

A DYNAMIC APPLICATION ANALYSIS FRAMEWORK

BY

MARIE THERESE CONTE

B.E.E., University of Delaware, 1995

M.S., University of Illinois, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

© Copyright by Marie Therese Conte, 2003

A DYNAMIC APPLICATION ANALYSIS FRAMEWORK

Marie Therese Conte, Ph.D.

Department of Electrical Engineering

University of Illinois at Urbana-Champaign, 2003

Wen-mei W. Hwu, Adviser

In this thesis we address the problem of interprocedural analysis on a dynamic application. We present a framework for performing partial analysis ahead of time and using it to facilitate a large range of runtime analyses and optimizations. We demonstrate one such analysis by performing swift, safe analysis during profiling of threaded, dynamically linked, adaptively compiled applications. In our framework, we focus on one such language, Java; however, our techniques are adaptable to others within this realm. We also present models for adaptive compilation utilizing our framework to verify compilation assumptions in the event of dynamic class loading. We present our system for performing a subset of analyses ahead of time by constructing a graph called a *Compact Dataflow Graph* (CDG), of the object references used intraprocedurally. The CDG is designed to be independent of the internal representation used by the runtime and general enough to facilitate a large range of dynamic interprocedural analysis and optimizations. We present our design and implementation of one such use of the CDG by using it to swiftly construct a form of a unification points-to graph we call an *Object Connection Graph* (OCG), which is used to determine swiftly a set of method local allocations that could be safely stack allocated. We present results for the use of the OCG using a subset of the threaded Java Grande benchmarks, and a set of small Java threaded applications.

This thesis is dedicated to my mother, Anne M. Conte, whose encouragement, love, and support helped me achieve this milestone but who died on March 5, 2003, just a few months before its completion.

ACKNOWLEDGMENTS

I would like to thank my adviser, Wen-mei Hwu, whose patience, training, and support over the past eight years helped me to grow both academically and personally. In addition I wish to thank the other members of my thesis committee, Carol Thompson, Steve Lumetta, and William Sanders, who provided invaluable editorial feedback on this thesis. I would also like to thank my brother, Thomas M. Conte, who encouraged me to stick it out on numerous occasions when the road to completing this degree got a little hard to tread. Additionally I would like to thank John Gyllenhaal who was my mentor when I started with the IMPACT research group. He helped me navigate the transition to graduate school, teaching me to balance the class work and the research work to succeed at both. Furthermore, I would like to acknowledge the contributions of Hong-Soek Kim, who helped me formalize the presentation of my research. Finally, I would like to thank the members of the IMPACT research group for their encouragement, support, and help over the past eight years. They provided invaluable feedback on numerous occasions that helped me focus my research, helping me develop and define my final thesis topic.

TABLE OF CONTENTS

CHAPTER	PAGE
1 INTRODUCTION	1
1.1 Object Oriented Languages and Java	1
1.2 Analysis and Optimization	4
1.3 Overview of Our System	6
1.3.1 Runtime architecture phases	7
1.3.2 First mode analysis	10
1.3.3 Second mode analysis	11
1.3.4 Consumption of analysis results	12
1.4 Primary Contributions	13
2 INTRAPROCEDURAL ANALYSIS	17
2.1 Overview	17
2.1.1 The Compact Dataflow Graph	20
2.2 CDG Formation	26
2.3 File Annotations	29
3 DYNAMIC ANALYSIS: INTERPROCEDURAL ANALYSIS	34
3.1 Class Hierarchy Representations	35
3.2 Adaptive Call Graphs	36
3.3 Analysis Information	39
3.4 Optimizations Enabled	43
3.4.1 Stack allocation of objects	44
3.4.2 Synchronization removal	50
3.4.3 Caching of redundant callee results and call elimination	54
3.4.4 Lifetime-based optimizations	66
3.4.5 Memory layout for better data locality	70
3.4.6 Code motion after inlining	72
3.4.7 Unsafe sharing, potential race determination	72
4 SPECIFIC APPLICATION: OBJECT CONNECTION GRAPHS	74
4.1 Problem Description	74
4.2 Our Solution	77
4.2.1 Definition	77
4.3 Extracting Points-To Information	78
4.4 Interprocedural Propagation	82
4.5 Runtime Structures	89

