation sites that invoke methods within *rt.jar*.

To specifically measure the impact of reused dependence summaries on the costs involved in performing dynamic dependency analysis, this experiment employed the following two metrics (presented as this experiment's dependent variables).

Dependent Variables. For the two forms of dynamic analysis, *i.e.,* exhaustive and summarized, and all 83 program executions, the experiment measured the following metrics:

Metric 1: Execution-Trace Size (S). Size of the execution trace due to each analysis.

Metric 2: Execution Time (T). Time to run the execution with each analysis.

Subject	T_O (sec)	Exhaustive T_E (sec)	Exhaustive $(T_E - T_O)/T_O$	Summary T_S (sec)	Summary $(T_S - T_O)/T_O$
NANOXML	.069	4.241	63.73	1.618	23.52
PL241	.191	8.445	39.69	2.780	13.01
JTOPAS	.095	23.143	252.68	21.185	231.22
ANTLR	.107	54.910	498.25	49.228	443.00
BLOAT	.418	188.065	454.15	39.270	97.42
PMD	.337	625.113	1766.20	356.852	1007.35
FOP	.384	202.628	509.40	88.158	244.39
JYTHON	.035	68.774	1963.97	34.268	978.09

Table 7.1: Median Runtime Overheads: T_O is median original running time of the program, T_E and T_S are median running times of whole execution analysis, for exhaustive and summary approaches. $(T_{E|S} - T_O)/T_O$ shows the median runtime overheads for each technique.

The experiment ran each subject program with their test inputs, under exhaustive and summary-based treatments, thus resulting in a distinct whole program trace and a summarized trace for each execution. The experiment collected performance statistics for each execution. The experiment measured the size of the resulting traces with a global size counter that kept track of the number of recorded runtime instructions. A timer that kept track of the elapsed time during a program's execution, enabled the collections of running times for all analyzed executions. (*N.B.,* the performance metrics were collected in the same manner for both summary and exhaustive execution analyses.)

Efficiency Experiment Results.

Tables 7.1 and 7.2 summarize the results of this experiment. For the test runs of a given client subject, Table 7.1 reports the median execution running times and the resulting median runtime overheads[1] due to the two forms of analysis. Each row in Table 7.1 reports the median running times for all executions of a single client subject. For each subject program, the table reports (a) the median running time of the original program's executions, *i.e.,*

[1]Runtime overhead is a measure of the slowdown of the original non-instrumented program, in terms of a multiplicative factor of the original time. The overhead is computed as $\frac{(\textit{running time with analysis}) - (\textit{original running time})}{(\textit{original running time})}$

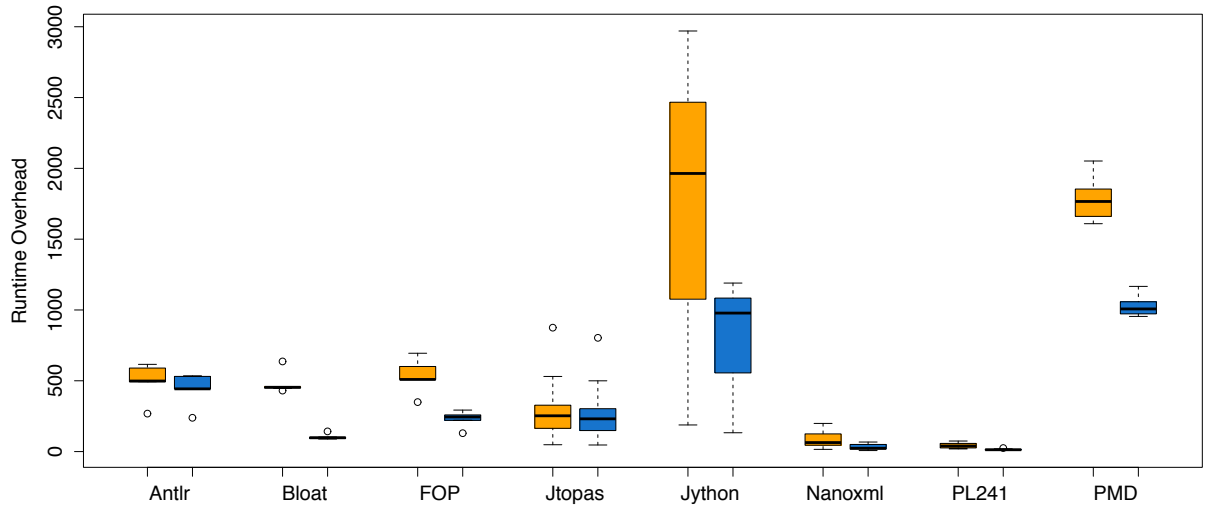


Figure 7.1: Runtime Overheads with the two analyses — Exhaustive (orange) vs. Summary (blue) — compared to original running times of the unanalyzed test runs.

without any analysis; (b) the median running time of the executions with exhaustive and summarized analyses; and (c) the resulting median runtime overheads due to the analyses, when compared with the original running times of the unanalyzed runs. For instance, the first row of the table, reports that the median running time for 20 executions of NANOXML is 69 milliseconds, without analysis. Executing the same test runs of NANOXML with exhaustive and summarized analyses, results in median running times of 4.241 seconds and 1.618 seconds, respectively. As such, for NANOXML the exhaustive and summarized analyses incur runtime overheads of 63.73× and 23.52×, respectively.

Just as Table 7.1 summarizes the median running times for the different subjects and analyses, Table 7.2 reports the median trace sizes and trace size reductions. For instance, the first row reports that analyzing NANOXML produced execution traces, with median trace sizes of more than $145{\times}10^3$ and $49{\times}10^3$ instruction instances as a result of exhaustive and summarized analyses, respectively. As such, the summarized analysis results in an median trace reduction of 67%.

Subject	**Exhaustive** S_E (# instr.)	**Summary** S_S (# instr.)	$(S_E - S_S)/S_E$
NANOXML	145,385.5	49,565	0.67
PL241	381,547	104,636	0.69
JTOPAS	1,350,910	1,264,239	0.06
ANTLR	1,745,946	1,550,560	0.12
BLOAT	4,832,656	1,114,822	0.77
PMD	17,765,531	10,702,644	0.40
FOP	4,843,109	2,287,778	0.53
JYTHON	2,431,598	1,204,888	0.50

Table 7.2: Median Trace Size Reductions: S_E and S_S are the median trace sizes for each technique. $(S_E - S_S)/S_E$ shows the median cost savings in trace sizes, with summaries.

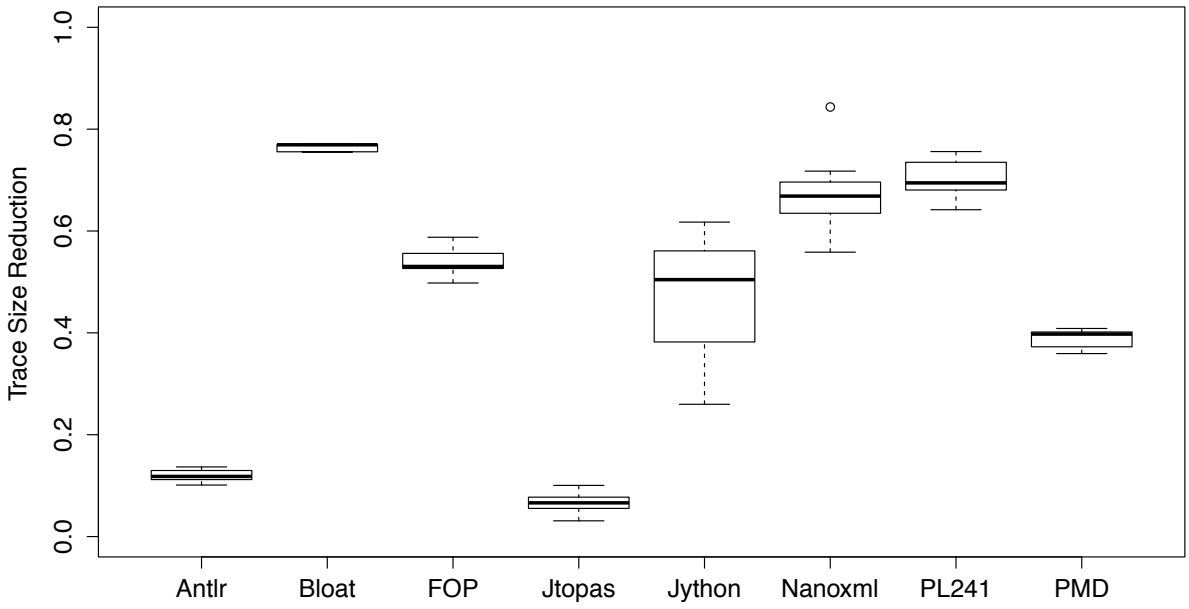
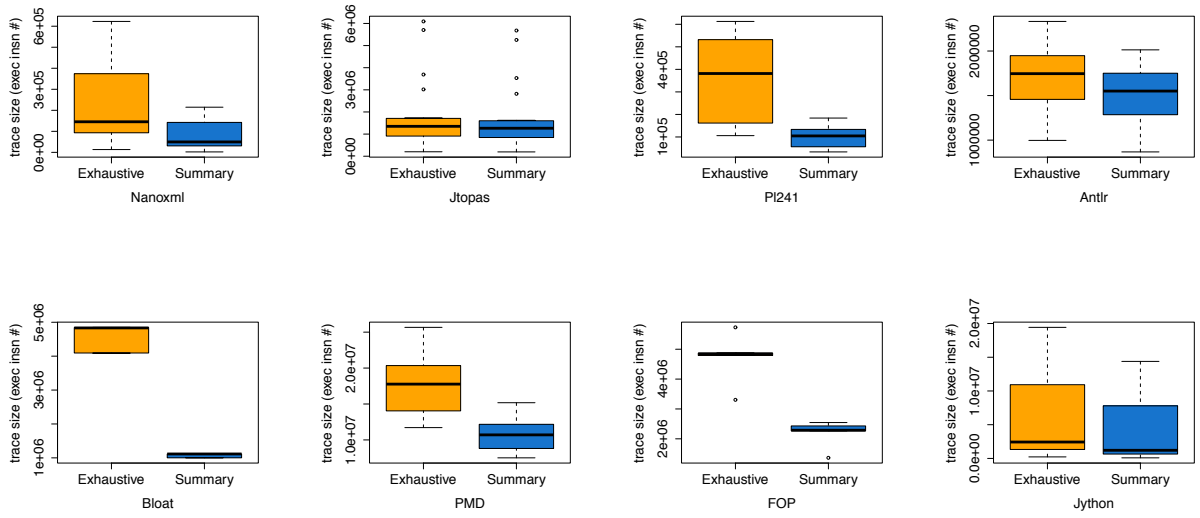
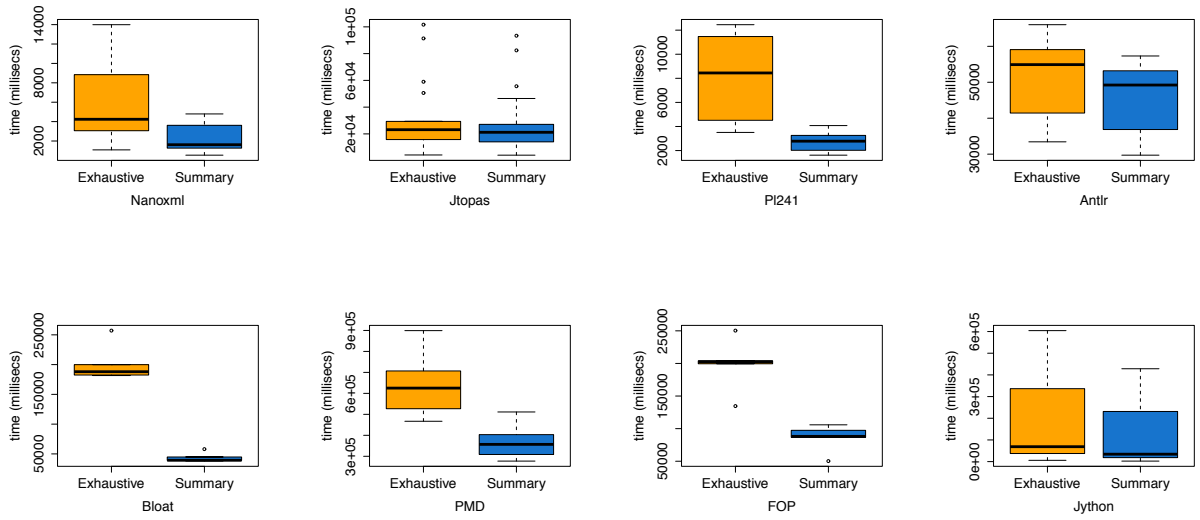


Figure 7.2: Trace Size Reductions as a result of Summary based analysis, over Exhaustive based analysis.

In addition to the median values, the actual runtime overheads and the trace size reductions, for all 83 executions across the eight subjects, are plotted in Figures 7.1 and 7.2. These figures, provide the distributions of runtime overheads and trace size reductions for all test runs of the individual subject programs; thus providing greater detail to the median values reported in Tables 7.1 and 7.2. For instance, Figure 7.1 shows that for subjects such as



(a) Trace Sizes (number of executed instructions) for multiple runs of client programs



(b) Analysis Running Times (milliseconds) for multiple runs of eight client programs

Figure 7.3: Running Times and Trace Sizes of the two analyses: Exhaustive (orange) vs. Summary (blue)

PMD, Bloat, and FOP the runtime overhead incurred by exhaustive analysis for all their executions are clearly more than the runtime overhead incurred due to summarized analysis. At the same time, for subjects such as Jtopas and Antlr both analyses incur similar overheads, with the exhaustive analysis resulting in the higher distribution of overheads.

Similarly, Figure 7.2 presents the distribution of trace size reductions for the multiple executions of the 8 client programs. Figure 7.2 shows that ANTLR and JTOPAS saw limited trace size reductions of less than 20%. Whereas, test runs of subjects like BLOAT, NANOXML and PL241 exhibit median trace size reductions of more than 65%. Such varying trace size reductions indicate the extent of a subject's reliance on methods in the Java Standard Library (*rt.jar*). Finally, parts (a) and (b) of Figure 7.3, further break down these performance numbers by presenting the actual values for the trace sizes and execution times for all executions, subjects and analyses.

These results show that, for the experimental subjects, performing exhaustive instrumentation incurred substantial runtime and space overheads. The highest reduction in median runtime overhead is observed in the case of BLOAT, where the median runtime overhead reduces from 454.15× (with exhaustive analysis) to 97.42× (with summarized analysis) — a reduction of 78.5%. Whereas, the least reduction in median runtime overhead is observed in the case of JTOPAS, where the median runtime overhead drops from 252.68× (with exhaustive analysis) to 231.22× (with summarized analysis) — a reduction of 8.5%. Additionally, the summary based approach reduced the median execution trace size by 61%, when examining the trace size reductions collectively for all 83 executions. When examining the median trace size reductions, for the executions grouped by their subject programs, we observe the largest reduction of 77% for BLOAT, and the least reduction of 6% in the case of JTOPAS.

For the experimental subjects in this work, the summary-based approach resulted in demonstrable reductions in performance costs for all 83 executions across the eight subjects. However, the summary based approach provided limited savings in the case of ANTLR and JTOPAS as noted earlier during the descriptions of Figures 7.1 and 7.2. That said, in absolute terms the modest savings in performance amounts for those specific subjects amounts to saving profiling costs of more than 100,000 instruction executions. Conversely, the usage of summaries while profiling subjects like PL241 and BLOAT provides clear and substantial

savings in profiling costs. Collectively, these results suggest that for programs that make extensive use of libraries, summarization of dependence summaries can save computational and storage costs during dynamic dependence analysis. Moreover, such cost savings were a result of summarizing only one designated library: Java Standard Library. Real world software applications often use several other external libraries, thus presenting further avenues for summarization and potential cost savings.

Chapter 8

Investigation of Accuracy Losses

Motivated by the savings found from the performance experiment in Chapter 7, this work investigated the extent to which these savings were realized at the expense of accuracy losses. As discussed in Chapter 1, both dynamically and statically generated dependence summaries are likely to model concrete dependence summaries for individual method invocations with some degree of inaccuracy. Inaccuracies are introduced in dynamic dependence summaries due to (a) reliance on dynamically collected data; and (b) aggregation of multiple abstract summaries into a single aggregate summary. On the one hand, different invocations of a method may exhibit possibly different external heap-data effects; thus, a reliance on dynamically collected data possibly may not render unseen dependence relationships for a future method-invocation. On the other hand, the aggregation of dynamic summaries, which attempts to model external effects of multiple and varying method-invocations, might result in spurious dependence relationships. Static dependence summaries are likely to introduce inaccuracies due to the potential for static analysis to conservatively model heap-data effects, essentially due to control flow structures. As such, this accuracy experiment focuses its investigation on the inaccuracies introduced as a result using aggregated dynamic dependence summaries and static dependence summaries.

The experiments presented in this chapter create concrete summaries for methods in the Java Standard Library (*rt.jar*) that are invoked in executions of NANOXML, JTOPAS, PL241, ANTLR, BLOAT, PMD, FOP, and JYTHON. By treating those concrete method summaries as ground truth, this investigation determined the accuracy of abstract aggregated dynamic summaries and static dependence summaries.

8.1 RQ2a: Investigating Accuracy Losses with Dynamic Dependence Summaries

In order to answer the following research question, this experiment constructed aggregate summaries by abstracting and aggregating concrete summaries of a method, which model other method invocations of the given method, within an individual execution.

> **RQ2a:** How does the use of aggregated dynamic dependence summaries affect the accuracy of dynamic analysis for designated library methods?

Experiment Independent Variable. To understand the effect of aggregation of summaries, the experiment created an aggregated abstract summary using only a sample of available concrete summaries for the given method. As such the experiment used the following as its independent variable:

Sample Size (SS). The number of concrete summaries for a single method, within a single program execution, selected for aggregation, as a percentage of the total concrete summaries for the given method within the single program execution. For a given method and program execution, this is computed as:

$$\mathbf{SS} = \frac{|\{\text{concrete summaries selected for aggregation}\}|}{|\{\text{all concrete summaries}\}|} \times 100$$

The experiment used the following four Sample Sizes: 1%, 10%, 25%, 50% and 100%. These percentages represent fractions of the concrete summaries for a method that are selected for creating an aggregated dynamic dependence summary.

After creating an aggregated dynamic summary using a sample of the concrete summaries, of a given Sample Size(%), the experiment compared remaining concrete summaries — not used for summary aggregation — with the aggregated dynamic summary. The experiment compared the aggregated summary with each remaining concrete summary in terms of the dependence relations between the method's inputs and outputs as captured in the two summaries. To enable an appropriate comparison with the aggregated abstract summary, the hold-out concrete summary was itself abstracted. This enabled an understanding of the extent of the inaccuracies, as a result of the over/under-approximations, within the aggregated dynamic dependence summary when it was used to model the dependencies exhibited by individual method invocations.

The usage of only a limited set of concrete summaries (*i.e.,* SS = {1%, 10%, 25%, 50%}) allowed the simulation of situations where the aggregated dynamic dependence summary is unaware of certain dynamic dependencies that are unique to a specific method-invocation that was not used for summary aggregation. In other words, such a modeling would possibly under-approximate the actual dependencies in a method-invocation.

Simultaneously, the experiment also created aggregated dynamic dependence summaries by using all available concrete summaries for a method within a given execution, *i.e.,* SS = 100%. The experiment compared such an aggregated dynamic summary with all concrete summaries that were aggregated to create the aggregate summary. Such a modeling permitted the simulation of situations where the aggregated dynamic summary modeled every known dependency across all invocations for a given method, within an execution; thus potentially leading to over-approximations with spurious dependencies.

For each sample size, the experiment obtained a random sample from the pool of concrete summaries, for a given method. For instance a sample size of 100% would imply that all available concrete summaries are used to create the aggregated summary. Similarly, a sample size of 50% would mean that half of all method summaries are selected, at random, to construct the aggregate summary. Note that for a method's summaries to be aggregated at a sample size of 50%, at least two concrete summaries must be present, with one available for aggregation and one concrete summary to compare with. If for a given method and sampling size, the minimum number of summaries are not available to select a random sample, then aggregation and comparison with the remaining concrete summaries were excluded for the study.