4.6	Experimental Setup and Results	95
4.6.1	Experimental results: potential benefits	99
4.6.2	Experimental results: estimated costs	102
4.7	Conclusion on iOCG	106
5	DYNAMIC OPTIMIZATION VALIDATION AND ROLLBACK	107
5.1	Optimization Models and Validation	107
5.1.1	Always safe	109
5.1.2	Sometimes safe	115
5.1.3	Speculatively safe	116
5.1.4	Mixing optimization models	117
5.2	Validation Failure, Rollback, and Recovery	120
5.2.1	Preregion execution state	123
5.2.2	In-region execution state	126
5.2.2.1	Approach 1: continue executing	126
5.2.2.2	Approach 2: checkpoint, rollback, and re-execute	129
5.3	Additional Optimization Examples	133
5.3.1	Stack allocation of object instances	133
5.3.1.1	Flushing of stack allocated object instances	135
5.3.2	Synchronization removal	137
5.3.3	Code motion after inlining	141
5.4	Conclusions	141
6	RELATED WORK	143
6.1	Static Analysis	143
6.1.1	Class hierarchy and call graphs	144
6.1.2	Interprocedural propagation	148
6.2	Java Specific Research	150
6.2.1	Escape analysis	150
6.2.2	Static stability, “leaf” class/procedure determination	155
6.2.3	Addressing dynamic applications	156
6.2.4	Enabling aggressive optimization through annotations	161
6.2.5	Validation	162
6.2.6	Java runtime designs	163
6.3	Other Forms of Object Oriented Optimizations	165
7	CONCLUSIONS AND FUTURE WORK	169
7.1	Context-Specific Analysis Result Retention	170
7.2	Quantifying the Validation Framework	173
7.3	Accomidating Additional Java Features	173
7.3.1	Interfaces	174
7.3.2	Interprocedural exception tracking	174
7.4	Additional Validation and Verification Uses	175
7.4.1	Race detection	175

7.5	Extrapolation into Other Languages	176
7.5.1	Eliminated boundary conditions	177
7.6	Exploiting Hardware Specific Features	178
APPENDIX A BYTECODE LEVEL ALGORITHM AND ANNOTATIONS . .		179
A.1	Intraprocedural Algorithm for Forming CDG	180
A.1.1	Breaking BBs for exceptions and subroutines	180
A.1.2	Reducing anyalysis bytecode and assigning arguments	187
A.1.3	Forming the intermediate graph	190
A.1.4	Iterative, backwards, dataflow algorithm	206
A.1.5	Connecting defines and usages	211
A.2	Annotations	213
APPENDIX B SIMULATOR ARCHITECTURE		220
B.1	Simulated Execution Engine	221
B.2	Thread States	225
B.3	Method Invoker	229
B.4	Memory Manager	232
REFERENCES		235
VITA		241

LIST OF TABLES

Table	Page
2.1 Definition of entries in a Dataflow graph.	20
2.2 Rules to add edges in initial DGs. The <i>l</i> 's represent line numbers.	24
2.3 Rules to add edges to the initial graph to form the extended graph.	25
2.4 Percentage of actual methods used that were single basic block.	32
3.1 Analysis needed based on optimization.	44
4.1 Description of benchmarks and applications used.	96
4.2 The actual number of dynamic bytecode instructions executed at the time of GC.	104
A.1 Exception scoping example.	181
A.2 Potential implicit runtime exceptions and errors thrown by specific bytecode instructions.	183
A.3 Reference affect events and their bytecode instructions.	188
A.4 Percentage of dynamic instructions containing the bytecode instructions in Table A.3.	189
A.5 Definition of the intraprocedural analysis graph.	191
A.6 Rules for adding nodes and edges based on RAE entries, part a.	196
A.7 Rules for adding nodes and edges based on RAE entries, part b.	197
A.8 Property nodes divided into "defines" and "uses."	210
B.1 Sample line showing fields in the invocation line from a trace file.	230

LIST OF FIGURES

Figure	Page
1.1 An overview of the Dynamic Application Analysis Framework.	7
1.2 A conceptual view of the phases during the lifetime of an application.	8
2.1 A conceptual view of a procedure.	18
2.2 A conceptual view of intraprocedural analysis.	19
2.3 Example for illustrating construction of the CDGs.	26
2.4 CDG construction steps for method <code>Bar</code>	27
2.5 CDG construction for <code><Clazz></code> in Figure 2.3.	29
2.6 CDG construction for <code>Hoe</code> in Figure 2.3.	29
2.7 CDG construction for <code>Foo</code> in Figure 2.3.	30
2.8 The percentage of unique methods in each benchmark that are from the benchmark versus library class files.	31
3.1 Example subclasses for the class in Figure 2.3.	37
3.2 ACGs for the three potential types of <code>Clazz</code> <code>o</code> in Figure 2.3.	37
3.3 Final abstract source-level CDGs for the methods in Figures 2.3 and 3.1. . .	40
3.4 Results from interprocedural propagation of the three types of <code>Clazz</code> in Figure 2.3.	41
3.5 Results from removing the <i>super nodes</i> from the iCDGs in Figure 3.4.	49
3.6 Synchronized version of the class file from Figure 2.3.	53
3.7 CDGs for the class file shown in Figure 3.6.	54
3.8 The interprocedural propagation result for the class in Figure 3.6.	55
3.9 Example list class.	56
3.10 User class for the list class shown in Figure 3.9.	57
3.11 CDG formation from the class files in Figures 3.9 and 3.10.	59
3.12 Initial ACG and corresponding iCDG for <code>ListFlattener</code>	60
3.13 ACG and corresponding iCDG after resolving <code>ListCopier</code> and <code>append</code>	60
3.14 ACG and corresponding iCDG after resolving <code>getElement</code> and <code>add</code>	61
3.15 ACG and corresponding iCDG after resolving the next tier <code>getElement</code> and <code>add</code>	62
3.16 ACG and corresponding iCDG after resolving the next tier <code>getElement</code> and <code>add</code>	63
3.17 Final ACG and corresponding iCDG.	64
3.18 Final iCDG with super nodes removed.	65
3.19 Final iCDG with field <code>s</code> reads removed.	66

3.20	Another version of the class <code>ListClazzUser</code> that uses the classes <code>ListClazz</code> and <code>ClazzNode</code> shown in Figure 3.9.	68
3.21	The ACG for the procedure <code>FooList</code> shown in Figure 3.20.	68
3.22	The CDG for the procedure <code>FooList</code> shown in Figure 3.20.	69
3.23	The resulting iCDG from the interprocedural propagation in <code>FooList</code>	70
3.24	The resulting iCDG after <i>super node</i> removal.	71
4.1	Reduced CDG example for the method <code>Bar</code>	78
4.2	OCG propagation through the ACG for the type resolution <code>ClazzB</code>	80
4.3	Definition of the Object Connection Graph.	81
4.4	OCGs for Figure 2.3.	82
4.5	OCG for the subclass <code>ClazzA</code>	84
4.6	Example of ACG/iOCG construction.	85
4.7	OCG propagation through the ACG for the type resolution <code>ClazzA</code>	87
4.8	Runtime table used for representing the OCG from class <code>Clazz</code>	90
4.9	Example of additional conservation from combined transitive closure and edge downgrading.	93
4.10	State propagation through the Parameter Maps (circles indicate changes during iOCG formation).	95
4.11	An abstract overview of the simulated runtime environment.	98
4.12	Percentage of <i>method local</i> using iOCG compared to oracle method.	101
4.13	Percentage of method local memory location collected using iOCG compared to oracle method.	103
4.14	Percentage of OR operations to dynamic bytecode instructions.	105
4.15	Percentage of unique methods containing allocations.	106
5.1	The optimizing time ACG for the versions of <code>Clazz</code> in Figures 2.3 and 3.1. . .	112
5.2	Example user for the class <code>Clazz</code> from Figure 2.3.	112
5.3	Interprocedural CDG for <code>HoeUser</code> in Figure 5.2.	113
5.4	Subclass for the list class in Figure 3.9.	117
5.5	ACG for <code>append</code> in Figure 3.9 with the new subclasses in Figure 5.4.	118
5.6	Driver class for the subclasses of <code>ListClazz</code>	118
5.7	Sometimes safe regions of optimizations for <code>listBuilderDriver</code>	119
5.8	Speculative optimization in <code>listBuilderDriver</code>	120
5.9	Abstract view of the necessary fields in a validation registration.	122
5.10	Abstract view of code space and insertion of redirection stub.	125
5.11	A user class for the class file <code>ListClazzUser</code> in Figure 3.9.	128
5.12	The iCDG for the procedure <code>addJobs</code> shown in Figure 5.11.	130
5.13	The intermediate graphs and final iCDG after removing field nodes.	131
5.14	A new subclass of <code>syncClazz</code> in Figure 3.6.	138
5.15	The conceptual view of the <i>sometimes safe</i> inlined version of <code>Hoe</code>	139
5.16	The conceptual view of the <i>sometimes safe</i> inlined version of <code>Hoe</code> utilizing the CDG.	140
5.17	The new version of <code>Hoe</code> after code motion for Figure 5.14.	142