Experiment Dependent Variables. To assess the inaccuracies by the means of over/under-approximations within a dynamic dependence summary, the experiment treated the concrete summaries as the ground truth. Each concrete summary is then compared against the aggregated summary in terms of the dependence relations modeled by the two summaries. As such the experiment used the following metrics as the dependent variables.

Metric 1: Precision (P). The fraction of the dependence relations in the aggregate summary that are also present in the concrete summary. This is computed as:

$$\mathrm{P}_{AGGREGATE} = \frac{|\{\text{dependencies in summary}_{AGGREGATE}\} \cap \{\text{dependencies in summary}_{CONCRETE}\}|}{|\{\text{dependencies in summary}_{AGGREGATE}\}|}$$

Metric 2: Recall (R). The fraction of the dependence relations in the concrete summary that are also present in the aggregate summary. This is computed as:

$$\mathrm{R}_{AGGREGATE} = \frac{|\{\text{dependencies in summary}_{AGGREGATE}\} \cap \{\text{dependencies in summary}_{CONCRETE}\}|}{|\{\text{dependencies in summary}_{CONCRETE}\}|}$$

Ideally, both precision and recall for the aggregate summary, with respect to a concrete

summary, should be 1.0, suggesting a perfect modeling of the dependencies for the method invocation represented by the concrete summary. That said, a low precision score would indicate that the aggregate summary modeled dynamic dependencies that were not part of the concrete summary, (*i.e.,* over-approximation). At the same time, a low recall score would indicate that the aggregate summary does not model all dependencies that were a part of the concrete summary, resulting in an under-approximated modeling of the method invocation's dependencies.

Accuracy Experiment Results. The results for this experiment are summarized in Figures 8.1a to 8.1d as a set of box plots that show the distribution of precision and recall scores along the vertical axis, for randomly aggregated method summaries, using specific sampling sizes (*i.e.,* SS = {1%, 10%, 25%, 50%}).

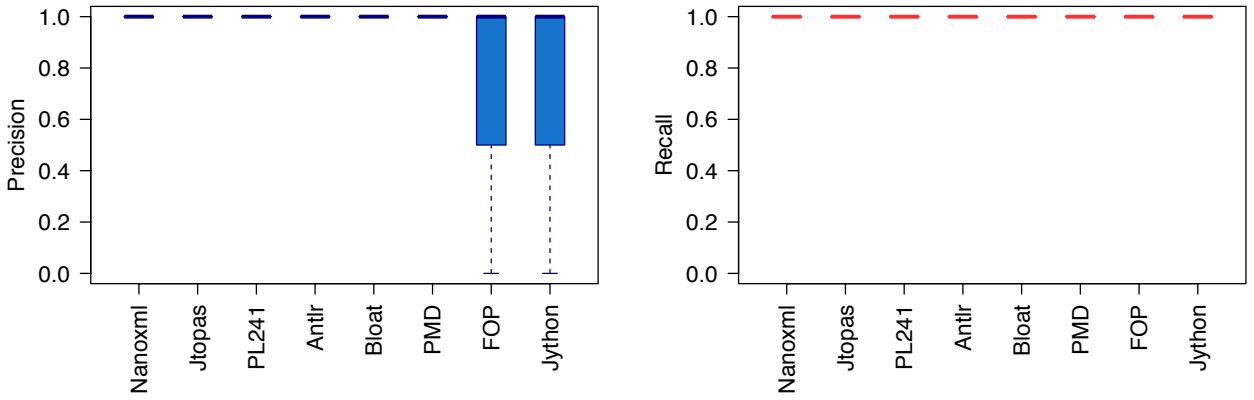
The figures present the precision and recall box plots, shown in blue and red respectively, for all client subjects, at a specific Sampling Size(%). For instance, Figure 8.1a presents the precision and recall scores for methods summarized across all executions, of all method, when only 1% of a method's concrete summaries were used to create the aggregate dynamic summary.

It is important to note that the summaries that are aggregated and compared to one another belong to the same execution. Such a grouping of the precision and recall distributions, *i.e.,* by sampling sizes, allows an understanding of the impact on accuracy at varying degrees of aggregation — from SS = 1% to SS = 50%, which imply limited aggregation. Additionally, Figure 8.2 shows the impact on accuracy (precious and recall) when concrete summaries were compared with an aggregate summary created after aggregating all concrete summaries, *i.e.,* SS = 100%.

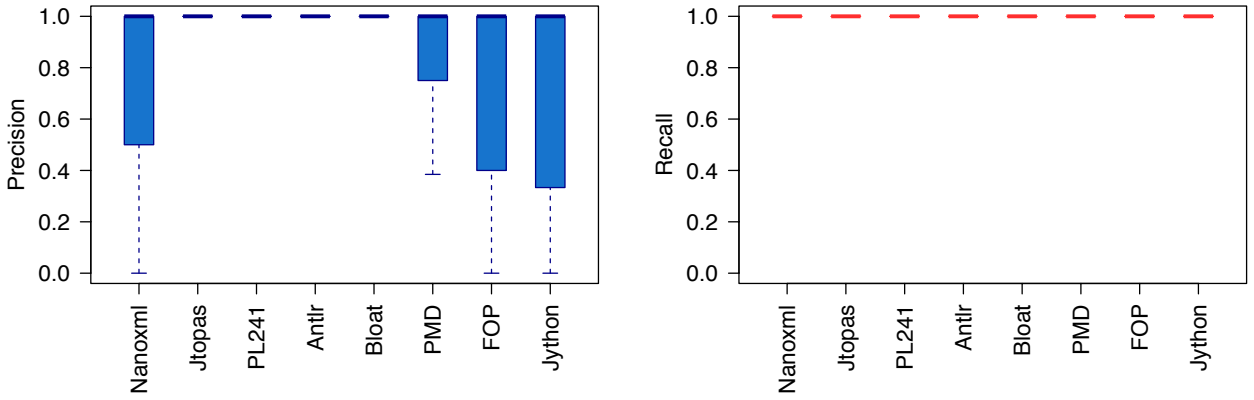
Recall Scores. The box plots in Figures 8.1(a)–(d) and Figure 8.2 depict high median scores of recall for randomly aggregated abstract summaries, across method executions and even

across different subject programs, with the median recall score of 1.0 for all subjects and sampling sizes. This suggests that all dependencies for nearly every given method-invocation in a single execution are being modeled. Note that a 1% sampling rate would require nearly all method-invocations for their respective methods to be similar in behavior, in order to attain a very high recall score. Such sound modeling could be attributed to a possible consistency in the behavior of methods in the Java library as used by NANOXML, JTOPAS, PL241, ANTLR, BLOAT, FOP, PMD, and JYTHON.

Precision Scores. All subjects, continue to maintain a high median value of 1.0 for precision scores, even for all of their randomly aggregated summaries with varying sampling sizes. However, unlike with recall distributions certain subjects, particularly NANOXML,

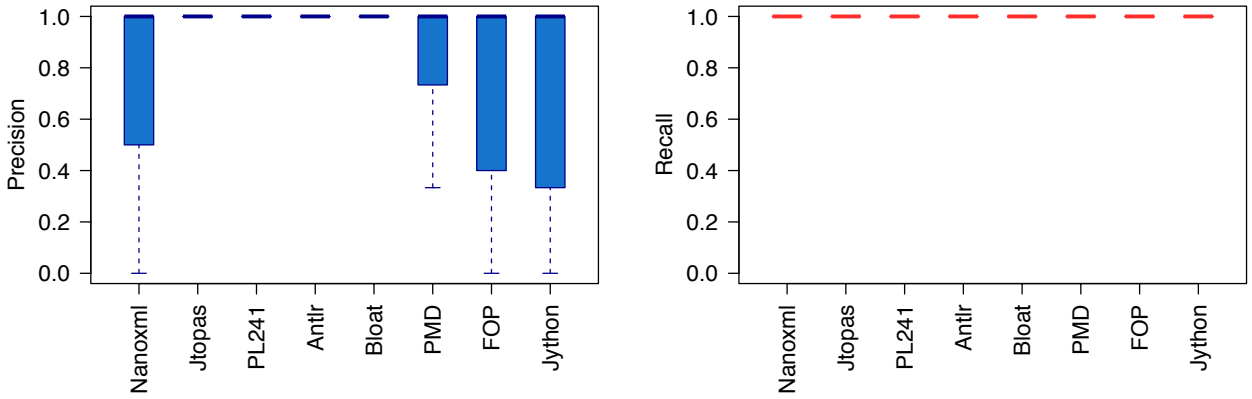


(a) Sample Size(%): 1%

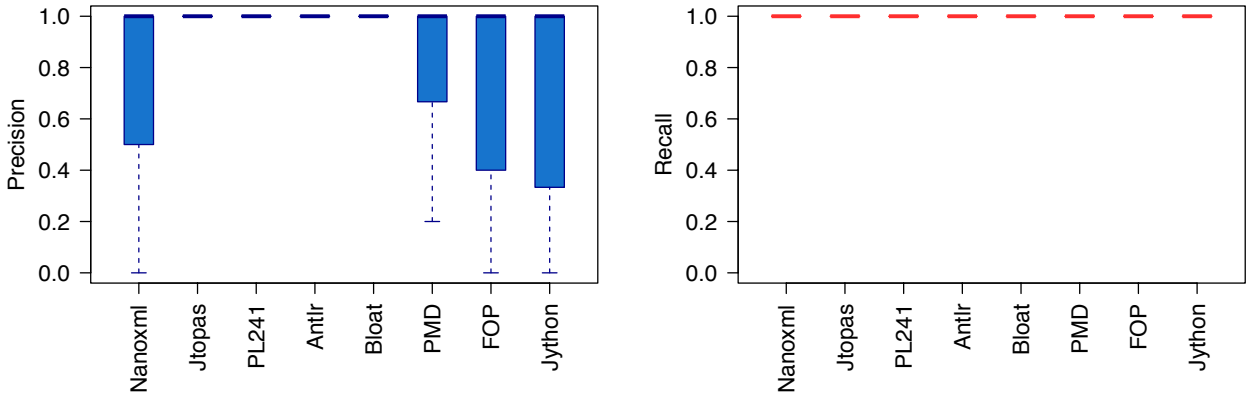


(b) Sample Size(%): 10%

Figure 8.1: Precision and Recall Scores of Aggregate Summaries at varying Sample Sizes(%)



(c) Sample Size(%): 25%



(d) Sample Size(%): 50%

Figure 8.1: (contd.) Precision and Recall Scores of Aggregate Summaries at varying Sample Sizes(%)

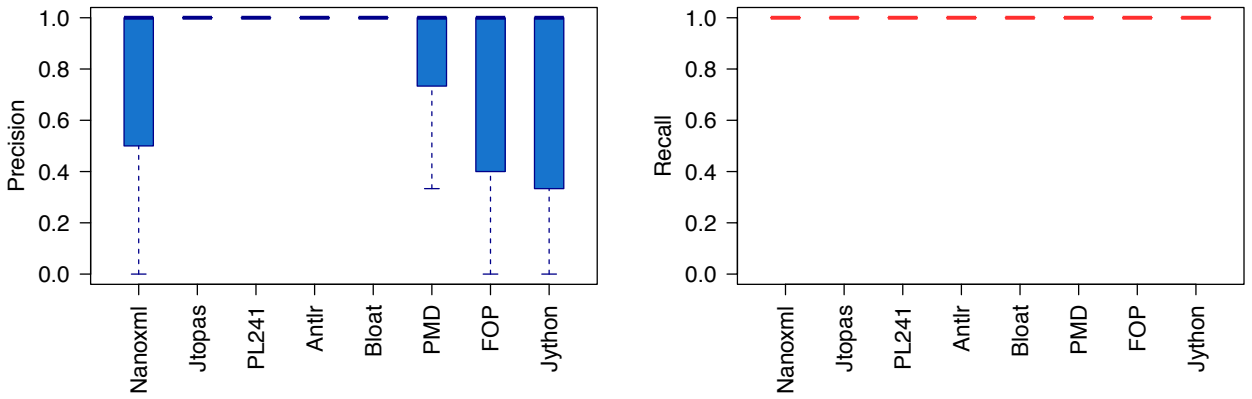


Figure 8.2: Precision and Recall Scores for Aggregate Summaries using all concrete summaries (Sample Size: 100%), for a given method within a single program execution.

PMD, FOP, and JYTHON, exhibit wider distributions of precision scores, suggesting an over-approximation in the modeling of method-invocations. Such distributions of precision

scores highlight the effect of spurious relations due to aggregation.

Taken as a whole, the plots for the precision and recall scores suggest that dynamic dependence summaries can be effective at accurately modeling the external heap-data effects for reused methods within all experimental subjects presented in this dissertation. Notably, the consistently high recall and precision scores are also observed for the varying sample sizes (1%, 10%, 25%, 50%, 100%). Such high scores in accuracy metrics could be attributed to a possible consistency in the behavior of methods in the Java library as used by the different subject programs. However, the results also indicate a less than perfect modeling of dynamic dependencies for every single method-invocation, as noted in the variance in the precision scores across all sample sizes. Such imprecisions might manifest as possible inaccuracies in downstream client analyses that rely on dynamic dependence summaries, which I investigate with a case study.

8.2 RQ2b,c: Investigating Accuracy Losses with Static Dependence Summaries

The accuracy results reported in Section 8.1 suggest that while aggregated dynamic dependence summaries model the dependencies exhibited by individual method invocations, they introduce additional, spurious dependencies that were not exhibited by the method invocations. Given such results and the presumption that statically generated summaries would also introduce spurious dependencies, it becomes important to investigate accuracy losses as a result of using static dependence summaries, and compare such losses with that of aggregated dynamic dependence summaries.

This section reports the results for two separate experiments on the investigation of accuracy losses. The first experiment compares statically generated dependence summaries with the

concrete dependence summaries for individual method invocations. Such an experiment is similar to that presented in the previous section (see Section 8.1), where instead of creating aggregated dynamic summaries, static dependence summaries are created by performing static analysis on the compiled binaries of the methods to be summarized.

The second experiment compares the aggregated dynamic summaries with the static dependence summaries. Such an experiment will highlight how similar dependence summaries are when created statically and dynamically, for a specific method. Moreover, such an experiment will reveal the number of additional dependence edges introduced in a method's summary strictly due to static analysis, over dynamic analysis.

The following subsections of this chapter will first provide an overview of statically creating method dependence summaries. The subsequent two sections will detail the experimental results that will look to answer the research questions (RQ2b and RQ2c).

Statically Generating Dependence Summaries: An Overview.

This experiment uses an inter-procedural static analyzer to identify dependencies between the inputs and outputs for a given method, based on the inter-procedural, finite, distributive, subset (IFDS) framework conceptualized by Sagiv *et al.* (1996). The IFDS framework models data and control flow analysis as a graph reachability problem. The experiment specifically uses the Soot Infoflow infrastructure[1] — a generic taint analyzer with support for Java Bytecode — to perform the static analysis. The Soot Infoflow infrastructure was originally used by Arzt *et al.* (2014) in their work on FlowDroid, and later by Arzt & Bodden (2016) in creating static method summaries for the android framework. This implementation is based on the Soot static analysis framework[2] as first described by Vallée-Rai *et al.* (1999).

The static analyzer computes static dependence summaries in four stages: (1) creating the

[1]Soot Infoflow: https://github.com/secure-software-engineering/soot-infoflow
[2]Soot: https://github.com/Sable/soot

call graph; (2) creating an inter-procedural control flow graph; ((3) creating an exploded supergraph as defined by the IFDS framework that models the data/control flows through the control flow graph; and (4) tracing the flows from the inputs to the outputs of a specific method in the exploded supergraph, thus creating a set of dependence edges between inputs and outputs of a given method.

The static analyzer creates the call graph using a context-sensitive, flow-insensitive points-to analysis, based on geometric encoding introduced by Xiao & Zhang (2011). Geometric encoding is able to efficiently encode and compress points-to and pointer assignment relations as geometric figures; thus, enabling a full context sensitive model for points-to analysis. The analyzer uses an implementation of the geometric-encoded, points-to analysis as provided in the Soot static analysis framework. Further, this implementation of points-to analysis uses a 1-CFA model for handling strongly connected components, as in the case of recursive calls. Such a context-sensitive points-to analysis is particularly important to accurately resolve a method invocation site to a method body, in the presence of polymorphic implementations; and thus prune spurious edges in a statically generated call-graph. Such a statically generated call graph enables the creation of an inter-procedural control flow graph. Each node in the control flow graph is a compiled instruction modeled using the Jimple three-address representation (provided in Soot) for Java Bytecode instructions.

In order to analyze a specific method, the analysis creates an artificial invocation site for the given method. Such an artificial invocation site then serves as the entry point for any subsequent analysis. Additionally, different sets of dynamically-observed argument types that are used to invoke the method, are used to create different artificial invocation sites. As such, separate static summaries are created for a method, for each set of dynamically-observed argument types with which the method was invoked.

It is important to note that the when analyzing a method, the static analyzer does not assume any aliasing relationship between the method's arguments, or their fields in the case

of reference types. As such, the resulting summaries may potentially be incomplete with missing dependence edges. As noted later in Chapter 11, empirical results suggest that such cases occur infrequently. Moreover, accounting for such aliasing relations between the method's arguments, in an effort to yield sound results from static analysis is a non-trivial problem as demonstrated by Chatterjee *et al.* (1999). Given that static analysis is an instrument of empirical evaluations, and is not a central contribution of this work, such a sound approach to static analysis is considered beyond the scope of this work.

The static analyzer models data- and control-flow using flow functions as prescribed by the IFDS framework. The analyzer specifically uses an implementation of the IFDS framework provided in the Soot Infoflow infrastructure. The analysis computes flow functions for instructions in the control flow graph. The flow functions propagate data flow facts through the instructions in the inter-procedural control flow graph. The propagation of flow-facts is used to model the data and control-flow between the connected instructions of the control flow graph. Moreover, the flow-facts are modeled as access-paths by the static analyzer, thereby enabling a comparison with the access-paths used to create dynamic dependence summaries in this work.

For instance, given an instruction of the form: `x.f = y`, the analysis will map the incoming data flow fact, {`y`} to the outgoing fact set {`x.f, y`}; thus implying that the value in variable `y`, flows into the field `x.f`. The flow facts {`x.f, y`} would then be available as incoming facts to the next instruction(s) in the control flow graph.

As a generic taint analyzer, Soot Infoflow models the necessary flow functions to perform the adequate data and control flow analysis, necessary for dependence analysis, with the exception of assignments to field references. For instance, given a field assignment instruction of the form `x.f = y`, Soot Infoflow originally maps the data flow fact {`y`} into {`x`}, instead of {`x.f`}. While sufficient for taint-analysis, such flows would be inadequate for an implementation of dependence analysis. As such, this work makes the necessary modifications to

the Soot Infoflow flow functions to enable the proper flow of data into the dereferenced field (`x.f`), instead of the field's parent (`x`).

Computing flow functions for instructions in the inter-procedural control flow graph of a given method, produces an exploded supergraph that encodes the individual data/control flows between the instructions in the exploded supergraph. The static analyzer finally computes dependencies between a method's inputs and outputs (modeled as access-paths) by tracing the flows in the exploded supergraph starting from the inputs at method's artificial invocation site, to the outputs at each return instruction within the method.

The resulting static dependence summary for a single method is then compared with the concrete and aggregated dynamic dependence summaries of the method to answer research questions RQ2b and RQ2c, respectively.

Comparing Static and Concrete Dependence Summaries.

The first experiment in this investigation — comparing static dependence summaries with concrete summaries for individual method invocations — is framed with the following research question.

> **RQ2b:** How does the use of static dependence summaries affect the accuracy of dynamic analysis for designated library methods?

This experiment used the following metrics of precision and recall, similar to that used in Section 8.1, to measure the extent of accuracy losses when using static dependence summaries.