7.1	Theoretical modified method pointer to handle multiple context versions. . .	171
A.1	A graphical hierarchy of the implicit bytecode exceptions.	185
A.2	Definition of the <i>copy</i> , <i>kill</i> , and <i>transfer</i> operations.	195
A.3	Bytecode representation of <i>Bar</i> from Figure 2.3.	199
A.4	RAE representation of <i>Bar</i> from Figure A.3.	200
A.5	Graph construction for <i>Bar</i> in Figure A.4.	201
A.6	The algorithm for processing CFG to form graph	206
A.7	Property nodes broken into definition and usage nodes.	212
A.8	Final CDG formation from the graph for <i>Bar</i> in Figure A.5.	213
A.9	The format for the table entries used to annotate the CDG information into a class file.	215
A.10	The format for the table entries used to annotate the CDG information into a class file.	216
A.11	The format for an index stub within the annotated table.	218
A.12	Entry type specification for usage bit fields.	218
A.13	The format of the attribute field used to hold the table.	219
B.1	An abstract overview of the simulated runtime environment.	220
B.2	Section of the trace file for the <i>_227_mtrt</i> benchmark from the SpecJVM98 benchmark suite.	221
B.3	Bytecode disassembly for the method <i>GetVert</i>	222
B.4	The format of the thread state entries.	226
B.5	The format of the call stack entry.	227
B.6	The format of the initialed methods used by the method invoker.	229
B.7	Simulator memory format.	232

CHAPTER 1

INTRODUCTION

1.1 Object Oriented Languages and Java

Object oriented languages have established themselves as an enabling technology for large enterprise level applications. Among the set of commonly used object oriented languages in this domain is the Java programming language introduced by Sun Microsystems in 1995 [1]. Like other object oriented languages, Java is based on the notion of a class. A class is a user defined type that contains elements, called fields, and procedures for manipulating those elements. Classes are polymorphic, meaning that a class can inherit from a parent class allowing it to have access to fields within the parent class, specialize a procedure to its needs even if the procedure is defined within the parent class, and introduce additional fields and procedures. Java limits the number of parents to one but places no limitations on the number of children a parent can have. The inheritance allows a programmer to use the parent type of an object when writing code, relying on the dynamic runtime type of the object instance to choose the correct procedure. For example, a scene rendering routine can be written to take objects of type `shape` calling the procedure `draw` on each individual object. Then all drawable objects that inherit from `shape` can be passed to the scene rendering routine, allowing their runtime types

to choose the correct `draw` routine. In this way, the programmer can rely on the runtime type of the object instance for implementing the desired functionality. This form of choosing the appropriate functionality based on runtime-type virtual-call resolution is in contrast to the more traditional control structures used in languages such as C. Due to this form of control, object oriented languages tend to contain a large number of small procedure calls, each of which could have multiple potential targets based on the dynamic resolution of the objects.

The Java language also incorporates features that increase the level of programmer flexibility. One such feature is the ability to dynamically locate and link in the necessary class files only when the application first accesses them. This allows a programmer to change individual classes or even introduce new ones without the need to change the entire application. Additionally, Java has dynamic discovery mechanisms such as introspection and reflection which enable an application to dynamically discover the properties of a class and instantiate an object of that class even if the the class did not exist when the application was first written. This facilitates the incorporation of multiple packages from multiple independent software vendors as well as allowing the creation of applications that can dynamically create new classes to suit changing user needs.

Java is also designed to be machine independent. It targets a virtual machine architecture allowing the application to be written, compiled, and tested for only one architecture yet run on multiple targets. Furthermore, Java standardizes and simplifies interactions that were traditionally dependent on the operating system or server implementation. The thread model allows only one type of locking and a simple set of calls for accessing those

locks as well as guaranteeing thread safety for a set of library procedures. The network model defines a simple set of calls for establishing and using the desired network connections. The database model uses a simple set of library calls to abstract away database design issues. These standard interfaces also allow programmers to easily incorporate the use of packages that implement them into their applications. Therefore, these features further facilitate the integration of packages from multiple independent software vendors.

On top of all of this, Java also uses a memory manager relieving the programmer of the responsibility of tracking memory references and trying to free unused memory when the last reference to it expires. Programmers can write their code without fear of runaway memory usage growth. They rely on the virtual machine's memory manager to track live memory locations and recover dead ones.

With the additional benefits also come additional overheads. To overcome some of these overheads, Java relies on runtime optimizations to increase execution efficiency. However, Java's dynamic properties limit the applicability of traditional static analysis techniques. The lack of appropriate analysis techniques also limits the scope and aggressiveness of the optimizations applied. This thesis presents the design of our framework to facilitate aggressive runtime optimization by performing efficient and effective dynamic application analysis.

1.2 Analysis and Optimization

Interprocedural optimization is a critical means to enhance performance for object oriented languages. Since object oriented programs contain a large number of small procedure calls, most of the optimizations start by inlining procedure calls and optimizing over what used to be the procedure boundaries. Most of these forms of optimizations share a common assumption, that the set of class files used by the application is known. They have a “closed-world” view [2] - [21]. Based on this assumption, conservative call graphs are constructed and aggressive optimizations are performed. Significant performance gains have been achieved by using these techniques.