Metric 1: Precision (P). The fraction of the dependence relations in the static summary that are also present in the concrete summary. This is computed as:

$$\mathrm{P}_{STATIC} = \frac{|\{\text{dependencies in summary}_{STATIC}\} \cap \{\text{dependencies in summary}_{CONCRETE}\}|}{|\{\text{dependencies in summary}_{STATIC}\}|}$$

Metric 2: Recall (R). The fraction of the dependence relations in the concrete summary that are also present in the static summary. This is computed as:

$$\mathrm{R}_{STATIC} = \frac{|\{\text{dependencies in summary}_{STATIC}\} \cap \{\text{dependencies in summary}_{CONCRETE}\}|}{|\{\text{dependencies in summary}_{CONCRETE}\}|}$$

Experiment Results. The experiment compared the statically generated summaries with the concrete summaries associated with specific method invocations across all executions of all experimental client programs of this work. The resulting precision and recall scores are presented in Figure 8.3 and Table 8.1. Figure 8.3 presents the distribution of precision and recall scores grouped by each client program, with a series of boxplots. The distribution in Figure 8.3 indicate consistently high recall scores and precision scores with a high degree of variance. Such results support a well understood expectation that static analysis is conservative an as such introduces additional, spurious dependence edges that were not exhibited by individual method invocations.

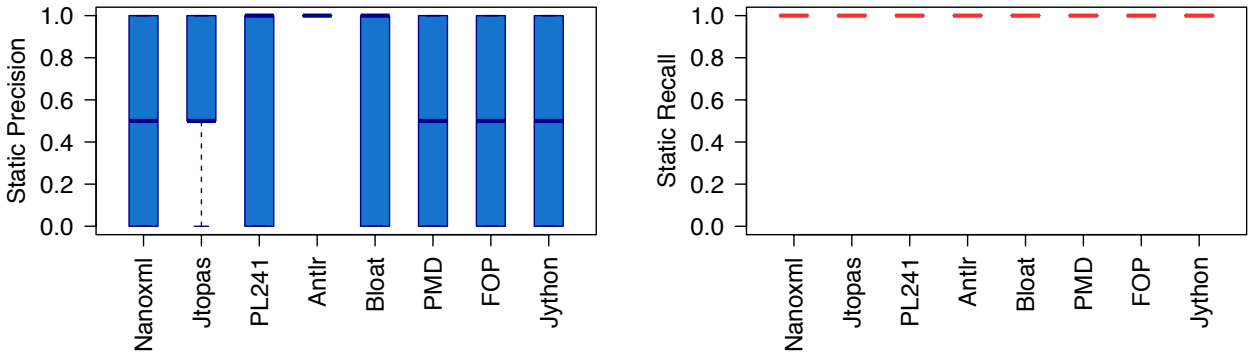


Figure 8.3: Precision and Recall scores when comparing Static Dependence Summaries with Concrete Dependence Summaries

Given the high degree of variance in the precision scores, Table 8.1 further presents the summary statistics for the precision scores across the eight client programs. For instance,

Precision	NANOXML	JTOPAS	PL241	ANTLR	BLOAT	PMD	FOP	JYTHON
Minimum	**0.0**	**0.0**	**0.0**	1.0	**0.0**	**0.0**	**0.0**	**0.0**
1^{st} Quart.	**0.0**	**0.5**	**0.0**	1.0	**0.0**	**0.0**	**0.0**	**0.0**
Median	**0.5**	**0.5**	1.0	1.0	1.0	**0.5**	**0.5**	**0.5**
3^{rd} Quart.	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Maximum	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 8.1: Precision of Static Dependence Summaries, when comparing with Concrete Dependence Summaries

the 3^{rd} Quartile statistic in Table 8.1 suggests that at least, static generation of summaries was able to correctly model concrete dependencies in about 25% of the cases for all eight subjects. However, static summaries do correctly model about 50% of method invocations in the case of BLOAT and PL241, and nearly all method invocations in the case of ANTLR. Such variance in the precision scores, along with the consistently high recall scores for static dependence summaries suggest that static analysis is likely to introduce spurious dependence edges, when modeling concrete dependencies for method invocations.

Comparing Static and Aggregated Dynamic Dependence Summaries.

In comparison with the distribution of precision scores for aggregated dynamic dependence summaries (see Figure 8.2), static summaries do exhibit a wider range, and often lower levels of precision scores. This leads the investigation to compare the static and aggregated dynamic summaries directly. Such an investigation is guided by the following research question.

> **RQ2c:** To what degree are static and aggregated dynamic dependence summaries similar, across different client programs for designated library methods?

In order to answer RQ2c, this experiment compares static and aggregated dynamic summaries and quantifies such comparison using the following metrics.

Metric 1: Similarity. The similarity between a static and aggregate summary is computed using the Jaccard similarity coefficient, as follows:

$$\mathrm{J}(\mathrm{summary}_{STATIC}, \mathrm{summary}_{AGGREGATE}) =$$
$$\frac{|\{\text{dependencies in summary}_{STATIC}\} \cap \{\text{dependencies in summary}_{AGGREGATE}\}|}{|\{\text{dependencies in summary}_{STATIC}\} \cup \{\text{dependencies in summary}_{AGGREGATE}\}|}$$

Metric 2: Extra Static Edges. The amount of dependence edges that are uniquely in a static dependence summary is computed as follows:

$$\mathrm{E}_{STATIC} =$$
$$\frac{|\{\text{dependencies in summary}_{STATIC}\} - \{\text{dependencies in summary}_{AGGREGATE}\}|}{|\{\text{dependencies in summary}_{STATIC}\} \cup \{\text{dependencies in summary}_{AGGREGATE}\}|}$$

Experiment Results. The experiment created the aggregate dependence summaries for the library methods invoked in each client subject separately. Specifically, the experiment aggregated all concrete dependence summaries of a method invoked within individual executions of each client subject. Such aggregated summaries were then compared with the statically generated dependence summaries for the respective methods.

The subsequent results of comparing static and aggregated dependence summaries, across the eight client programs are presented in Figure 8.4. Figure 8.4 presents two sets of boxplots. The upper set of boxplots show the distribution of the extra edges in a static summary, when compared with a aggregated dynamic summary. The lower set of boxplots show the distribution of similarity scores when comparing static and aggregated dynamic summaries across all client subject programs. The distributions for similarity and static edges show high degrees of variance. As such, the summary statistics for those distributions are further reported in Tables 8.2 and 8.3.

	NANOXML	JTOPAS	PL241	ANTLR	BLOAT	PMD	FOP	JYTHON
Min.	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
Median	1.0	1.0	**0.5**	1.0	**0.5**	1.0	1.0	**0.5**
Max.	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 8.2: **Similarity**: Comparing Static vs. Dynamic Dependence Summaries

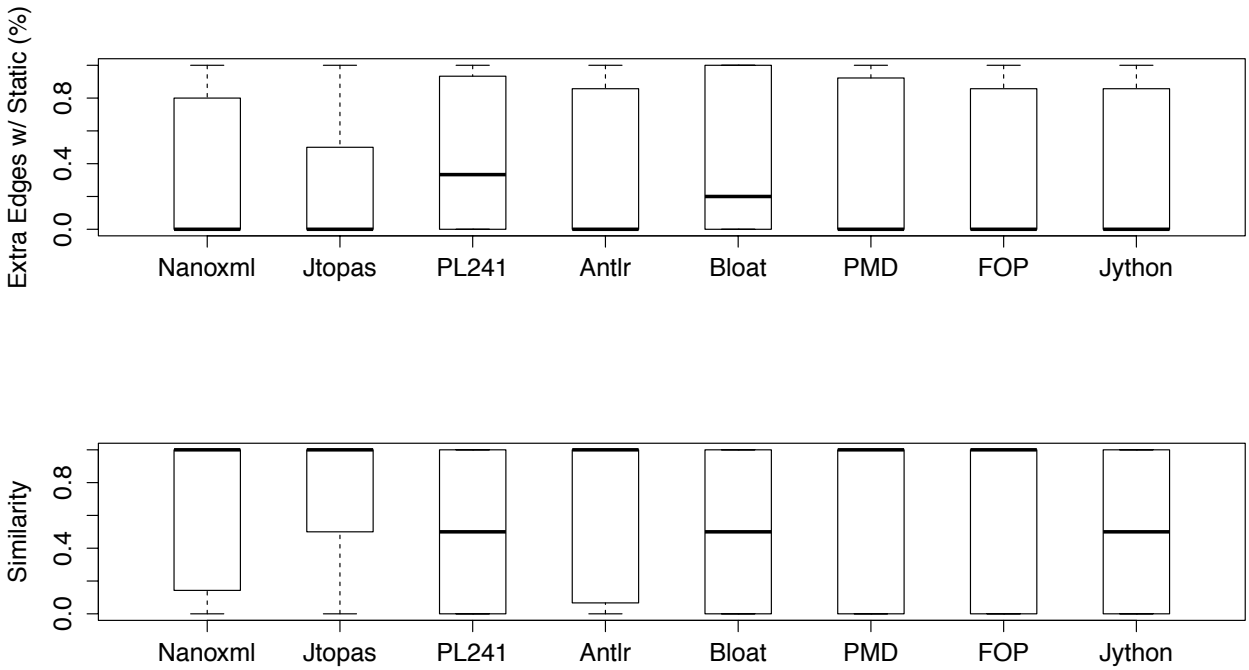


Figure 8.4: Static Dependence Summaries vs. Dynamic Dependence Summaries

	NANOXML	JTOPAS	PL241	ANTLR	BLOAT	PMD	FOP	JYTHON
Min.	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
Median	**0.0**	**0.0**	**0.33**	**0.0**	**0.2**	**0.0**	**0.0**	**0.0**
Max.	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 8.3: **Extra Static Edges**: Comparing Static vs. Dynamic Dependence Summaries

These results collectively indicate a perceptible degree of similarity between dependence summaries when they are created statically and dynamically. The high median similarity scores of 1.0 for five out of eight subject (*i.e.,* NANOXML, JTOPAS, ANTLR, PMD, and FOP) suggests that summaries for about 50% of the methods invoked in the five client subjects could have been created correctly with static analysis. However, the variance in the similarity distributions in all subjects and the modest median scores in subjects such as PL241, BLOAT and JYTHON suggest the introduction of spurious dependence edges as a result of static analysis, for the methods invoked across the different client subjects.

In addition to the majority of the similarity results reported at 1.0, one may notice that the statistics reported in Tables 8.2 and 8.3 are remarkably regular, with common values such as

0.0 and 0.5. We get such results because a vast proportion of methods typically have limited number of formal method arguments, leading to a limited number of summarized dependence edges between the method inputs and outputs. As such, the limited set of dependence edges in a vast majority of methods are modeled similarly by either static or dynamic analysis, with differences emerging in the form of a few additional edges due to the conservative modeling of control flow in static analysis. Indeed, even in cases where static and dynamic analysis model different dependence edges (similarity score 0.0), the number of actual dependence edges are limited.

8.3 RQ3: Investigating Accuracy Losses at varying call graph positions

Sections 8.1 and 8.2 reported that both aggregated dynamic and static dependence summaries exhibit a clear degree of variance in their precision scores when such summaries were compared with concrete dependencies. Such variance, and specifically a drop in the precision scores demonstrates the presence of spurious edges in static and aggregate dependence summaries.

Such variance, and modest precision scores can potentially be explained by the scope of a method's execution. The heap-data effects of a method's invocation are likely to impact the heap-data effects exhibited by the method's caller. As such I speculate: methods that invoke no other method, known as leaf methods in a calling hierarchy, tend to be limited in scope and are less likely to exhibit varying behavior across its invocations. Conversely, the behavior of methods that potentially invoke (directly or transitively) a large number of other methods are more likely to demonstrate varying summarized dependencies between the inputs and outputs of its invocations.

Given such insights, this chapter's final section investigates any correlation between the precision of a method's (aggregate/static) dependence summary, when compared to concrete dependencies, and the number of other methods that a method can potentially invoke. Such results may inform on the decisions on when to summarize. As such, this chapter reports the experimental variables and results for answering the following two research questions.

> **RQ3a:** For methods at varying positions in a call-graph, how does the use of aggregated dynamic dependence summaries affect the accuracy of dynamic analysis?
> **RQ3b:** For methods at varying positions in a call-graph, how does the use of static dependence summaries affect the accuracy of dynamic analysis?

Experiment Variables.

Independent Variable. This experiment quantifies the position of a method in a calling hierarchy by the size of the static call graph that is rooted in the given method. Each node in the graph is a method that is potentially invoked, directly or transitively, by the method in question. The static call graph is computed using the same context-sensitive points-to analysis that is used in computing the static dependence summaries (see Section 8.2). The size of the static call graph will serve as the independent variable in answering both research questions RQ3a and RQ3b, and is defined as follows:

Static Call Graph Size (method#). The number of methods in the static call graph that is rooted at the given method.

Dependent Variable. The experiment uses the following precision metrics as the dependent variables. Specifically, the precision metric for aggregate dependence summaries and static dependence summaries, of a given method, will serve as the dependent variables in answering research questions RQ3a and RQ3b, respectively.

Precision ($\mathbf{P}_{AGGREGATE}$). The fraction of the dependence relations in the aggregate dynamic summary that are also present in the concrete summary.

$$\mathrm{P}_{AGGREGATE} = \frac{|\{\text{dependencies in summary}_{AGGREGATE}\} \cap \{\text{dependencies in summary}_{CONCRETE}\}|}{|\{\text{dependencies in summary}_{AGGREGATE}\}|}$$

Precision ($\mathbf{P}_{STATIC}$). The fraction of the dependence relations in the static summary that are also present in the concrete summary.

$$\mathrm{P}_{STATIC} = \frac{|\{\text{dependencies in summary}_{STATIC}\} \cap \{\text{dependencies in summary}_{CONCRETE}\}|}{|\{\text{dependencies in summary}_{STATIC}\}|}$$

Experiment Results.

This experiment computed two Kendall rank correlation coefficients between a method's call graph size, and the precision of the method's aggregate and static dependence summaries. Such correlation coefficients were computed with each method invoked within specific executions, across all client subject programs. The experiment specifically uses the Kendall Tau-b test that handles ties in ranks, when computing the correlation coefficients. The Kendall Tau test is a non-parametric test that does not rely on, or assume a normal distribution in the data on which the test is performed. Further, it is does not assume any relation between variables being compared. Such characteristics made the Kendall Tau test a robust choice for computing the correlation coefficients.

The resulting correlation coefficients (Kendall tau values) between the static call graph sizes and the precision of the aggregate summaries, for methods invoked in each client subject, are reported in Table 8.4. Similar coefficients (Kendall tau values) for call graph sizes and static summary precisions are reported in Table 8.5. Further, both sets of correlations across all eight subject programs, are graphically plotted with scatter plots in Figures 8.5 and 8.6. The experiment anticipated a lowering of precision scores with an increase in the size of the

static call graph. As such, these coefficients were computed with the null-hypothesis that there is no negative correlation between the static call graph size and summary precisions (aggregate/static), for a given method in an execution. The resulting p-values are reported along with the tau values, for each client subject in Tables 8.4 and 8.5.

	NANOXML	JTOPAS	PL241	ANTLR	BLOAT	PMD	FOP	JYTHON
tau	-0.15	-0.34	-0.18	-0.09	-0.29	-0.28	-0.26	-0.19
p-val.	0.005	0.005	0.004	0.153	< 2.2e-16	2.2e-16	3.0e-14	4.6e-05

Table 8.4: Kendall Tau Correlation Coefficients for Aggregate Dependence Summary Precision vs. Method#

	NANOXML	JTOPAS	PL241	ANTLR	BLOAT	PMD	FOP	JYTHON
tau	-0.21	-0.22	-0.18	-0.13	-0.23	-0.29	-0.33	-0.28
p-val.	0.0002	0.0552	0.0071	0.0667	1.2e-07	1.5e-11	5.3e-16	4.1e-07

Table 8.5: Kendall Tau Correlation Coefficients for Static Dependence Precision vs. Method#

Tables 8.4 and 8.5 report low to modest correlation coefficients (tau value) indicating a weak effect of increasing call graph sizes on the precision scores for both aggregate and static summaries. Moreover, in the case of aggregate and static summary precision for the methods invoked in ANTLR and the static summary precision for methods invoked in JTOPAS, the p-values are greater than 0.05; thereby preventing the rejection of the null-hypothesis in those specific cases. Indeed, the tau values are consistently negative, thus suggesting that precision scores for method summaries do indeed reduce with increasing call graph sizes. However, given the low tau values, the results do not provide conclusive evidence of a clear correlation between the size of a method's static call graph and the method's summary precisions.

Such limited degrees of correlation can be explained by the notion that a method may invoke a variety of other methods that do not necessarily introduce any significant heap-data effects of their own. For instance, methods tend to invoke a chain of constructors that typically lack control structures (conditionals or loops); thus exhibiting consistent behavior across invocations, and resulting in moderate to high precision dependence summaries.

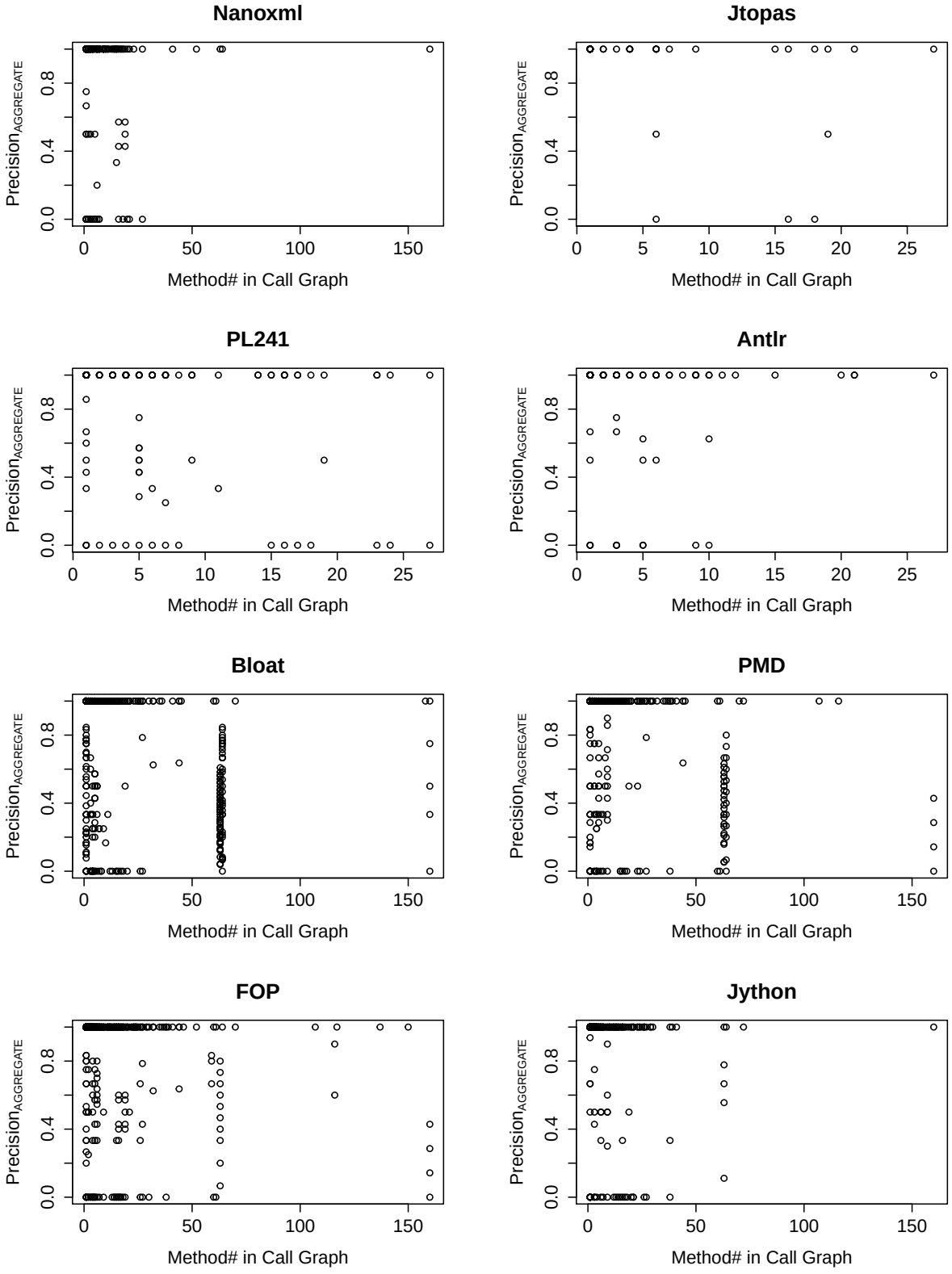


Figure 8.5: Precision$_{AGGREGATE}$ vs. Method#, when comparing concrete and aggregate dependence summaries

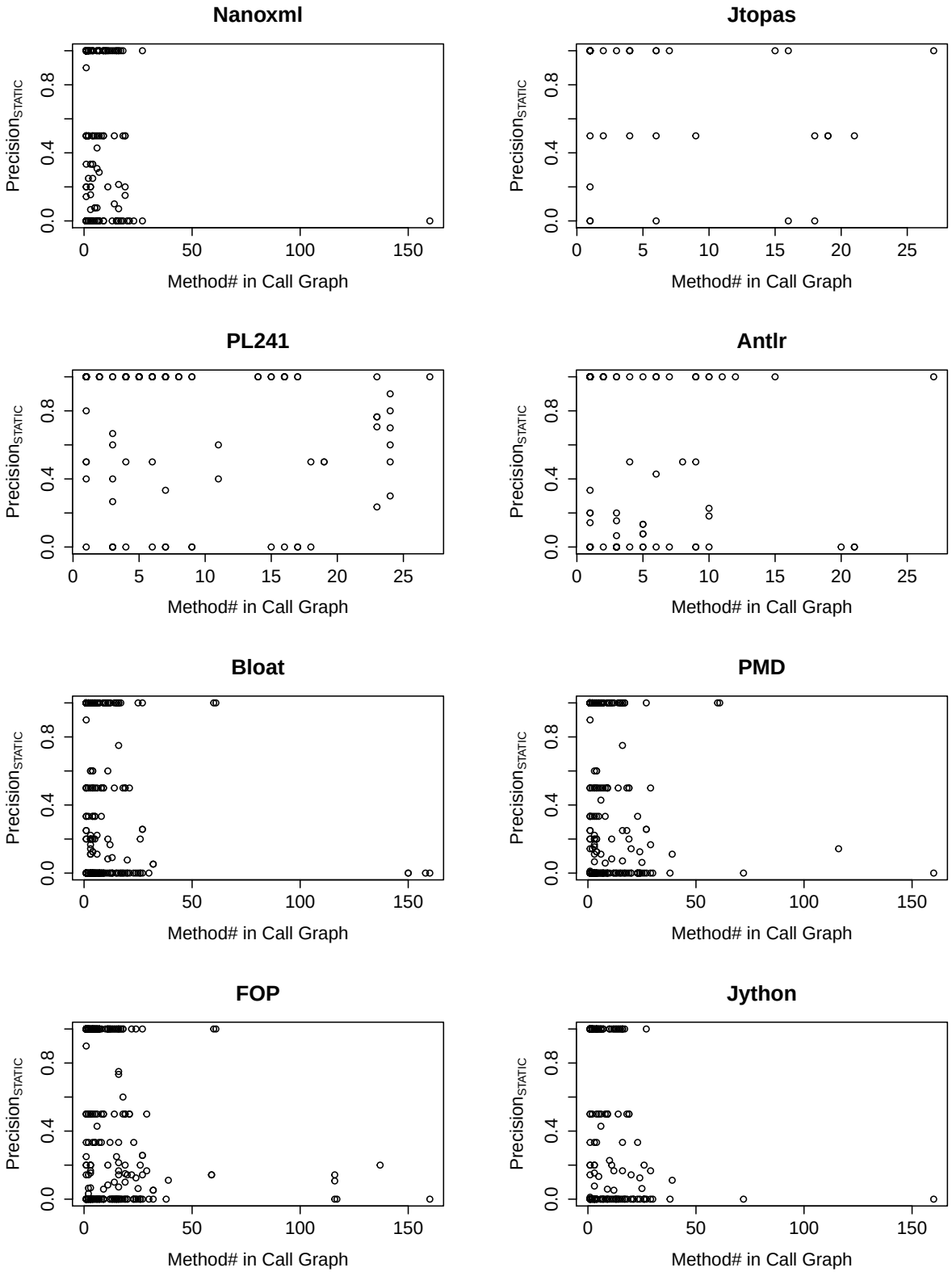


Figure 8.6: Precision$_{STATIC}$ vs. Method#, when comparing concrete and static dependence summaries

Chapter 9

Investigation of Similarity in Method Behavior

When taken together, Chapters 7 and 8 suggest that using method dependence summaries, in place of concrete summaries, results in both substantial performance gains, and modest accuracy losses. Particularly, the results of the accuracy investigation (see Section 8.2) suggest that statically generated summaries tend to add a perceptibly higher number of summary edges for a method, than its dynamic counterparts. Such results by themselves would motivate the use of dynamic generation of dependence summaries, over static generation. However, statically generating method summaries only incurs a one-time analysis cost for each version of a given method. Dynamically observed summaries may indeed incur the profiling costs of an ever-increasing number of invocations, due to varying method behavior from one invocation to another. As such static generation stands to present itself as a cheap, practical approach to creating dependence summary generation; despite its tendency for lesser accuracy when compared with dynamically generated summaries. This raises the question: to what extent can we save on profiling costs for when creating dynamic dependence summaries for methods?

Creating dynamic dependence summaries may require the profiling of a vast majority of method invocations, thus setting an expectation of high profiling costs for creating dynamic dependence summaries. However, if methods behave similarly, and exhibit similar dependence summaries across their multiple invocations, then only a limited number of invocations for a method would require exhaustive profiling to build a dependence summary that comprehensively models all of the method's externally observable heap-data effects. This essentially prompts the question: how similarly do methods behave across their invocations, when the method behavior is modeled as a dependence summaries?

Answering such a question on method behavior similarity stands to justify the usage of dynamic generation, over static generation, of dependence summaries. Similar behavior across method invocations would imply that only a limited number of method invocations need exhaustive profiling to create a comprehensive method dependence summary, thereby potentially incurring limited profiling costs. Such limited profiling costs would in turn support the use of dynamic dependence summaries, over static summaries. However, a lack of similarity in method behavior would indicate a wide variety of method behavior, thereby justifying the use of static analysis to comprehensively model the method's heap data effects. As such, a lack of method behavior similarity would essentially justify the application of static dependence summaries, despite their greater inaccuracies.

This chapter's goal with answering research questions RQ4a-c, is to understand the similarity in method behavior, as modeled by dynamic dependence summaries, (a) within executions, (b) across executions of the same subject and (c) across subjects. For instance, if a method's behavior is the same across all its invocations within a program run, then only one invocation of the method requires profiling to create the method's dependence summary. Similarly, if a method's dependence summaries are similar across different test runs of a program, then the method dependence summaries created from exhaustively profiling one test run can be used to effectively model method invocations in a different test run of the same program.

Such reuse of method dependence summaries, across program executions, and even across different programs, can effectively afford the omission of creating different method summaries for different executions, and thereby save on the profiling costs to create dynamic summaries. Moreover, understanding any such similarity will guide in sampling representative method invocations to dynamically build a dependence summary. Such an investigation will further inform on whether to reuse a dynamic dependence summary for a method that was created with invocations from a different software execution, or perhaps a different software program.

It is also important to note that this study of method behavior similarity, is carried out with the goal of creating generic models of dependence summaries that will model varying sets of dependencies for the same method. This is different from the accuracy studies presented in Chapter 8, where the intent was to neither under-approximate or over-approximate the set of dependence summary edges. With this investigation, the idea is to sample enough method invocations such that the resulting summaries do not under-approximate method behavior.

Further, all experiments in this chapter use the same set of 83 program executions and the resultant dynamic summaries for library methods, from the eight client programs, as used in Chapters 7 and 8. Additionally, all methods within the Java Standard library (*rt.jar*) are targets for summarization, just as with the prior experiments in this work.

9.1 RQ4a: Investigating Method Behavior Similarity within Software Runs

This section begins by describing the experimental setup for the following research question:

> **RQ4a:** How does the behavior of a method vary with successive invocations in a given software execution?

In the following experiment the goal will be to ascertain for a given summarized method the number of invocations that require dynamic profiling before all possible dependencies between the inputs and output of the method are discovered, within a single execution. Further, such a count of invocations will be studied in comparison with the number of edges in the dependence summary of the summarized methods.

As such, this experiment will measure and record the following two metrics.

Number of Summary Edges (summary#). The count of summary edges for a method's dependence summary discovered by a program execution.

Point of Saturation (invoke#$_{SAT}$). The number of invocations necessary to observe in order to model all summary edges in the method's dynamic dependence summary, for a given method executed in an execution.

Given the need to understand the similarity of dependence summaries of individual method invocations, this experiment analyzes the exhaustive traces of all client program test runs, used in this work. To compute the above two metrics, the experiment identifies all method invocations for a given method in each exhaustive execution trace, for all client subject programs. The experiment computes the concrete dependence summaries for the method invocations, and sorts those summaries in the order in which the corresponding method invocations were invoked, *i.e.,* chronologically. By iterating through such a list of chronologically ordered concrete dependence summaries, for a given method in an execution, the experiment is able to compute the number of additional, or new summary edges exhibited by a method invocation, with each passing concrete summary.

Thus, such a chronological analysis enables the identification of both (a) the total number of dependence edges exhibited by all concrete summaries for a method in an execution, *i.e.,* number of summary edges (summary#), and (b) the minimum number of chronologically ordered concrete summaries, or method invocations that required analysis to identify all

possible summary edges exhibited by all invocations of the method in an execution, *i.e.,* the point of saturation (invoke$\#_{SAT}$). The number of summary edges (summary#) and point of saturation (invoke$\#_{SAT}$) is computed for every method, in each execution that the method was invoked in, across all eight subject programs. The resulting data on summary# and invoke$\#_{SAT}$ are collected and reported for each client subject separately, in the following sub-section.

Chronology Experiment Results.

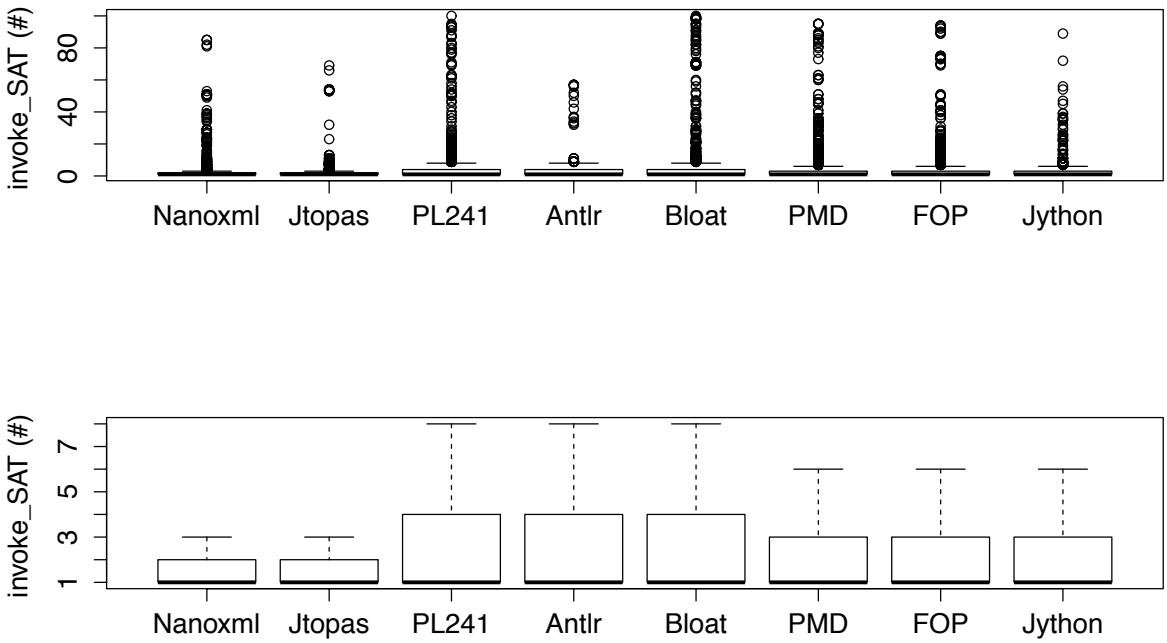


Figure 9.1: Distribution of Point of Saturations for methods executed in multiple executions of the client subjects (with and without the long-tail distributions)

invoke$\#_{SAT}$	NANOXML	JTOPAS	PL241	ANTLR	BLOAT	PMD	FOP	JYTHON
Median invoke$\#_{SAT}$	1	1	1	1	1	1	1	1
tau (invoke$\#_{SAT}$ vs. summary#)	0.60	0.65	0.70	0.62	0.74	0.70	0.71	0.66

Table 9.1: Median values for Points of Saturations (invoke$\#_{SAT}$), and Kendall Correlation (tau): Points of Saturations (invoke$\#_{SAT}$) vs. Number of Summary Edges (summary#), for multiple test runs across all eight subjects; as portrayed in Figures 9.1 and 9.3.

The results for the chronology experiment are reported collectively in Figures 9.1, 9.2 and 9.3,

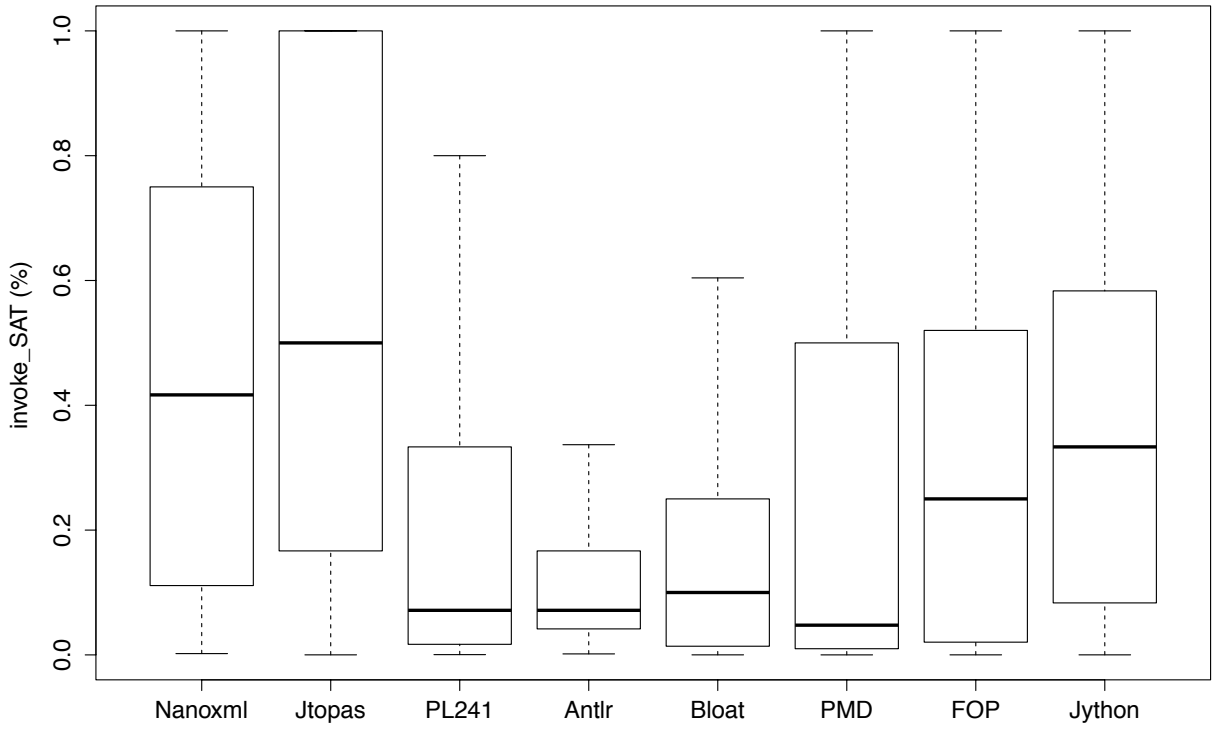


Figure 9.2: Distribution of Point of Saturations, as a percentage of a method's total number invocations within a subject execution, for different methods executed in multiple executions of the client subjects

and in Table 9.1. Table 9.1 in particular, reports certain summary statistics that are plotted graphically in Figures 9.1, and 9.3. Figure 9.1 plots a series of eight box plots (one per subject program) that show the distribution of invoke$\#_{SAT}$ for various methods invoked within each of the 83 client program executions. Figure 9.1 depicts two sets of box plots for the same data set of points of saturation, with and without depicting the long-tail distributions. Additionally, for an easier reading of the distributions the first row of Table 9.1 presents the median invoke$\#_{SAT}$ values that are plotted graphically in the lower part of Figure 9.1.

The results suggest, for instance, that for a typical method invoked in a test run of NANOXML, it required no more than three execution, at most, to identify all possible dependence edges exhibited by that method through out the execution. In fact these results suggest that for a

typical method, invoked across all eight subjects, no more than the first 8 method invocations were required to be analyzed to recover all of the method's summary edges, as exhibited by an execution.

The absolute number of invocations needed to create a comprehensive summary is relatively limited, for a typical method across executions. However, such a limited number of invocations for a given method in a specific execution may represent the total number of invocations for that method in the given execution. This is showcased by Figure 9.2 that plots the invoke$\#_{SAT}$ distributions as a percentage of the total number of invocations that were observed for a method in a given execution. For instance, a typical method executed in a single execution of NANOXML needs the analysis of only a single invocation. However, that typical method may have been executed just twice in a given execution of NANOXML.

Moreover, as depicted by the presence of a long-tail distribution in the upper boxplots of Figure 9.1, many methods indeed require more than a handful of invocations to be analyzed, to dynamically generate a comprehensive method dependence summary. As such, while the results suggest that a majority of the observed methods need limited analysis of their invocations, it is important to gain a further understanding of when additional methods may need to be analyzed.

Point of Saturation vs. Summary Edge Count.

More invocations of a given method may need analysis to uncover all of the method's exhibited dependence summary edges for two potential reasons. A method simply might have more dependence edges in its summary and the analysis of additional invocations in the execution may help recover those dependence edges. Alternatively, the method may indeed have few summary edges but it may take multiple executions to discover some of those limited edges, perhaps because they represent exceptional flows in a method's invocation, thus resulting in a higher point of saturation for such methods.