However, dynamically linked and loaded applications such as Java have the potential to introduce new subclasses of a given class at any time. This violates the “closed-world” assumption of static analysis. Some researcher have tried to tackle the problem of the elimination of the “closed-world” assumption by focusing on a subset of the application that can be considered closed [22], [23]. This form of optimization restricts inlining to only procedures that can be determined impossible to override at runtime. We call these procedures *monomorphic* procedures, meaning one and only one implementation of them exists within the application. The *monomorphic* procedures are either `final` procedures, meaning no other class can subclass them or they can be proven to be “sealed” procedures within a sealed package as defined in [22]. The optimizations and inlining of *monomorphic* procedures can be performed statically while allowing for additional runtime optimizations. In Chapter 5 we extend the notion of monomorphic procedures via

the use of our framework. We introduce the concept of context-based monomorphic procedures and describe the advantages and limitations imposed by restricting optimizations to only this subset.

However, although there is some gain from the inlining of provable *monomorphic* procedures, it has been shown that there are still substantial opportunities if more procedures are identified and inlined [10] - [12], [23] - [28]. Therefore, more aggressive runtimes make assumptions based on some form of profile information and determine a set of additional virtual calls that can be transformed from multiple potential call targets to inlined procedures guarded by control blocks. These procedures are then inlined and optimized along with their caller's code. Unlike the inlining of provable *monomorphic* procedures, this second set is not guaranteed to remain closed in the presence of dynamic class loading. This then leads to design concerns about how to detect that the current call graph and class hierarchy assumptions have changed and what to do in the presence of these changes. Both of these are addressed in our framework.

Sometimes it is beneficial to use known calling context and perform swift, on-the-fly analysis to enable first invocation optimizations. However, traditional forms of interprocedural analysis are too costly in both time and space for application to this analysis domain. As a result, most runtimes that attempt to employ on-the-fly optimizations restrict the analysis and scope of the optimizations to a swift, safe, intraprocedural subset. Our framework is designed to enable efficient and effective on-the-fly interprocedural analysis as well as provide more comprehensive intraprocedural information. This combination enables optimizations previously believed to be too costly for runtime deployment.

In Chapter 4, we describe and show results for an implementation of our framework to perform swift, safe, on-the-fly, interprocedural analysis.

1.3 Overview of Our System

The proposed dynamic optimization framework consists of three major building blocks: the static analysis engine, the dynamic analysis engine, and the dynamic optimization engine. These are shown conceptually within the corresponding portion of the Java runtime in Figure 1.1.

The static analysis engine operates on each individual class file at compile time. It produces a compact summary of each method, upon which the dynamic analysis engine performs various types of efficient runtime analysis. The summaries become available to the dynamic analysis engine through the standard annotation mechanism defined by Java specification [29]. The verification of this annotation is important to uphold the tight Java security model, and in Chapter 2, we present an approach for meeting this constraint that incorporates the best of both worlds.

The proposed summary of a method is referred to as a *Compact Dataflow Graph* (CDG). It is a dataflow graph since useful dataflow information can be easily extracted from this graph. However, it contains more than just dataflow information. It is a compact graph since all internal units, such as local variables, are removed from the graph. The details will be explained in Chapter 2.