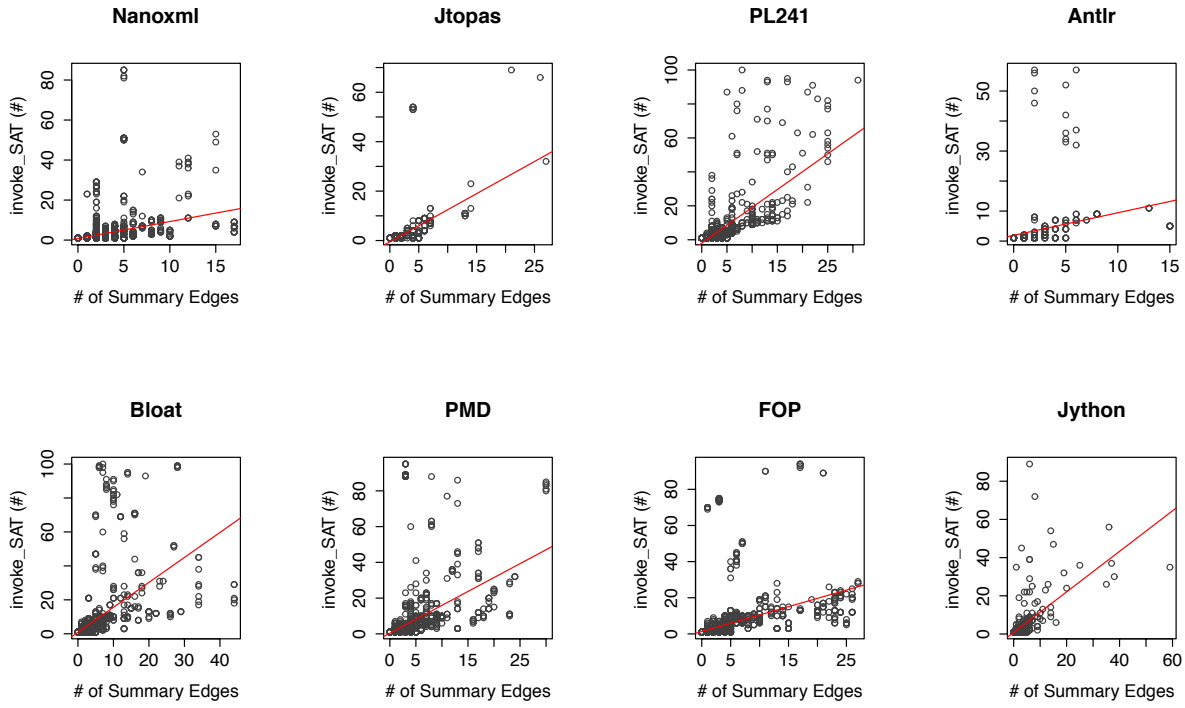


Figure 9.3: Point of Saturation and the Number of edges in the corresponding Dynamic Dependence Summaries

To better understand which reasoning to be more likely, this experiment investigates a correlation between the number of summary edges (summary#) and the point of saturation (invoke$\#_{SAT}$) for a give method invoked in a specific execution. A positive correlation between summary# and invoke$\#_{SAT}$ for a given method would suggest that the point of saturation would increase for a method with an increase in the number of summary edges. Thus indicating that methods with limited summary edges would indeed have a smaller point of saturation with a given invocation, and not require multiple invocations to discover a limited number of summary edges.

As such, this investigation computed a Kendall rank correlation coefficient between the number of summary edges and the points of saturation, for the methods invoked within specific executions. Different Kendal rank correlation coefficients were computed for each client subject program. The experiment specifically uses the Kendall Tau-b test that handles

ties in ranks. The Kendall Tau rank test is non-parametric test that does not rely on, or assume any distributions in the data on which the test is performed; thus making this test a robust choice for computing the correlation coefficients.

The resulting correlation coefficients (Kendall tau values), for all eight client subjects, are reported in the second row of Table 9.1. The correlations for all eight subject programs are plotted with scatter plots in Figure 9.3. These coefficients were computed with the null-hypothesis that there is no positive correlation between the number of summary edges and the points of saturation for a given method in an execution. For all eight correlation coefficients, the p-value is $< 2.2e-16$, suggesting that the correlation coefficients are statistically significant, allowing the rejection of the null-hypothesis.

The tau correlation coefficients, as reported in Table 9.1, range from a minimum of 0.6 for NANOXML, to a high of 0.74 for BLOAT. Such positive coefficients indicate a clear, positive correlation between the number of summary edges (summary#) and the point of saturation for a given method invoked in a specific test run, across all eight subject programs. Such a correlation between the number of summary edges and the point of saturation for a method, helps explain that certain methods need more invocations to be analyzed early in a program's execution — they likely have more summary edges to be uncovered.

These results collectively indicate that when a method's invocations are analyzed chronologically within a single program test run, they are likely to reveal all dependence summary edges between the method's inputs and outputs. The results for the test runs of the experimental subjects suggest that most methods may require no more than eight of their first invocations to be analyzed to create comprehensive summaries for that method. However, as indicated by the long-tail distributions shown in the upper boxplots of Figure 9.1 there are exceptional methods that may require more of their invocations analyzed. Such methods require the analysis of additional invocations given that the likelihood that they simply have more summary edges to be discovered.

While necessary in exceptional cases, analyzing an extended number of invocations for a method may actually result in the analysis of all invocations of the method in a given execution, as shown in Figure 9.2; thereby resulting in limited savings in profiling costs while building a comprehensive dependence summary for that method. As such, this leads the investigation to the next question: can a method's dynamic dependence summary that is created by profiling one test execution of a program, be used to model the method's invocations in a different execution of the same program? The following section discusses this question and provides experimental data to answer it.

9.2 RQ4b: Investigating Method Behavior Similarity across Software Runs

The goal with this line of investigation is to assess the extent to which aggregate dynamic dependence summaries for a method can be created by monitoring method invocations in a single software run, for usage in the dynamic dependence analysis of a different execution, of the same software system. This investigation is framed with the following research question:

> **RQ4b:** To what degree is the behavior of a method, as modeled by dynamic dependence summaries, similar across different executions of a client software subjects?

As such answering RQ4b will require the experiment to create aggregate dynamic dependence summaries for methods within each individual test execution, for a given subject program. The experiment then compares resulting dependence summaries from one test run with the dependence summary produced from entirely different test runs of the same subject program. The experiment specifically compares the summary edges between the dynamic dependence summaries generated from two different program executions. The degree of similarity is quantified by the experiment with a set comparison of such summary edges, specifically with

the use of the Jaccard Similarity coefficient metric.

Summary Similarity (J(summary$_A$, summary$_B$)). The Jaccard similarity coefficient for two aggregate dynamic dependence summaries, of the same method.

$$\mathrm{J}(\mathrm{summary}_A, \mathrm{summary}_B) = \frac{|\{\text{dependencies in summary}_A\} \cap \{\text{dependencies in summary}_B\}|}{|\{\text{dependencies in summary}_A\} \cup \{\text{dependencies in summary}_B\}|}$$

Dynamic Summary Similarity Across Program Executions.

The results for this experiment are presented in Figure 9.4. Figure 9.4 presents a set of eight boxplots that represent the distribution of Jaccard Similarity scores when dynamic dependence summaries for methods are created and compared across executions of the same program, as described in the previous paragraph.

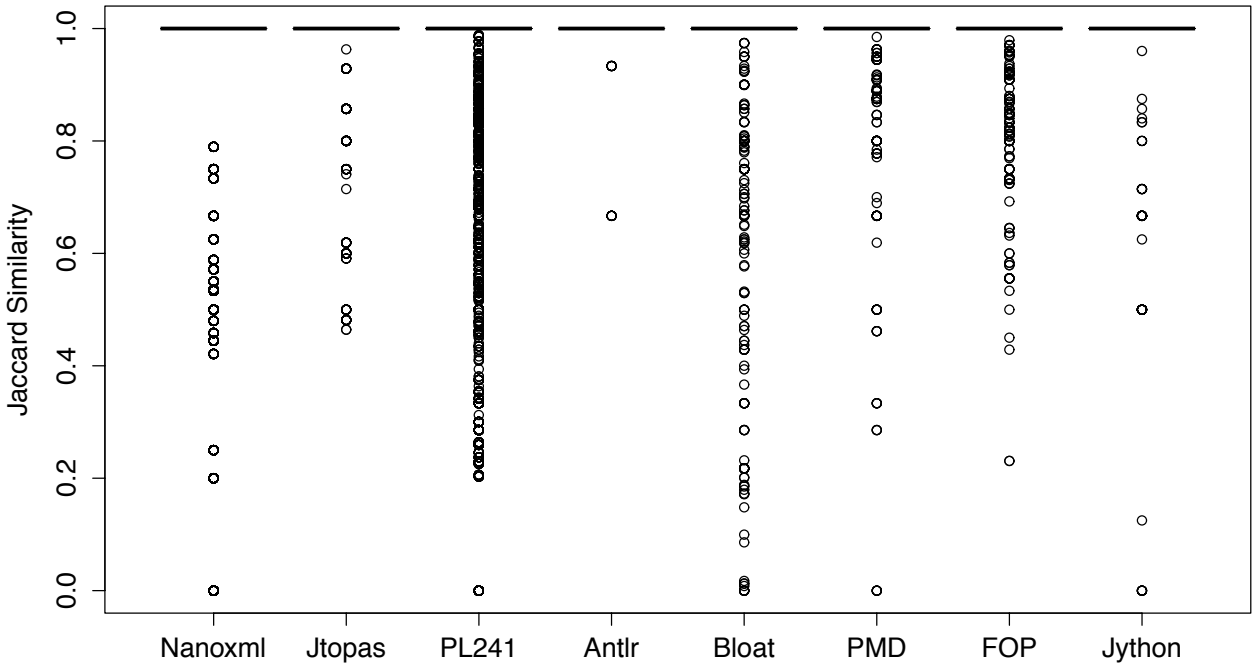


Figure 9.4: Distribution of Jaccard Similarity scores when comparing dynamic dependence summaries for methods, across executions of a given program.

The boxplots show consistently high similarity scores of 1.0 when method summaries are created and compared across executions of a given client program. Such consistently high similarity scores can be explained by the fact that the summaries are created as a result of observing individual system-wide tests. Such system-wide tests, even with varying test inputs, are likely to exercise a similar and exhaustive set behaviors. Such high similarity scores essentially suggest that the dependence summaries created for a method from two different executions of a client program are likely to be the same. As such, these results support the idea that method dependence summaries when created from one execution of a client program, can likely be reused to effectively model the dependencies between inputs and outputs for invocations in entirely different executions of the same client program.

9.3 RQ4c: Investigating Method Behavior Similarity across Software Systems

The goal with this investigation is to assess the extent to which aggregate dynamic dependence summaries for a method can be created by monitoring method invocations across executions of a software system, for usage in the dynamic dependence analysis of a different software system. Formally, this line of investigation is framed with the following research question.

> **RQ4c:** To what degree is the behavior of a method, as modeled by dynamic dependence summaries, similar across different client software subjects?

As such answering RQ4c will require the creation of aggregate dynamic dependence summaries for all methods across multiple executions of a software system, for multiple different software systems. Just as with the experiment for answering RQ4b, this experiment compares such aggregate dynamic dependence summaries for similarity. Specifically the experiment

obtains a similarity score for each such comparison using Jaccard similarity coefficient metric, which is restated here from Section 9.2.

Summary Similarity (J(summary$_A$, summary$_B$)). The Jaccard similarity coefficient for two aggregate dynamic dependence summaries, of the same method.

$$\text{J}(\text{summary}_A, \text{summary}_B) = \frac{|\{\text{dependencies in summary}_A\} \cap \{\text{dependencies in summary}_B\}|}{|\{\text{dependencies in summary}_A\} \cup \{\text{dependencies in summary}_B\}|}$$

Dynamic Summary Similarity Across Subject Programs.

This experiment compared a method's dependence summaries that were generated from all executions of one subject, with the method's dependence summaries from an entirely different subject. When comparing the dependence summaries generated from the executions of 8 subjects, a total of twenty-eight comparisons between subject pairs were obtained. Figure 9.5 plots the distributions of similarity scores for method dependence summaries across the twenty-eight subject pairs.

As shown in Figure 9.5, a majority of the comparisons across the subject pairs demonstrate consistently high levels of similarity. Moreover, for all subject pairs, the median similarity scores are at 1.0 — the maximum possible similarity score — suggesting that at least half the methods compared across all subject pairs resulted in the same dynamic dependence summary.

In practical terms, such perfect median similarity scores suggest that a dynamic dependence summaries created from the execution of one client program, has at least a 50% likelihood to model the dependence summary for method invocations in an entirely different client program — for commonly executed methods between the two client programs.

However, Figure 9.5 also shows variance in the distribution of similarity scores for twelve

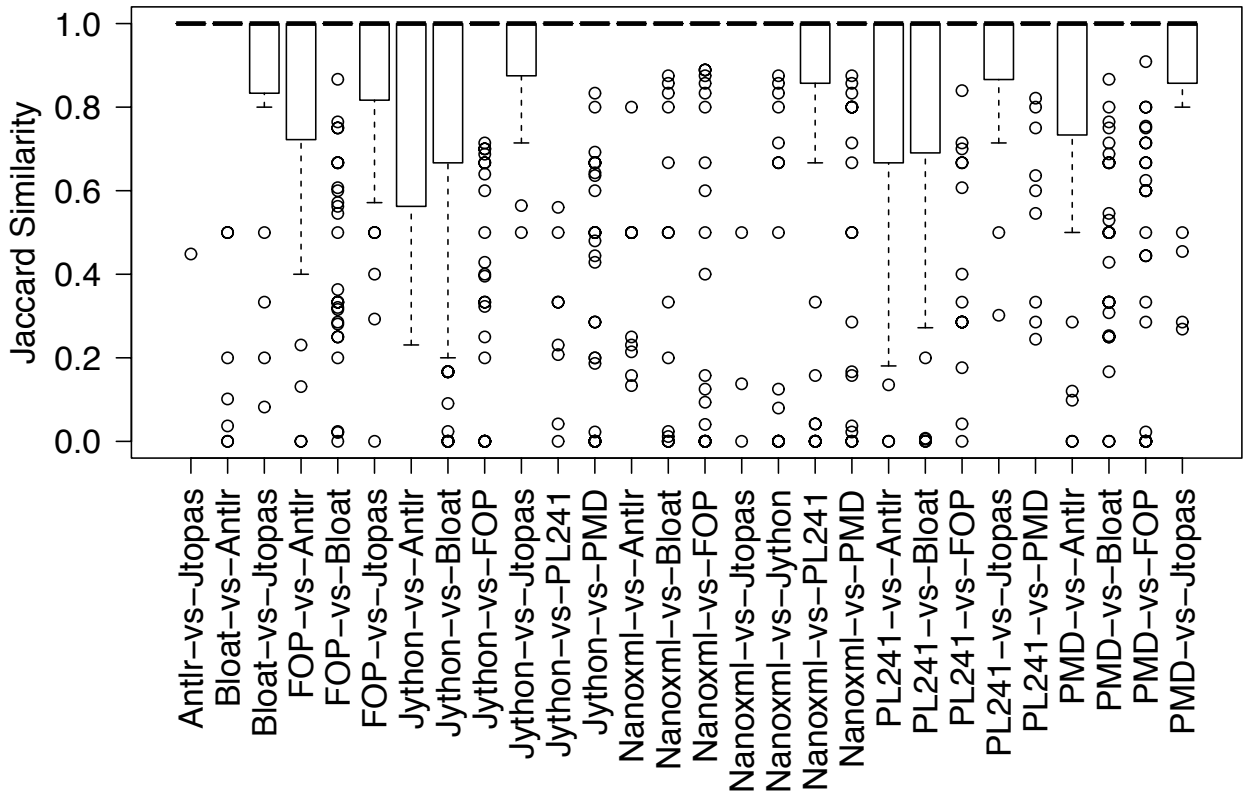


Figure 9.5: Similarity between Dynamic Dependence Summaries for Methods, across Multiple Client Program

Subject$_A$ vs. Subject$_B$	Minimum	1^{st} Quartile	Median	3^{rd} Quartile	Maximum
PL241 vs. Antlr	**0.18**	**0.67**	1.00	1.00	1.00
Jython vs. Bloat	**0.20**	**0.67**	1.00	1.00	1.00
Jython vs. Antlr	**0.23**	**0.56**	1.00	1.00	1.00
PL241 vs. Bloat	**0.27**	**0.69**	1.00	1.00	1.00
FOP vs. Antlr	**0.40**	**0.72**	1.00	1.00	1.00
PMD vs. Antlr	**0.50**	**0.73**	1.00	1.00	1.00
FOP vs. Jtopas	**0.57**	**0.82**	1.00	1.00	1.00
Nanoxml vs. PL241	**0.67**	**0.86**	1.00	1.00	1.00
PL241 vs. Jtopas	**0.71**	**0.87**	1.00	1.00	1.00
Jython vs. Jtopas	**0.71**	**0.88**	1.00	1.00	1.00
Bloat vs. Jtopas	**0.80**	**0.83**	1.00	1.00	1.00
PMD vs. Jtopas	**0.80**	**0.86**	1.00	1.00	1.00

Table 9.2: Similarity Scores where methods exhibit varying behavior across client subjects, not including the outliers displayed in the box plots of Figure 9.5

(out of 28) subject pairs. Such a variance in the similarity scores suggest that dependence summaries produced for methods across client subjects were not always the same. Table 9.2 presents statistics for the similarity score distributions, for the subject pair comparison that exhibit variance in the similarity scores. For instance, Table 9.2 reports that a low similarity score of 0.18 was exhibited between a method's dynamic dependence summaries when generated from execution of PL241 and ANTLR. Additionally, as reported with the 1^{st} Quartile statistic, about 75% of the methods common to the executions of PL241 and ANTLR showed a similarity of 0.67 or greater in their dependence summary edges produced independently from the two client programs.

Despite the high median similarity scores of 1.0, such variances exhibited by a substantial number of subject pairs indicate that about 50% of the methods common within these subject pairs were indeed used differently by different subject programs. As such, the dynamic dependence summaries generated from observing the executions of one subject program are likely to model the method invocations in an entirely different program with inaccuracies.

Moreover, the experiment in Section 9.2 compared dynamic dependence summaries across executions of the same subject, and reported consistently high similarity scores of 1.0. As such, it is clear that executions of the same client program exhibit similar usage of the methods that were summarized and compared, as opposed to executions from different client subject programs. Thus, as a practical concern, the likelihood of accurately modeling the heap-data effects of method invocation is higher when using dependence summaries generated from observing executions of the same client program, as opposed to the execution of an entirely different client program.

Method's Characteristics vs. Summary Similarity Across Subjects.

A method is likely to exhibit greater variance in its behavior, and thus its dynamic dependence summary, due to varying control flow. Further, such control flow is likely to compound

with the control flow of other methods that are invoked directly or indirectly from the original method of interest. Thus, such characteristics of a method are likely to affect how its dynamic dependence summary varies when created by observing the executions of different client programs. This experiment quantifies such characteristics for a given method with the following metrics:

Static Call Graph Size (method#). The number of methods in the static call graph that is rooted at the given method.

Number of Flow Redirections (jump#) The number of jump and invoke instructions in the method's inter-procedural control flow graph.

After creating and comparing the aggregate dynamic dependence summaries across client subject pairings, this experiment compared the resulting similarity scores with the static call graph size (method#) and the number of flow redirections (jump#) of the methods in question. Specifically, the experiment computed Kendall rank correlation coefficients between (a) the similarity score for a method's dynamic dependence summary when compared across two client programs; and (b) method's static call graph size (method#) and number of flow redirections (jump#).

The experiment specifically uses the Kendall Tau-b test that handles for ties in ranks. The Kendall Tau rank test is non-parametric test that does not rely on, or assume any distributions in the data on which the test is performed; thus making this test a robust choice for computing the correlation scores.

The resulting correlation coefficients (Kendall tau values), for the methods whose summaries were compared across client subjects, are as follows:

summary similarity vs. method# : tau = -0.2732877; p-value $<$ 2.2e-16

summary similarity vs. jump# : tau = -0.3047052; p-value $<$ 2.2e-16

Additionally both sets of correlations — with static call graph size, and number of flow

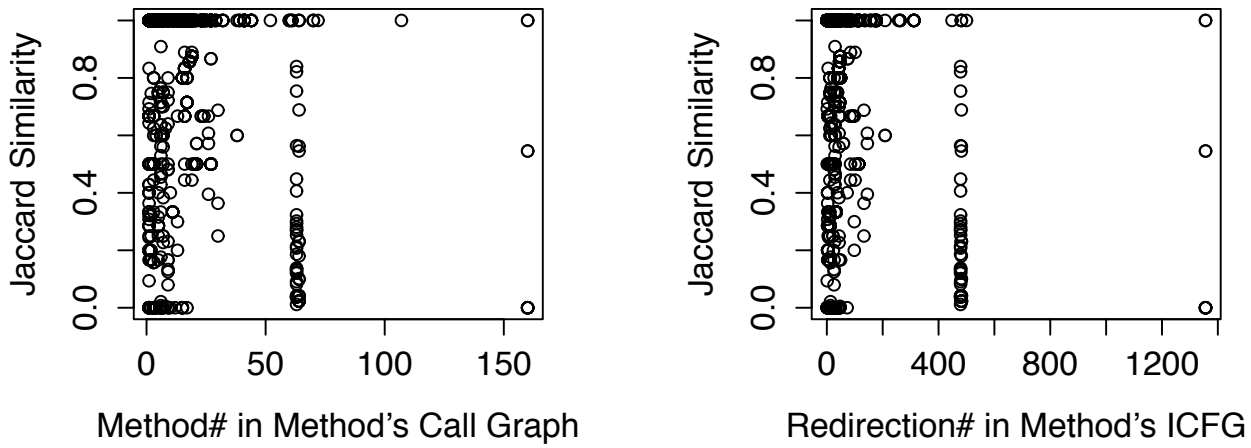


Figure 9.6: Similarity for methods across multiple client programs with a) increasing method call graph sizes, and b) increasing number of flow redirections in method ICFGs

redirections — are plotted with two scatter plots in Figure 9.6. These coefficients were computed with the null-hypothesis that there is no negative correlation between the similarity of method dependence summaries across client programs and the two metrics of this experiment (method# and jump#). For both correlation coefficients, the p-value is $< 2.2e-16$, suggesting that the correlation coefficients are statistically significant, allowing the rejection of the null-hypothesis.

These correlation coefficients suggest a modest negative correlation with increasing method# and jump#. In other words, the similarity between a method's dependence summaries — computed from the executions of different client subjects — has a weak likelihood of reducing with an increase in the size of the method's static call graph (method#) or the number of redirections in the method's inter-procedural control flow graph (jump#).

Such limited degrees of correlation can be explained by the notion that different branches of a control flow structure, such as an if-statement, can exhibit similar, and even the same set of dependence relations between the externally accessible inputs and outputs for a given method. Conversely even a single jump expression may result to drastically different set of dependence relations between the externally accessible inputs and outputs for a given method. As such, the actual set of dependence relations between a method's inputs and

outputs are affected to a limited extent by the number of other methods invoked by the method, or the number of control structures that are likely to be executed when invoking the method.

The results presented in this chapter collectively indicate that methods behave similarly across their different invocations, within and across executions. The results for RQ4a suggests that a limited number of a method's invocations need exhaustive profiling to construct a comprehensive summary of the method's heap-data effects. The results for RQ4b and RQ4c suggests that methods when invoked across program executions tend of exhibit similar heap data effects, and method summaries created from exhaustively profiling one program execution can be use to adequately model the invocations of the method in an entirely different execution of the same program. However, the results for RQ4c also observes that summarized dependencies modeled from the execution of one program, may not always correctly model the method invocations within the execution a different program.

From a practical standpoint, consider that a developer needs to dynamically analyze the dependencies between instructions and memory locations in a software program under development, similar to the ones used in this study. The results in this chapter imply that the developer would require to exhaustively profile at least a single test run to create dynamic dependence summaries. The developer can then reuse such dynamically created summaries, for analyzing the dependencies across multiple other test runs of the same program, without exhaustively profiling the other test runs, thus saving costs in both: creating dynamic dependence summaries, and dynamically analyzing the dependencies within the program's test runs.

Chapter 10

Case Study with Runtime Client Analysis: Dynamic Slicing

Motivated by the cost savings found from the performance investigation (see Chapter 7) and the accuracy measures of dynamic and static dependence summaries in Chapters 8 and 9, this work next investigated the extent to which these savings are realized at the expense of accuracy for an actual client analysis. Through this case study, this work investigate the impact of reused dependence summaries on the accuracy of dynamic dependence analysis, specifically dynamic slicing for debugging. As discussed in Chapter 1 summary-based dynamic dependence analysis can be both unsound and imprecise. However, the degree to which this affects the results of an analysis in practice, is yet to be known within the context of an actual software engineering technique. As such, this case study is designed to answer this question, at least for the experimental software subject — NANOXML.

Using a training input of NANOXML, and the 20 subsequent faulty inputs, this case study investigates this issue by performing backward dynamic slicing to determine accuracy of the slice. The training input for NANOXML, which is different from the 20 faulty inputs,

is used to generate dynamic summaries for the Java library methods used by NANOXML. The investigation injected corruptions, or "faults", in the the 20 inputs to NANOXML that resulted in corrupt and incorrect execution state early in NANOXML's execution — during the reading of the data files. Further, the investigation identified the slicing criterion, for each corruption across the 20 inputs by monitoring the output stream and identifying the first point that the output from the faulty input differs from the output of the correct input. That is, the slicing criterion is defined as the output instruction (and its outputted data) being executed at the moment that the test-case output first violates the test oracle (*e.g.,* the first character difference). The investigation then slice based upon that instruction, the variable that was used to hold the externally observed incorrect data, and the specific execution instance of that instruction (remember, the execution traces in this work include all execution instances of each instruction). The goal was to localize the first runtime instruction in each of NANOXML's 20 executions that read in the incorrect data, which are treated as the faulty instruction instance for the purpose of this experiment. As such, the sliced faults are "deep" — the fault execution occurs at the beginning of the execution trace during the reading of input data, the propagation of the fault's state infection spans nearly the entire execution, and the slicing criterion is placed at the output manifestation near the end of the execution.

Case Study Metrics. The case study presents its results with the following metrics.

Metric 1: Found Bugs (B). The ratio of the dynamic slices that include the fault. This metric determines the degree of unsoundness that the summary-based dynamic analysis brings.

Metric 2: Size of the Slice ($\mathbf{S_{slice}}$). The size of the resulting slice, as a set of runtime-instructions.

Metric 3: Slicing Time ($\mathbf{T_{slice}}$). The time to required to compute the dynamic slice.

Metric 4: Program Runtime Overhead (RO). A measure of slowdown of the original non-instrumented program, in terms of a multiplicative factor of the original time. The overhead is computed as

Metric	Exhaustive	Summary
Found Bugs (ratio)	20/20	18/20
Mean Size of the Slice (# of runtime-instructions)	5019.2	7093.1
Mean Slicing Time (seconds)	84.5	67.2
Mean Program Runing Time (seconds)	527.74	1.70
Mean Program Runtime Overhead (ratio)	7538.1	23.3

Table 10.1: Slicing Study Results

$$\text{RO} = \frac{(\textit{instrumented time}) - (\textit{non-instrumented time})}{(\textit{non-instrumented time})}.$$

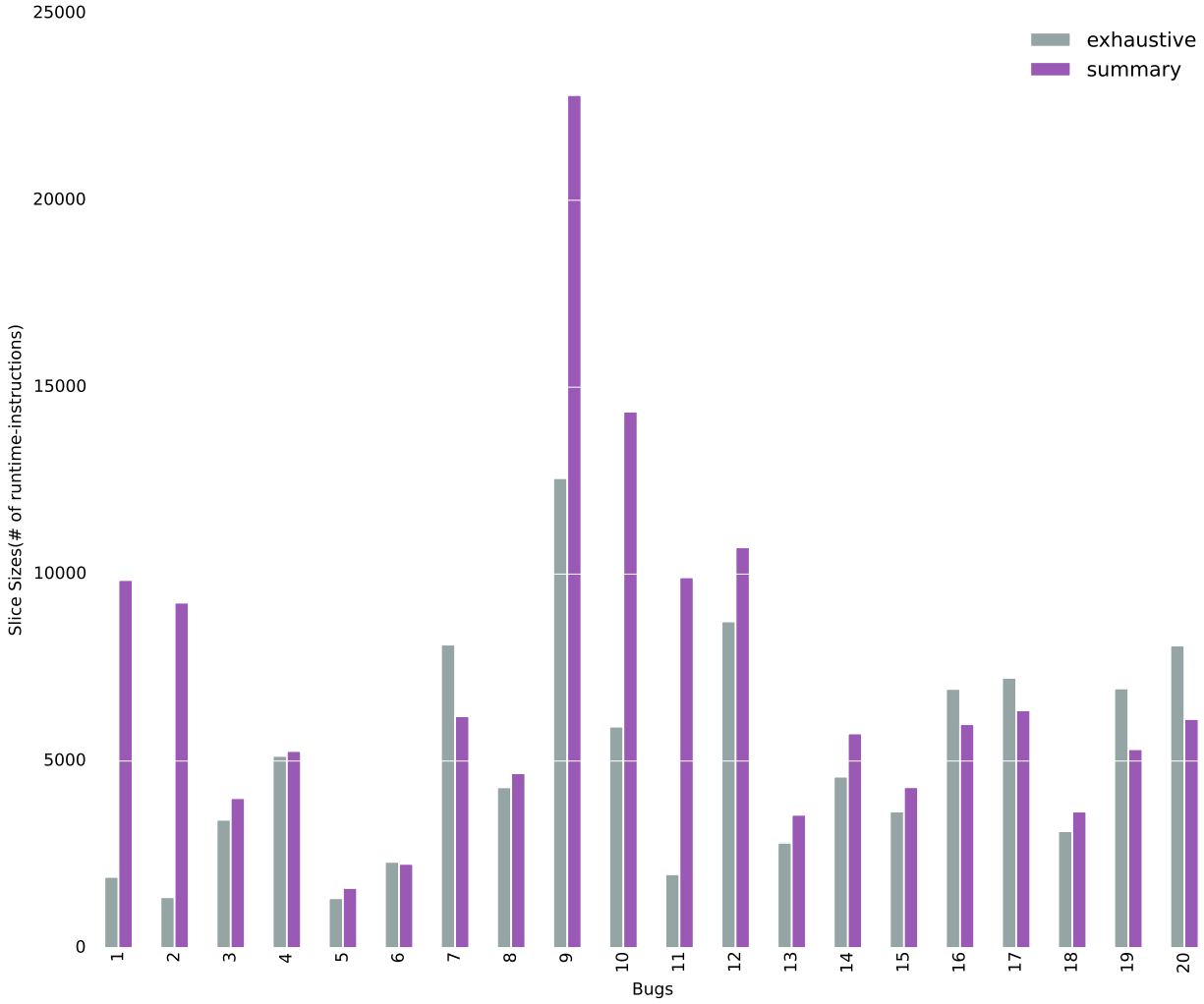


Figure 10.1: Slice Sizes (S_{slice}).

Table 10.1 and Figures 10.1–10.2 collectively report the results for the case study. Table 10.1 shows the aggregated data across all 20 bugs, whereas Figures 10.1–10.2 break down results

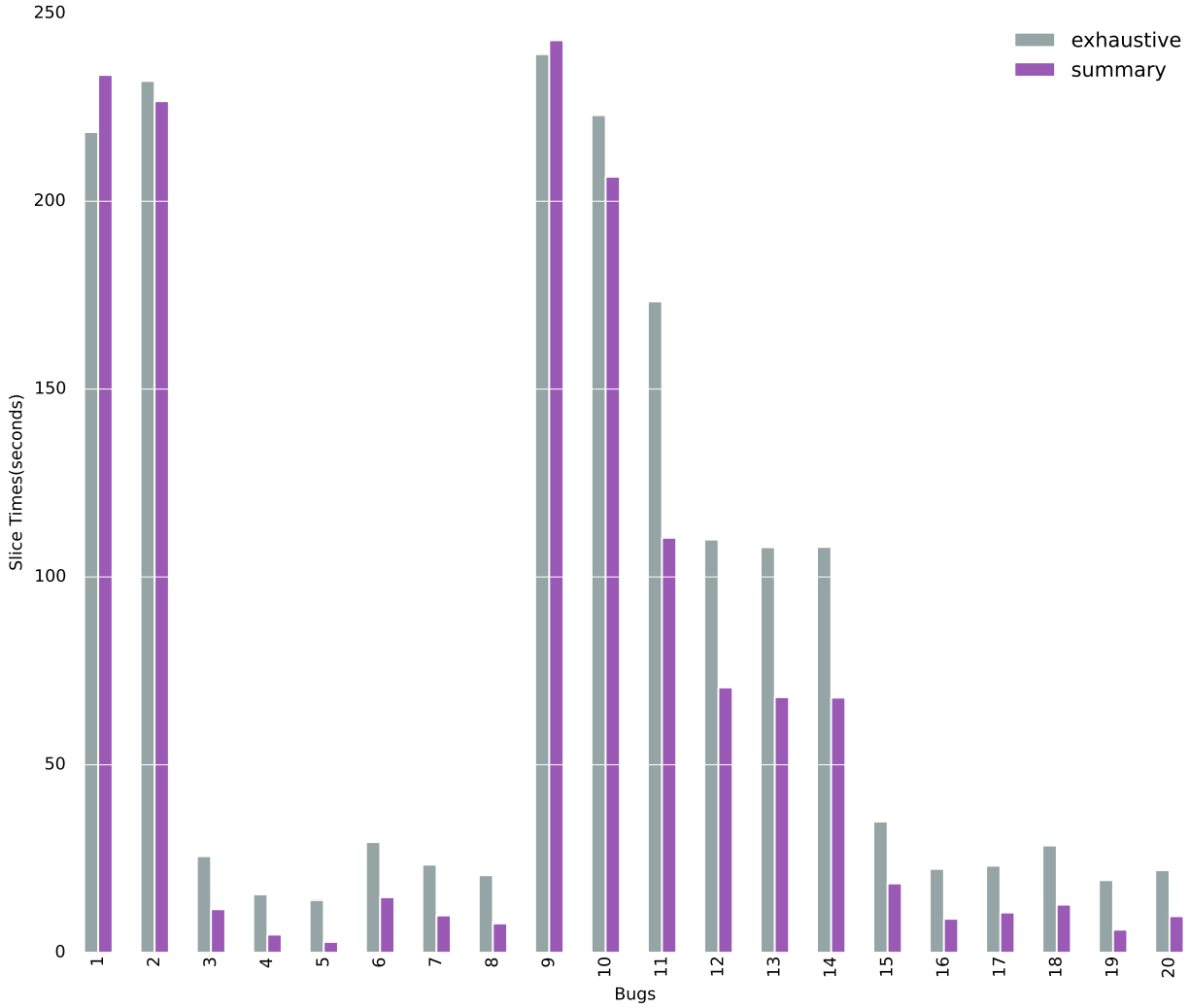


Figure 10.2: Slicing Time (T_{slice}).

per bug. Figure 10.1 presents the size of the resulting slices for each bug, for each technique, in terms of the unique source-code instructions in the slice. And, Figure 10.2 presents the time required to compute the dynamic slice.

Case Study Results. When examining the bugs that were being inspected, the investigation found that 18 out of the 20 bugs were localized in the dynamic slices that were computed with the summary-based technique. Such a localization rate is attributed to faults propagating through multiple dependency chains, and as such, the ability to slice back to the fault depends on at least one such chain persisting. The exhaustive slicing technique found 100%

of the bugs, as expected. The summary-based technique missed 2 out of the 20 bugs, and this was due to missing dependencies from the training of dynamic summaries.

It is also worth noting that the slice sizes on average are larger with the usage of dependence summaries, suggesting an introduction of spurious dependence edges by the dynamic dependence summaries. However, certain slices are smaller when using dependence summaries, suggesting dependence edges that were not modeled by the summaries for certain methods. These effects are indicative of the unsound and imprecise nature of how dependencies are modeled by dynamic dependence summaries. Essentially, these results are congruent with the results presented in the Accuracy Experiment (see Chapter 8) that indicated modest losses of accuracy upon the reuse of dynamic summaries.

Considering the costs of analyzing the program's execution (indicated by the Program Runtime Overhead) and the cost of computing the dynamic slice, the costs for exhaustive dynamic slicing are substantially more than the costs for summarized dynamic slicing. Particularly, the runtime overhead costs of exhaustive monitoring of a program's execution were dramatically greater than those of summary-based approach — and these are costs that would be incurred for *every* execution to be sliced.

Chapter 11

Analysis and Discussion

This chapter discusses the results of all four studies presented in Chapters 7 to 10, and looks to draw more general conclusions. Such a discussion is followed by specific discussions for all four studies, where each such discussion revisits the research questions that motivated this work's investigation. Further, the discussion of any broader conclusions are contextualized with threats to validity for this work.

When viewed collectively, this work's empirical results suggest that using method dependence summaries to model method invocations can bring down the costs of profiling and analyzing dynamic dependencies in program runs, while introducing inaccuracies in the dependencies between the inputs and outputs of method invocations. Indeed, the results suggest that both statically and dynamically created method dependence summaries over-approximate the set of input-output dependencies exhibited by a method invocation. However, the results also suggest that method dependence summaries when created dynamically within the context of individual program test runs, are likely to be more precise than their static counterparts. Moreover, such dynamic dependence summaries can be reused to effectively model dependencies exhibited by method invocations in other test runs of the same program.

From a practical standpoint, such results suggest that when performing dynamic dependence analysis for a given software program, the use of dependence summaries for methods in a specific library, can bring down the costs of computing dynamic dependencies with in the executions of the original program under analysis. The extent of such cost savings are dependent on the degree to which the program under analysis invokes the methods within the underlying library. Moreover, given the relative similarity between the statically and dynamically generated method dependence summaries, creating such summaries statically would be more effective, especially due to the cheap, one time cost of statically analyzing the binaries of the components being summarized. In particular, generating summaries statically is a safer option, as opposed to dynamic summaries, when the eventual client program that uses the library is unknown. For instance, vendors of software libraries may chose to package such dependence summaries along with the library's binaries. Such library vendors would then statically generate such summaries to conservatively model all possible dependencies exhibited by a method's invocation, given that the library will be agnostic to the client programs that actually invoke the methods.

That said, consider that a developer of a particular client program is using a library, and wants to use dependence summaries for the library methods in order to dynamically analyze the dependencies between the instruction and memory locations within execution of her own program. Indeed, the empirical results in this work suggest that dynamic dependence summaries, are likely to be more precise, than their static counterparts. In such an event the developer would be better suited to create dynamic dependence summaries by profiling at least one test execution of the program, and reusing such dependencies in analyzing multiple other executions of the same program. Recall that dynamic summaries created from one program run can be used to effectively model the executions of an entirely different test run of the same program. As such, the costs incurred in creating and effectively using dynamic summaries would be limited, and would be better suited to the executions of the specific client program under analysis.

Given such broader highlights of the empirical results of this work, the remainder of this chapter revisits the specific research questions of this work and discusses the related empirical results in an effort to answer those specific research questions.

Discussion of the Performance Investigation.

The first experiment investigated the extent of cost savings in analyzing software executions, when applying summary based analysis to methods in the Java Standard library. The results of this investigation suggested that the eight client subject programs, on average, relied substantially on methods from the Java Standard Library. For instance, the summary based analysis saved up to up to 77% in median execution trace sizes for certain program executions that relied on methods from library code. Conversely, the summary based analysis for test runs of JTOPAS resulted in the least trace size reductions (6%). That said, even the 6% reduction in trace sizes represented the analysis — and associated cost savings — of greater than hundred thousand instructions, on average, in each test execution of JTOPAS. Similar reductions in runtime overheads were observed when comparing the overheads introduced by exhaustive and summarized analyses. The summary based analysis resulted in the largest reduction of runtime overheads in the case of BLOAT, with a reduction from 454.15× (with exhaustive analysis) to 97.42× (with summarized analysis). Simultaneously, the least reduction was observed in the case JTOPAS, with a reduction in runtime overhead from 252.68× (with exhaustive analysis) to 231.22× (with summarized analysis).

Such variance in the degrees of cost savings are largely dependent on the degree to which a specific subject program invokes methods from the standard Java libraries. Indeed, methods not necessarily from the standard Java libraries can be targets for summarization, potentially prompting further cost reductions. It is also worth noting that such significant cost reductions, particularly in trace sizes, are further passed on to downstream client analyses that can avoid the analysis of library specific execution traces, when performing specific

software engineering analyses — for instance, memory bloat analysis, fault localization with dynamic slicing, detection of security vulnerabilities with taint analysis, or understanding code changes with change impact analysis.

As such, **to RQ1, this investigation assesses:** That the reuse of dependence summaries caused the costs involved in performing dynamic dependency analysis to be significantly reduced; both in terms of the additional time spent in performing the analysis and memory necessary to store the results of the analysis.

Discussion of the Accuracy Investigation.