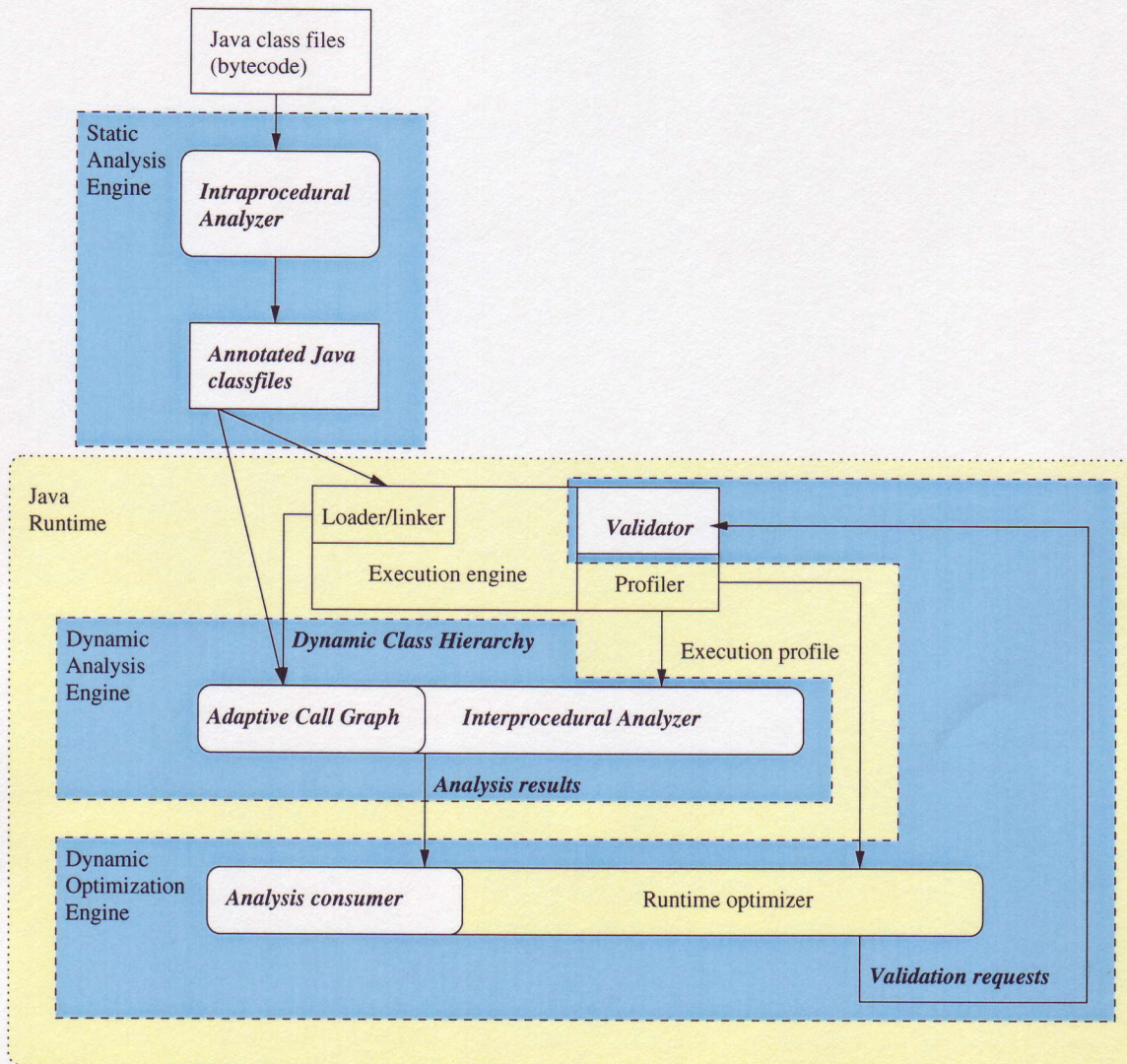


Figure 1.1 An overview of the Dynamic Application Analysis Framework.

1.3.1 Runtime architecture phases

At runtime, the dynamic analysis engine generates useful optimization tips in two different modes. The two modes are defined as *first mode*, or in the same process as the executing application, and *second mode*, or in a separate process from the executing application.

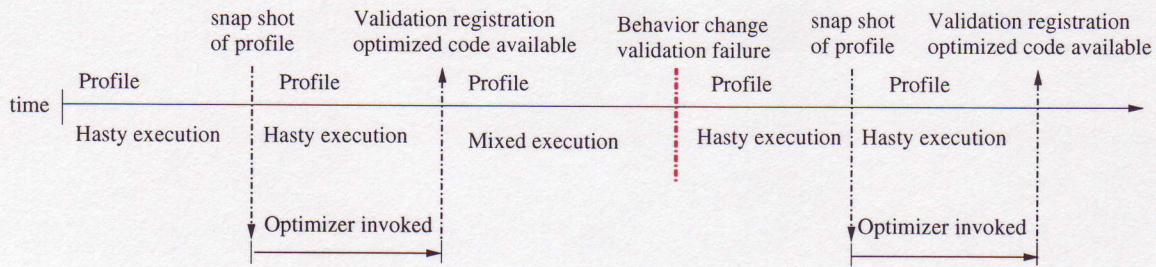


Figure 1.2 A conceptual view of the phases during the lifetime of an application.

Figure 1.2 shows a conceptual view of an application's progression through several phases during its lifetime. In this figure, the horizontal lines can be thought of as different processes executing over time. The top horizontal line is the main application while the horizontal lines labeled *Optimizer invoked* are separate processes running in parallel with the actual application. The vertical lines represent events. Some of these events also involve a production and consumption of information, and therefore the vertical lines indicate a direction of information flow. For example, the snapshot of the profile produces information that is consumed by the runtime optimizer. When the runtime optimizer completes, it produces new code and validation requirements that are consumed by the Java runtime. The profiling of the application is continuous and our analysis can also be conducted in conjunction with it. This is the first mode of analysis. The analysis results produced here can either be instantly consumed, such as deciding on stack allocation of a new object instance, or passed on to the optimizer for later consumption.