In terms of analysis accuracy, this work investigated the effects of aggregating concrete summaries and performing static analysis towards creating method dependence summaries along three directions. First, the investigations compared concrete summaries, which model dependencies between the inputs and outputs for individual method invocations, with the aggregated dynamic summaries and static dependence summaries of the methods in question. Second, the accuracy investigation examined the similarity between a static dependence summary and aggregate dynamic summary, for a specific method of interest. And finally, this work also studied the presence of any correlation between the accuracy of a method's dependence summary (static and aggregate), with the size of the method's static call graph.

The first set of investigations into the accuracy of dependence summaries – generated statically, or dynamically – when compared to dependencies exhibited by individual method invocations, point to the following tendencies:

1. First, both static, and aggregated dynamic dependence summaries are likely to model all dependencies between a method invocation's inputs and outputs. Such an inference is based on the consistently high recall scores of 1.0 for both aggregated dynamic dependence summaries (see Section 8.1 and Figure 8.2), and the statically analyzed

dependence summaries (see Section 8.2 and Figure 8.3).

2. Second, both aggregated dynamic dependence summaries and static dependence summaries, introduce additional, spurious dependencies when compared with the concrete summaries for individual method invocations. The presence of such spurious dependencies are identified with a loss of precision score in a method's dependence summary, when compared to a concrete summary.

The high recall scores and the modest precision scores for both static and aggregated dynamic dependence summaries suggest that summarization is likely to result in the over-approximation of method behavior, as opposed to under-approximating the set of dependencies between the inputs and outputs of individual method invocations.

To study the potential for aggregated dynamic summaries to result in under-approximation, the investigation aggregated limited, random samples of concrete summaries for individual method invocations within individual executions and compared the resulting summary with concrete summaries of the method in question. The results of the aforementioned experiment suggest (see Figures 8.1a to 8.1d) that even with limited sampling of method invocations for summary aggregation, the aggregated summaries were able to over-approximate the dependencies (see high recall scores in Figure 8.1) for an overwhelming majority of methods invocations across multiple executions of all eight client programs.

I also manually inspected some instances of method invocations for which precise summaries were not generated with dynamic aggregation. The anecdotal evidence of such manual investigation revealed that imprecision in modeling summaries in some cases occurs due to exceptional control flow exhibited by corner cases. For instance, consider the `get(int index)` method in `java.util.ArrayList`, whose summary shows that its return value depends on an internal data array which is a field to the `ArrayList` object, and the integer argument `index`, similar to the example showcased in Figure 4.4. However, certain invocations of the

`get(int)` method would result in an `ArrayOutOfBounds` unchecked exception, resulting in the lack of a return value and thus, exhibiting no outputs with dependencies on the method's inputs. However, such an invocation would still be modeled by a dependence summary that includes a return value as an output, resulting in an incorrect method summarization. Similarly, consider the `get(Object key)` method in `java.util.HashMap` that can either return an object, or a null value if the input `key` does not exist. While the aggregated dynamic summary for this method models both dependencies, in an actual invocation only one of the two dependencies will be exhibited: "`return` value depends on `key`", or "`return` value depends on `null`". Thus, resulting in an over-approximated dependence summary for the `HashMap.get`'s `return` value.

A similar manual analysis of the statically generated summaries revealed that for many methods the static and dynamically aggregated dependence summaries were indeed the same. However, in cases of disagreement, statically generated summaries would introduce spurious summaries due to conservative analysis of control flow. Indeed, the introduction of spurious dependence edges were particularly prominent for methods that exhibited complex control flow. For instance, the `put(Object key, Object value)` method in `java.util.HashMap` is a good example for which static analysis added a prominent number of spurious edges due to a conservative modeling of control flow. In the case of the `HashMap.put`, method static analysis would conservatively model the control path that handled the complex case of expanding the underlying key-value table, in the event that the number of entries in the key-value table was approaching its maximum threshold. In order to handle such control paths, static analysis would add multiple spurious edges with fields such as `HashMap.threshold` and `HashMap.hashseed` serving as outputs of the method `HashMap.put`. Indeed, such dependencies were observed in concrete summaries where the corresponding method invocation executed the control path to expand the underlying key-value table.

Notably, in exceptional cases dynamically aggregated dependence summaries would report

dependence relations that were not identified by static analysis. As noted in Section 8.2, when computing static dependence summaries for a method, the static analyzer does not assume any aliasing relations between the method's actual arguments; thereby resulting in potentially missed summary edges. Additionally, a closer inspection also revealed an incomplete handling of implicit data flows for field assignments in the implementation of static analysis. Consider the two successive field assignments of the form: `y = x.f; y.g = z`. Collectively, these two assignments result in the data flow from the field `x.f` into variable `y`, and from variable `z` into field `y.g`. Implicitly, such assignments should also result in the data flow from variable `z` into field `x.f.g`. While such implicit flows are handled in most cases, the static analyzer misses the modeling of such flows in limited, exceptional cases.

It is worth noting that a majority of differences between static and dynamic dependence summaries are contributed by static analysis. An analysis of the distribution of the number of dynamic-only dependence edges suggests that occurrences of dependence edges solely due to dynamic analysis are statistical outliers, and otherwise amount to zero, for the methods whose dynamic and static summaries are compared — further indicating the exceptional nature of the cases where data flows are missed by static analysis.

That said, in the majority of cases the resulting summaries did exhibit sound and precise approximations of the dynamic dependencies between inputs and outputs for specific invocations of specific methods. Such accuracy results for different sampling sizes and subject programs indicate that a method's invocation may be summarized effectively by monitoring only a sample of the method's invocations, instead of observing the totality of its invocations in a given execution. Based on these preliminary findings one can speculate that methods exhibit a limited set of behaviors that may be modeled by observing a limited number of invocations. As such, I envision practical applications where we are able to use one set of method inputs to generate method summaries, and use the resulting summaries to model behavior of method invocations that accept entirely different inputs.

As such, **to RQ2a, the investigation assesses:** Dependence summaries created using dynamically observed data resulted in the generation of sound dependence summaries within the context of a single execution, with a perceptible loss of accuracy, leading towards imprecise dependence summaries in some cases.
Further, **to RQ2b, the investigation assesses:** Dependence summaries created using static analysis of methods resulted in the generation of sound dependence summaries with clear losses in accuracy, leading towards imprecise dependence summaries in a significant number of cases.

A second line of investigation into summary accuracy compared statically created dependence summaries, and dynamically aggregated dependence summaries, and it points to the following trends:

1. Both static and aggregated dynamic summaries showed high degrees of similarity in the set of summary edges they model for various methods — as evident in the high median similarity scores between static and aggregate summaries, presented in Table 8.2 and Figure 8.4.

2. Additionally, the comparison also reveals that a perceptible degree of dependence edges are modeled solely due to static analysis of a given method, as evident in the results reported in Table 8.3 and Figure 8.4.

The agreement between static and aggregated dynamic dependence summaries suggests that the aggregation does indeed model a varying set of method behaviors, observed across multiple invocations of the methods in question. Such agreement also indicates that in many instances static analysis can accurately substitute the aggregation of dynamically observed dependencies between a method invocation's inputs and outputs. Notably, static analysis

is a cheap alternative to aggregation of dynamically observed data/control flows, given the limited one-time cost of static analysis for producing a method's summary.

However, the presence of summarized dependence edges that are unique to a method's static summary, is a further indication that methods do not necessarily exhibit all possible behaviors — as suggested by static analysis — even across multiple invocations, within program executions. Taken together, this implies that while static dependence summaries are cheap to create, they are more likely to model a wider set of dependencies between a method's inputs and outputs when compared to the method's dynamically observed data and control flows, across multiple invocations.

As such, **to RQ2c, the investigation assesses:** Static dependence summaries show high degrees of similarity with dynamically aggregated dependence summaries, while simultaneously modeling static dependence relationships that are not observed dynamically across multiple method invocations, for a perceptible number of methods.

A final study of summary accuracies studied the presence of any correlation between the a method's static call-graph size and precision of the method's static and aggregated dynamic dependence summaries, when compared to concrete dependence summaries for the method's individual invocations. This study revealed a weak correlation between the precision of a method's dependence summary and the call graph size. Such a correlation suggests that with an increasing number of other methods that a method can directly or transitively invoke, there is a weak, but increasing possibility that the method's static and aggregate summaries will over-approximate the dependencies of method's individual invocations.

As such, **to RQ3, the investigation assesses:** There exists a weak likelihood that dependence summaries — static or dynamic — are less likely to precisely model dependencies between a method's inputs and outputs, with increases in the method's call graph size.

Discussion of the Investigation in Method Behavior Similarity.

This work investigated the similarity of method behavior, as modeled by dynamic dependence summaries. Studying method behavior similarity could guide the construction of comprehensive method dependence summaries, by sampling a limited number of representative method invocations. The investigation studied the similarity of a method's behavior along three lines of study: (a) the similarity of a method's invocation within the context of an individual program execution; (b) the similarity of a method's aggregated behavior across multiple executions of the same program; and, (c) the similarity of a method's aggregated behavior across different programs.

Such an investigation points to the following broad trends:

1. To build a comprehensive dependence summary for a typical method, as invoked within a single program run, it required the analysis of at most the first eight successive invocations of the method within the program run. Further, methods required a greater number of their invocations analyzed to build comprehensive dependence summaries, it was likely because such methods had higher number of dependence relations between their inputs and outputs.

2. A method's dynamic dependence summary aggregated from one test run of a client program, is likely to be the same as a summary aggregated from a different test run of the same program.

3. A method's dynamic dependence summary aggregated from multiple test runs of a client program is likely to be similar, but with some differences, to the summary aggregated from the test runs of an entirely different program. Further, such a similarity in aggregated method summaries across client programs, tended to reduce to a minimal degree with an increase in the static call graph sizes of the methods in questions. Such a negative correlation was also observed with the number of redirections (*e.g.,* jump

or invoke instructions) that can potentially be executed within the invocations of such methods.

These trends suggest that in order to build a comprehensive, aggregated dependence summary for a given method a limited number of method invocations, early in a program run need to be analyzed. Further, a method's dependence summary built by aggregating the concrete summaries from one system-wide program run, can be reused to comprehensively model the concrete dependencies exhibited by the method in an entirely different system-wide run, of the same program. However, reusing the aggregate dependence summary for a method, created from the executions of one subject program, in the executions of entirely different subject programs may result in incomplete modeling of such method invocations.

As such, **to RQ4a, the investigation assesses:** Within a program run, invocations of a method tend to exhibit the method's range of behaviors early in the program's run, and repeat the initial set of behaviors in the remainder of the program run.
Further, **to RQ4b, the investigation assesses:** The behavior of methods – when modeled as dependence summaries – tends to be similar across test executions of the same program.
Finally, **to RQ4c, the investigation assesses:** The behavior of methods – when modeled as dependence summaries – tends to show similarities across different programs, with perceptible differences in method behavior in some cases.

Discussion of the Dynamic Slicing Case Study.

In order to investigate the cumulative effects of using dynamic dependence summaries in a real world software engineering context, the final study in this work used dynamic summaries towards improving the performance of dynamic slicing, with the goal of localizing bugs in program executions. This study also examined the accuracy of the summary-based dynamic

slicing, in the context of performance gains made as a consequence of summarization. When examining the results of this dynamic slicing case study, we find that although we are utilizing summarized results from a past execution to approximate the dependencies of external components in subsequent executions, the summarized slicing technique produced accurate results: 90% of the bugs were found with the summary-based technique.

As such, **to RQ5, the investigation assesses:** The reuse of dynamic summaries caused a small loss of accuracy as a consequence of the gains in performance.

Taken as a whole, these results suggest that the reuse of dynamic summaries can provide a way to make dynamic dependency analysis more feasible for real-world use, with modest losses in accuracy. In software-development projects that are able to tolerate inaccuracies (*e.g.,* in non-critical systems), or for analyses that are more heuristic in nature (*e.g.,* bloat analysis), the results suggest that the trade-offs favor reuse of dynamic summary information for external components.

11.1 Threats to Validity

Threats to external validity arise when the results of the experiment are unable to be generalized to other situations. The experiments in this work evaluated the impact of using dynamic and static dependency summaries on eight client programs, with their set of dependent external components. As such, I am unable to definitively state that based on the findings of this work will hold for programs in general. However, I am confident that these results are indicative of the impacts of such summarization. The external components in this study are the Java Standard Library, which is a dependency that many other programs will also have and thus the effects on efficiency and effectiveness of dependence summarization for the methods in that common library is likely to hold for those programs. Moreover, the

significant gains exhibited in the empirical results gives strong evidence that the approach presented in this work, at least, shows promise for use in practice.

Threats to construct validity arise when the metrics used for evaluation do not accurately capture the concepts that they are meant to evaluate. The experiments measured the costs involved in performing tracing and dynamic dependence analysis in terms of computational time and data storage. Although the results give an indication of the degree of such costs, the implementation used in this work can be greatly optimized in both regards. The tracing information is verbose and its implementation is not optimized. However, this limitation does not affect the overall result, as this same implementation was used for both treatment techniques (exhaustive and summarized); *i.e.,* the direction and magnitude of the difference between the results should not significantly change when these factors are optimized. Also, the experiments measured the inaccuracy introduced in method summaries as a process of abstraction and aggregation; and due to the conservative modeling of dependencies with static analysis. Although these metrics give an indication of the accuracy of the results, they do not give a sense for how these will affect either developer time or client analyses that build upon such results. Further studies will need to be conducted to assess the impacts on such clients of these analyses.

Chapter 12

Related Work

Two main bodies of prior work are related to the work proposed in this document: summary based program analysis, and dynamic dependence analysis.

Summary-Based Program Analysis. Procedural summaries have been computed and used widely in the static analysis community to achieve modularity and efficiency. Summary functions for interprocedural analysis date back to the functional approach in the work by Sharir & Pnueli (1981), with refinements for Interprocedural, finite, distributive, subset (IFDS) problems from Reps *et al.* (1995) and for interprocedural distributive environment (IDE) dataflow problems from Sagiv *et al.* (1996).

Yorsh *et al.* (2008) use conditional micro-transformers to represent and manipulate dataflow functions. Rountev and colleagues (Rountev *et al.* (2006, 2008)) propose a graph representation of dataflow summary functions, that generate and use summaries, to reduce the cost of whole-program IDE problems. Xu *et al.* (2009) propose a summary-based analysis that computes access-path-based summaries to speed up the context-free-language (CFL)-reachability-based points-to analysis. Dillig *et al.* (2011) propose a summary-based heap analysis targeted for program verification that performs strong updates to heap locations

at call sites. Ranganath & Hatcliff (2004) statically analyze the reachability of heap objects from a single thread, for slicing concurrent Java programs. Salcianu & Rinard (2005) propose a regular-expression based, purity and side effect analysis for Java, to characterize the externally visible heap locations that a given method mutates. Works in static taint analysis by Bastani *et al.* (2015); Arzt *et al.* (2014) use static analysis frameworks such as CFL reachability and the IFDS frameowrk to mine method specifications to support static taint analysis, with the aim of identifying security vulnerabilities in software systems. Tang *et al.* (2015) propose a static analysis based method to summarize library methods specifically in the presence of callbacks in the library methods. This is similar in spirit to the heap location-based dependence summarization as proposed in this work. Inspired by such static analyses, the work presented in this paper, is the first technique to compute and use dynamic dependence summaries. Dynamic dependence summaries, can potentially better inform dependence information more precisely than static dependence summaries, by leveraging information collected at runtime.

Other approaches have looked to use dynamic analysis to create models of method behaviors to support various kinds of static analysis. Works by Jha *et al.* (2010); Qi *et al.* (2012); Biermann *et al.* (1975); Heule *et al.* (2015) look to synthesize simplified code fragments that model the behavior of real world software components. Such code fragments essentially are presented as summarized models to other static analyses such symbolic execution engines. Indeed such works vary in the manner in which they dynamically analyze program runs to synthesize simplified component models. For instance, Qi *et al.* (2012) observe output values for actual component inputs to correlate such inputs and outputs to generate component models. Other works such as that by Heule *et al.* (2015) actually monitor execution traces to infer program behavior and synthesize component models. Unlike such works that use dynamic data to support static analysis, the ideas presented in this dissertation looks to use abstract, and static models of program behavior to create component models that can be applied in a dynamic analysis. Moreover, the component models presented in this work

does not synthesis code, but instead models dependencies between inputs and outputs for a method.

Dynamic Dependence Analysis. Since first being proposed by Agrawal & Horgan (1990) dynamic dependence analysis has inspired a large body of work on a variety of software engineering tasks from debugging to memory bloat analysis. Early work from Kamkar *et al.* (1992) introduces the theory behind summary-based dependence slicing for a stack-only language. This work attempts to be more general; in that it handles all features of a modern object-oriented language. Moreover, this work introduces the notion of reusing method summaries — created from one invocation — to model entirely different invocations of the same method.

The work by Zhang and colleagues (Zhang *et al.* (2003); Zhang & Gupta (2004a,b); Zhang *et al.* (2006b,a)) considerably improved the state of the art in dynamic dependence analyis and its applications like dynamic slicing and fault localization. This work includes, for example, a set of cost-effective dynamic slicing algorithms Zhang *et al.* (2003); Zhang & Gupta (2004a), a slice-pruning analysis that computes confidence values for instructions to select those that are most related to errors Zhang *et al.* (2006b), a technique that performs on-line compression of the execution trace Zhang & Gupta (2004b), and an event-based approach that reduces the cost by focusing on *events* instead of individual instruction instances Zhang *et al.* (2006a). Apart from Zhang's work, Wang & Roychoudhury (2004) develop optimizations to compress bytecode execution traces for sequential Java programs resulting in space efficiency. Moreover, they develop a slicing algorithm, applicable directly on the compressed bytecode traces. Hierarchical Dynamic Slicing Wang & Roychoudhury (2007) aims at guiding the programmer through large and complex dependence chains in a dynamic slice, with debugging as the primary application.

Such works look to attain cost savings with an effort to improve the performance of dy-

namic slicing. As such, improvements in the performance of dynamic slicing largely resides in traversing the underlying dynamic dependence graph efficiently — which does not imply reducing the cost of dynamically monitoring and performing dependence analysis itself. Contrarily, this work looks to use summarization to reduce the costs of dependence analysis itself. And such cost savings are attained in a general-purpose manner that may bring down the costs of any client analysis that relies on dynamic dependence analysis, not just dynamic slicing.

Deng & Jones (2012) propose a dynamic dependence graph that encodes frequency of inferred traversal in order to prioritize heavily trafficked flows. Xu *et al.* (2010) propose an abstraction-based approach, to scale a class of dynamic analyses that need the backward traversal of execution traces. This approach employs user-provided analysis-specific information to define equivalence classes over instruction instances, so that dynamic slicing can be performed over bounded abstract domains instead of concrete instruction instances, leading to space and time reduction. This work achieves efficiency from a different angle—this work uses summaries generated for library classes to speed up general dynamic data dependence analysis, and thus, all dynamic analyses that need dependence information may benefit from this technique.

Chapter 13

Summary

This work presents a summary-based approach to effectively perform dynamic dependence analysis for modern applications that rely on object-oriented libraries and components. Traditional approaches to dynamic dependence analysis are typically whole execution analyses. Whole execution analysis for a component necessitates a simultaneous and exhaustive analysis of both: (a) the component, and (b) any of the component's underlying dependencies. In other words, a whole execution approach analyzes a set of inter-dependent components, as one monolith. As such, whole execution analyses tend to exact high computational costs for systems with multiple inter-dependent components.

A consequence of a summary-based approach is the decoupling of dynamic analysis of inter-dependent components, unlike in whole execution analyses. Each executed component can potentially be analyzed independently, with the results encoded as summaries. Summarized analysis results of such components can be reused for future invocations of the components — without re-analyzing the components — thus reducing computational costs. Such reuse of analysis results can then be combined, in order to analyze the execution of the inter-dependent components as a whole.

Specifically in this work, designated library methods are analyzed and their analysis results are encoded as dependence summaries. Such summaries are later used for dependence analysis, instead of exhaustively re-analyzing the library methods, thus improving analysis efficiency. As discussed earlier, such summaries may be produced either through static analysis, or with the aggregation of the dynamically observed data and control flows within the components being summarized.

Particularly while computing dynamic dependence summaries for a library method, the approach first extracts concrete dependence relations, between a method invocation's inputs and outputs from its execution trace, and then abstracts such relations with symbolic data. Such abstracted dependence summaries resemble statically generated dependence summaries, and enable the modeling of future method invocations with a substitution of symbolic data with invocation specific concrete data.

Any form of information abstraction is likely to lead to a loss in accuracy. As such, this work investigates the extent of such accuracy losses, when balanced with performance gains. The empirical investigation was specifically guided by the following thesis statement (restated from Chapter 1).

> **Thesis:** Reusable dependence summaries for method invocations can reduce the computational costs involved in data- and control-flow based dynamic dependence analysis of software runs, while modeling such dynamic dependencies with moderate-to-high degrees of accuracy.

In summary, the empirical evidence suggests that real-world software are likely to exhibit a heavy reliance on external components and reusing such dependence summaries in dependence analysis can significantly save costs. The investigation further shows that using summaries — created statically, or aggregated dynamically — tends to effectively capture varying method behaviors despite potentially causing perceptible losses in accuracy.

The investigation confirmed a trade-off between the performance gains and accuracy losses due to the usage of summarized dependencies for dynamic analysis. Moreover, the investigation also revealed a trade-off between statically generated summaries and dynamically aggregated summaries, given a notable degree to similarity between the dependence relations between discovered by the static and dynamic aggregation approaches.

This work on summarized approach to analysis, employs a divide and conquer approach to dynamic analysis, instead of a monolithic whole execution approach. Specifically, this work divides dynamic dependence analysis along the lines of application and library code, and summarizes the library code. Such a division of dynamic analysis enabled a controlled, systematic investigation of summarizing methods from a common library, and the subsequent effects of such summarization across multiple complex, real-world, subject programs.

Indeed, a summary based approach can divide and conquer dynamic dependence analysis in potentially different ways. In other words, the question of, "what to summarize?" can be answered differently based on how or why a summarized analysis is used. For instance, such divisions in dynamic analysis can potentially be influenced by a need for greater degrees of accuracy or performance. The needs for different degrees of accuracy and performance may be guided by a specific software engineering task, which uses a summarized dynamic dependence analysis. The following section on future work, discusses such potentially varied applications of a summary based dynamic analysis.

13.1 Future Work

Summarizing for Specific Client Analyses. An important, and perhaps the next direction of research would be to extensively use and evaluate the impact of the summary based analysis on the accuracy of various program analysis and software engineering tech-

niques. As noted in Chapter 1, dynamic dependence analysis is a well studied technique that has impacted a variety of software engineering and program analysis research areas (*e.g.,* dynamic slicing, memory bloat analysis, information flow analysis, and change impact analysis). I envision the following directions of research based on summary-based approaches for the following client analyses that use dependence analysis: memory bloat analysis to efficiently identify memory performance issues in application runs; information flow analysis to efficiently identify security vulnerabilities in critical code paths; and summarized dynamic slicing. Summarized dynamic slicing by itself can be used towards various software engineering techniques, such as: fault localization, change impact analysis, interactive debugging for applications like WHYLINE as proposed by Ko & Myers (2008).

Indeed, Chapter 10 presents a study into the impact of summarized dependencies on dynamic slicing. The study in Chapter 10 shows that there were indeed gains in performance, while limited losses in accuracy. Further, such results were congruent with the results of the empirical investigations into the accuracies of the summarized dependencies themselves.

That said, each downstream software engineering or program analysis technique is different in terms of how it uses the results of dynamic dependence analysis and how it presents the technique's final results. As such, dependence summaries may impact the accuracy of downstream analyses to different degrees.

For instance, change impact analysis presents its results in the form of instructions that are likely affected by changes in some other instructions in the code base. Such an analysis indirectly derives such instruction dependencies from dependencies between concrete memory locations. As such, the dependencies between a method's input and output memory locations may have a limited, and indirect, impact on the final results for change impact analysis. Contrarily, memory bloat analysis uses dependencies between memory locations in order to identify superfluous, or rarely used memory locations in an application's runtime. Such an analysis identifies application memory that is symptomatic of performance degradations. As

such memory bloat analysis may perhaps be more directly, and severely, affected by imprecise dependencies between a method's input and output memory locations.

Given such nuances in how dependence analysis is used from one client analysis to another, studying the impact of a summary based approach, on each client analysis warrants its own separate research agenda. The key question in each such an agenda would be guided by a similar question around the trade-off between accuracy and performance, as raised by this work for method dependence summaries. However, the metrics to measure accuracy and performance will vary across different client analyses.

Adequacy Criteria and Suitability for Summarization. An important direction for future work could develop ways to assess suitability of summarization for methods. Such work would specifically develop adequacy criteria to inform when the training phase has sufficiently exercised all behaviors of a method for summarization. Such criteria could also indicate an unusually high degree of varying behaviors exhibited by a method, thus suggesting the avoidance of summarized analysis for a method. In other words, adequacy criteria may inform the degree of varying behaviors in a method, and the degree to which such varying behaviors are summarized.

The investigations in this work examined relationships between the decreasing accuracy of method dependence summaries, and the size of the method's call graph, as one example of adequacy criteria (see Sections 8.3 and 9.3). Such investigations revealed weak relations between the increasing size of a method's call graph, and the decreasing accuracy of the method's summary. Additional metrics such as code coverage, *e.g.,* line and branch coverage, might serve as potentially better indicators of summary accuracy, and thus, suitability for summarization. As such, a further investigation of such adequacy criteria warrant study.

Such work is particularly important given that a summary-based dependence analysis can introduce both unsoundness and imprecision. On one hand, the quality of an aggregated

summary relies on the coverage of the test runs used to train the summary. Thus, a summary may miss certain dependence relationships due to the lack of test cases. On the other hand, both the summary aggregation from multiple method invocations, and static analysis of the method code may introduce spurious dependence relations that would not have been added in a regular dependence analysis.

For instance, to reduce such negative effects, the adequacy criteria may indicate either inadequate summarization, or high degree of variance in method behavior; thereby prompting the avoidance of summarization for specific methods.

Reusing Summaries for Unobserved Argument Types. This work records dynamically-observed argument types with a method's dynamic dependence summary, to associate summarized dependence edges with specific argument types. When a future invocation of the method uses a previously recorded set of dynamically-observed argument types, then the corresponding set of summarized dependence edges can be used to model the invocation in question. As such, method summaries can only be used for invocations where the dynamically-observable argument types were previously recorded. This work does not prescribe an approach to reuse a method's summary when the method is invoked with a set of argument types that were not already recorded.

Future work can investigate approaches to reuse a method's summary even when the method is invoked with argument types that are not recorded as a part of the method's dependence summary. For instance, this can be done in the case where the method has a summary for a given set of argument types, but is invoked with subtypes. The reuse of such "supertyped" summaries will be imperfect. However, using a lightweight analysis of the fields that the subtypes inherit from the supertypes can enable inferences about the "supertyped" summary edges that may also be valid in the case of a "subtyped" summary. Alternatively, future approaches could also consider using static analysis for methods in the event that a method is

invoked with a set of dynamically-observed argument types that were never before recorded.

Summarizing Multiple Array Access and Loop Iterations. A critical direction for future work could provide yet further improvement to array summarization by developing ways to summarize access to multiple array indices. I envision techniques that rely on light-weight instrumentation to accurately identify accesses to array elements.

Although the technique in this work improves upon existing work in distinguishing array index accesses, this improvement is limited to method calls that access only a single index. The results of the studies revealed that a majority of the studied methods accessed a single index. Even for methods that access multiple array locations (such as `addAll` in many of the `java.util.*` methods), they are often implemented by invoking methods that access a single array location (such as `add`) multiple times. Hence, the approach in this work can precisely handle array accesses for most of the common and frequently-used collection methods.

Given that multiple array locations are often accessed within loop bodies, a related direction of research could be to produce dependence summaries for all iterations of a loop body, instead of an entire method body. A loop body could potentially be summarized by analyzing only a limited sample of the loop's iterations, instead of all iterations, just as with building dynamic dependence summaries for methods. The analysis of a sample of a loop's iterations can help determine the inputs and outputs of a loop body (*i.e.,* memory locations that are used and defined), and the dependence relations between them. Such loop specific summaries can inform how the results of one iteration are used in the loop's subsequent iteration.

Dependence Summarization and Other Hybrid Approaches. Dependence summarization is indeed a hybrid approach that combines static or abstract dependencies with dynamic dependence analysis. The reuse of dependence summaries completely avoids the analysis of components for which summaries exist. However hybrid approaches often use static analysis in other ways to bring down the costs of dynamic dependence analysis.

For instance, works by Zhang & Gupta (2004a); Tallam & Gupta (2007) looks to use static analysis to reduce the amount of dynamic data that needs to be collected. Such approaches statically compute data and control dependencies for all potential control paths, and use lightweight dynamic analysis to monitor the executed control paths. Such a lightweight dynamic approach allows the selection of appropriate data/control dependencies that were likely induced along the executed control paths. Notably, such approaches would still necessitate the monitoring of all components in an inter-dependent set of executing components. However, the dependence analysis results from such a lightweight dynamic approach, can still be used to build dependence summaries for a given set of components.

Specifically, future works could use lightweight dynamic analysis to account for varying behavior due to dynamically observed control flow, accesses to individual array elements and dynamic dispatch. The remainder of the dependence analysis can be performed with static analysis. The resulting hybrid dependence analysis can further be summarized, using the summary based approach discussed in this work. The summarization of such dependence analysis can in turn further bring down the costs of the lightweight dynamic analysis.

Further Extensions to Current Evaluation. Additionally, an important extension of this work would be the further expansion of the current investigations on a larger set of client applications, and libraries. Such an expansion would essentially include all facets of the current investigation: (a) creation and comparison of static and aggregate summaries with concrete summaries; (b) measuring the impact of summarization on the performance cost savings; and (c) evaluating the similarity of method behavior across multiple method invocations.

Such extensions would essentially replicate the current experiments of this work, and either confirm or reject the larger conclusions of this work, thereby extending greater confidence and generalizability to the results of this work.

13.2 Contributions

In its entirety, this dissertation delineates the background, motivations, theory, and results of an empirical evaluation that collectively support the central thesis of this work, as presented earlier in this chapter and original stated in Chapter 1. In summary, the following are the concrete contributions of this work:

1. A conceptual framework, as a set of enumerated challenges, definitions, and algorithms, that provide the necessary formalisms for the creation and usage of reusable dependence summaries for dynamic dependence analysis.

2. An extensive instrumentation framework for exhaustive, fine-grained software execution profiling for software systems developed in Java.

3. A comprehensive program analysis framework to perform static and dynamic dependence analysis, in a comparative setting, for software systems developed in Java and executed atop the Java Virtual Machine, with support for creating and using dependence summaries.

4. Experiments and empirical results to:

   (a) provide evidence of the performance gains as a result of using dynamic dependence summaries;

   (b) assess the extent of accuracy losses when using (dynamic and static) dependence summaries for library methods;

   (c) study the nature and extent of accuracy losses when using (dynamic and static) dependence summaries for library methods with regard to their position in their call graphs;

   (d) assess the extent of variability in method behavior, when modeled as dependence summaries, within individual software runs;

(e) assess the similarity in method behavior, when modeled as dependence summaries, across multiple execution of a software system;

(f) assess the similarity in method behavior, when modeled as dependence summaries, across different software systems; and,

(g) qualitatively assess the applicability of using reusable dependence summaries with dynamic dependence analysis, in the real software engineering technique of dynamic slicing.

Bibliography

Agrawal, Hiralal, & Horgan, Joseph R. 1990. Dynamic program slicing. *Pages 246–256 of: PLDI.*

Arzt, Steven, & Bodden, Eric. 2016. StubDroid: Automatic inference of precise data-flow summaries for the Android framework. *Pages 725–735 of: Proceedings of the 38th International Conference on Software Engineering.* ACM.

Arzt, Steven, Rasthofer, Siegfried, Fritz, Christian, Bodden, Eric, Bartel, Alexandre, Klein, Jacques, Le Traon, Yves, Octeau, Damien, & McDaniel, Patrick. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, **49**(6), 259–269.

Bastani, Osbert, Anand, Saswat, & Aiken, Alex. 2015. Specification inference using context-free language reachability. *Pages 553–566 of: ACM SIGPLAN Notices*, vol. 50. ACM.

Biermann, Alan W., Baum, Richard I., & Petry, Frederick E. 1975. Speeding up the synthesis of programs from traces. *IEEE Transactions on Computers*, **100**(2), 122–136.

Blackburn, S. M., & *et al.* 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *Pages 169–190 of: OOPSLA.*

Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., & Wiedermann, B. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *Pages 169–190 of: OOPSLA.*

Bruneton, E., Lenglet, R., & Coupaye, T. 2002 (November). ASM: A code manipulation tool to implement adaptable systems. *In: Adaptable and Extensible Component Systems.*

Chatterjee, Ramkrishna, Ryder, Barbara G, & Landi, William A. 1999. Relevant context inference. *Pages 133–146 of: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM.

Deng, Fang, & Jones, James A. 2012. Weighted System Dependence Graph. *Pages 380–389 of: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on.* IEEE.

Dillig, Isil, Dillig, Thomas, Aiken, Alex, & Sagiv, Mooly. 2011. Precise and compact modular procedure summaries for heap manipulating programs. *Pages 567–577 of: PLDI.*

Do, Hyunsook, Elbaum, Sebastian, & Rothermel, Gregg. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, **10**, 405–435.

Heule, Stefan, Sridharan, Manu, & Chandra, Satish. 2015. Mimic: Computing models for opaque code. *Pages 710–720 of: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* ACM.

Holewinski, Justin, Ramamurthi, Ragavendar, Ravishankan, Mahesh, Fauzia, Naznin, Pouchet, Louis-Noel, Rountev, Atanas, & Sadayappan, P. 2012. Dynamic Trace-Based Analysis of Vectorization Potential of Applications. *Pages 371–382 of: PLDI.*

Horwitz, Susan, Reps, Thomas, & Binkley, David. 1990. Interprocedural slicing using dependence graphs. *TOPLAS*, **12**(1), 26–60.

Jha, Susmit, Gulwani, Sumit, Seshia, Sanjit A, & Tiwari, Ashish. 2010. Oracle-guided component-based program synthesis. *Pages 215–224 of: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1.* ACM.

Kamkar, Mariam, Shahmehri, Nahid, & Fritzson, Peter. 1992. Interprocedural dynamic slicing. *Pages 370–384 of: Programming Language Implementation and Logic Programming*, vol. 631.

Ko, Andrew Jensen, & Myers, Brad A. 2008. Debugging reinvented. *Pages 301–310 of: Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on.* IEEE.

Korel, Bogdan, & Laski, Janusz. 1990. Dynamic slicing of computer programs. *Journal of Systems and Software*, **13**(3), 187–195.

McIlroy, Doug. 1968. Mass-Produced Software Components. *Pages 88–98 of: Proceedings of the NATO Conference on Software Engineering.*

Newsome, James, & Song, Dawn. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *In: NDSS.*

Orso, Alessandro, Harrold, Mary Jean, & Rosenblum, David S. 2001. Component Metadata for Software Engineering Tasks. *Pages 129–144 of: International Workshop on Engineering Distributed Objects.*

Orso, Alessandro, Do, Hyunsook, Rothermel, Gregg, Harrold, Mary Jean, & Rosenblum, David S. 2007. Using component metadata to regression test component-based software: Research Articles. *Journal of Software Testing, Verification and Reliability*, **17**(2), 61–94.

Qi, Dawei, Sumner, William N, Qin, Feng, Zheng, Mai, Zhang, Xiangyu, & Roychoudhury, Abhik. 2012. Modeling software execution environment. *Pages 415–424 of: Reverse Engineering (WCRE), 2012 19th Working Conference on.* IEEE.

Ranganath, Venkatesh Prasad, & Hatcliff, John. 2004. Pruning interference and ready dependence for slicing concurrent java programs. *Pages 39–56 of: Compiler Construction*. Springer.

Reps, T., Horwitz, S., & Sagiv, M. 1995. Precise Interprocedural Dataflow Analysis Via Graph Reachability. *Pages 49–61 of: POPL*.

Rountev, Atanas, Kagan, Scott, & Marlowe, Thomas. 2006. Interprocedural Dataflow Analysis in the Presence of Large Libraries. *Pages 2–16 of: International Conference on Compiler Construction*. LNCS 3923.

Rountev, Atanas, Sharp, Mariana, & Xu, Guoqing. 2008. IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. *Pages 53–68 of: CC*. LNCS 4959.

Sagiv, Mooly, Reps, Thomas, & Horwitz, Susan. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, **167**(1-2), 131–170.

Salcianu, Alexandru, & Rinard, Martin. 2005. Purity and Side Effect Analysis for Java Programs. *Pages 199–215 of: VMCAI*.

Sharir, M., & Pnueli, A. 1981. Two Approaches to Interprocedural Data Flow Analysis. *Pages 189–234 of:* Muchnick, S., & Jones, N. (eds), *Program Flow Analysis: Theory and Applications*. Prentice Hall.

Tallam, Sriraman, & Gupta, Rajiv. 2007. Unified control flow and data dependence traces. *ACM Transactions on Architecture and Code Optimization (TACO)*, **4**(3), 19.

Tang, Hao, Wang, Xiaoyin, Zhang, Lingming, Xie, Bing, Zhang, Lu, & Mei, Hong. 2015. Summary-based context-sensitive data-dependence analysis in presence of callbacks. *Pages 83–95 of: ACM SIGPLAN Notices*, vol. 50. ACM.

Vallée-Rai, Raja, Co, Phong, Gagnon, Etienne, Hendren, Laurie, Lam, Patrick, & Sundaresan, Vijay. 1999. Soot-a Java bytecode optimization framework. *Page 13 of: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press.

Wang, Tao, & Roychoudhury, Abhik. 2004. Using compressed bytecode traces for slicing Java programs. *Pages 512–521 of: ICSE*.

Wang, Tao, & Roychoudhury, Abhik. 2007. Hierarchical dynamic slicing. *Pages 228–238 of: Proceedings of the 2007 international symposium on Software testing and analysis*. ISSTA '07.

Weiser, Mark. 1981. Program slicing. *Pages 439–449 of: Proceedings of the 5th international conference on Software engineering*. IEEE Press.

Xiao, Xiao, & Zhang, Charles. 2011. Geometric encoding: forging the high performance context sensitive points-to analysis for Java. *Pages 188–198 of: Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM.

Xu, Guoqing, Rountev, Atanas, & Sridharan, Manu. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. *Pages 98–122 of: ECOOP.*

Xu, Guoqing, Arnold, Matthew, Mitchell, Nick, Rountev, Atanas, Schonberg, Edith, & Sevitsky, Gary. 2010. Finding Low-Utility Data Structures. *Pages 174–186 of: PLDI.*

Yorsh, Greta, Yahav, Eran, & Chandra, Satish. 2008. Generating precise and concise procedure summaries. *Pages 221–234 of: POPL.*

Zhang, Xiangyu, & Gupta, Rajiv. 2004a. Cost Effective Dynamic Program Slicing. *Pages 94–106 of: PLDI.*

Zhang, Xiangyu, & Gupta, Rajiv. 2004b. Whole Execution Traces. *Pages 105–116 of: MICRO.*

Zhang, Xiangyu, Gupta, Rajiv, & Zhang, Youtao. 2003. Precise dynamic slicing algorithms. *Pages 319–329 of: ICSE.*

Zhang, Xiangyu, Tallam, S., & Gupta, Rajiv. 2006a. Dynamic Slicing Long Running Programs through Execution Fast Forwarding. *Pages 81–91 of: FSE.*

Zhang, Xiangyu, Gupta, Neelam, & Gupta, Rajiv. 2006b. Pruning dynamic slices with confidence. *Pages 169–180 of: PLDI.*