The runtime optimizer is considered a separate process with a static set of input information. Analysis performed then is similar in concept to static analysis with the

added safeguards to handle dynamic class loading. This is the second mode of analysis. The code being executed by the runtime can be in *hasty execution* mode, which we define as interpreted or unoptimized code. This is shown below the timeline in Figure 1.2. At some point, the runtime decides that enough profile information has been generated and “snap shots” the profile collected. We show these events as vertical lines crossing the timeline in Figure 1.2. The runtime then invokes the runtime optimizer. Once the runtime optimizer completes, it passes the produced optimized code back to the runtime along with any validation requests. This exchange is shown as the vertical line marked “validation registration” and “optimized code available,” in Figure 1.2. The transferal of the optimized code and validation requests then transitions the runtime into the next phase of code execution, mixed mode. In this phase of execution, both optimized and unoptimized code coexist in the runtime. Note that the profiler is still running and whether or not to restart the profiling, discarding all previously collected profile information, is dependent on the particular runtime. In our framework, we now also transition back to the first mode of analysis. At some point during the execution, an event occurs which either significantly changes the behavior of the application or forces a recovery from a validation failure. At this point in our timeline in Figure 1.2, we assume the event was significant enough for the runtime to abandon optimized code and transition back to *hasty execution* mode. The transition to *hasty execution* mode then starts the cycle over again. In Chapter 5, we describe the rollback and recovery mechanisms necessary in the event of validation failure.

1.3.2 First mode analysis

The first mode of performing analysis, we define as occurring while the application is running. The Java runtime executes a Java application and performs profiling at the same time. Our analysis used during this phase has the advantage of being context sensitive and knowing the exact calling context of the method. However, there are overheads that can impede performance and therefore can impact the strength of the analysis. For example, if we use our framework to make swift decision on whether to allocate an object on the stack or the heap, we need to make this decision at the point the object is allocated. However, in order to decide, we need to analyze what will occur over the object's lifetime to determine if the object has the potential to survive its allocating method. To do this, we construct a quick, safe interprocedural analysis at the point the method containing the allocation is executed. We describe an implementation of our framework for this form of analysis in Chapter 4. The analysis conducted in this first mode is not restricted to just this form of analysis. For example, it can be used to augment the profile data by constructing the interprocedural results as the profile is being collected. Then when the snap shot is taken, the context-sensitive analysis information is passed as part of the input set for the optimizer. The profiling is ongoing and as such the first mode of analysis for this example is considered continuous. We describe several forms of first mode analysis in Chapter 3.

If the analysis results are consumed in the first mode, the optimized code is specialized to a particular calling context. Therefore the optimized code has a very limited lifespan

and in some cases becomes single use. This means that the optimizations may be discarded after execution and regenerated should the same calling context be encountered again. Therefore, not only must the analysis performed during in this mode be designed efficiently, weighing the costs versus benefits, but also the choice of optimization and overhead of implementing it. The structures we designed as part of our framework help facilitate this form of analysis by reducing some of the overheads.

1.3.3 Second mode analysis

The invocation of the runtime optimizer then brings us to the second mode for analysis generation. In the second mode, the dynamic analysis engine waits until the profiling stage snap shots. The profile, along with any output from first mode analysis collection, is then input to a separate process running the optimizer. The other inputs to this process include the currently known class hierarchy, which is used to construct a call graph. However, the second mode is not restricted to just a consumer of first mode analysis results. Our framework can also be used in conjunction with various adapted techniques developed in static algorithms to produce either context-sensitive or context-insensitive results. We discuss further the difference between the analysis in the second mode and the first mode in Chapter 3. By performing the analysis and consuming the results in the second mode, we avoid the extra overheads paid to keep the analysis information up-to-date dynamically during profiling. However, some optimization opportunities can be lost due to exact context information only present during the first mode. For example, notice that we are now missing the exact type information of formals passed to a procedure,

that was present during the first mode. To enable aggressive optimizations, the optimizer needs to incorporate safeguards and modifications to handle the incompleteness of the information due to the possibility of new classes being loaded into the system. We define several models of optimization and describe the necessary safeguards needed for each in Chapter 5. We also present our framework for validation, r rollback, and recovery based on these models, defining what is meant by the *validation registration* on the timeline in Figure 1.2. In order to enable context-sensitive optimization in the second mode, we need a system for identifying context and accessing the correct version. We discuss the restriction and potential design of such a system in Chapter 7 as part of future work.

1.3.4 Consumption of analysis results

The analysis results can be thought of as producing optimization tips. We classify optimization tips as falling into several main categories. The first category is when the optimization tip is absolutely valid for every possible execution path. We refer to this case as *always safe*. In this case, we can perform optimization without any trouble; however, the number of tips that fall under this subset is relatively small. The second category is when the optimization tip is conditionally safe since not every possible execution path has been exposed yet. In this case, the lifespan of the optimized code can become short.

In the second mode, if the optimizer chooses to perform aggressive optimization using conditional tips, it also incorporates the appropriate validation checks. We break this form of optimization into two subcategories: *sometimes safe* in which the optimizer embeds the validation checks into the optimized code, and *speculatively safe* in which

the optimizer relies on validation and rollback in the runtime. The primary difference between the two is the assumptions made about the state and stability of the class hierarchy contained within the runtime at the time the optimizer runs. We describe this further in Chapter 5. A combination of the two aggressive optimization techniques can also be employed where some assumptions may be speculative while others are validated in the code.

1.4 Primary Contributions

The primary contributions of this work as follows.

- An efficient and effective framework for dynamic application analysis and validation of a subset of runtime optimizations in the presence of dynamic class loading. We present our framework for analysis and discuss the types of optimizations enabled by it. We identify and classify the basic optimization models including the necessary validation, rollback, and recovery for each model. Our framework allows for validation using the CDG to enable techniques that swiftly verify the correctness of some of the optimization decisions with the potential to facilitate more aggressive optimizations and expand the lifetime of the optimized code for the given application segment.
- A graph that represents intraprocedural object instances and that is independent of the detail of the internal runtime representation called a *Compact Dataflow Graph* (CDG). We show that the CDG is a key mechanism for enabling a large range

of efficient dynamic analysis and optimizations. The CDG efficiently represents the important object instance information at the intraprocedural level and enables swift propagation of results interprocedurally.

- A design for performing swift interprocedural analysis based on the use of the CDG. This includes a call graph abstraction we call an *Adaptive Call Graph (ACG)*. The ACG differs from a traditional call graph in two important ways. First, it is formed using the procedure under consideration for optimization as the entry point and not necessarily the `main` procedure which is the entry point for the entire application. Second, it incorporates context information and represents points within the call graph that can change and therefore may require some additional adaptation.
- An example use of the CDG for dynamic interprocedural analysis to guide optimization. We extract points-to relations from the CDG and develop an undirected form of a unification points-to relation called an *Object Connection Graph (OCG)*. The OCG is designed to facilitate swift interprocedural analysis in a running, dynamically loaded application. This representation can be created swiftly from the information in the CDG and shows the power of the information representation contained within the CDG. The OCG is designed also to enable swift interprocedural propagation of the information, and in turn identify a subset of objects as local to the allocating method. These object are allocated on the stack at the time of allocation.

- Classification of optimization strategies based on the consumption of information generated by the framework. We also identify and classify the types of validation and recovery mechanisms needed for the different strategies. We then give examples of these strategies and how they impact different optimizations.

The structure of the remainder of the thesis follows roughly the main components identified in Figure 1.1. In Chapter 2, we present the construction and representation for the CDG, shown in Figure 1.1 as the section labeled *intraprocedural analysis*. We address the actual format of the annotations shown in the box labeled *annotated Java class files* in Appendix A. In Chapter 3 we describe the design of the dynamic analysis framework. This includes descriptions of the sections labeled *dynamic class hierarchy*, *adaptive call graph*, *interprocedural analyzer*, and *analysis results*, in Figure 1.1. Also in Chapter 3, we identify a set of optimizations that can benefit from the analysis results generated by our framework and present an overview of how the intermediate structures may be used to enable them. Next in Chapter 4, we present an actual implementation of one of the dynamic analysis techniques. We introduce an intermediate representation designed for efficiency and present results for a set of benchmarks. This is followed by Chapter 5 in which we present the framework for dynamic optimization and validation. We describe the *analysis consumer*, *validation requests*, and *validator* shown in Figure 1.1. We discuss the basic structures used in the framework, classify three primary optimization strategies, and present an overview of what types of validation and rollback are required for several types of optimizations under the different strategies. We follow this in Chapter 6 with

a review of related research both in the realm of static analysis and in the domain of dynamic analysis. Finally, in Chapter 7 we discuss the future directions for this research.

CHAPTER 2

INTRAPROCEDURAL ANALYSIS

Understanding the use and interactions of object instances within a dynamic object oriented application is essential to not only locating the correct target for a virtual method call, but also to enabling a large subset of optimizations and validations. However, since the application is dynamically loaded and linked, the actual interprocedural information may not be fully available until the application is running. Therefore, the goal of our intraprocedural analysis is not only to discover the use and interaction between unique object instances, but to represent it in such a way as to facilitate the swift connection and propagation of the information interprocedurally. In this chapter, we describe our intraprocedural analysis and representation that are the building blocks of our interprocedural analysis.

2.1 Overview

Conceptually we can view a procedure as shown in Figure 2.1. It has inputs - the formals F_0 , F_1 , and F_2 - which represent unique object instances locations entering a method. It also has outputs, P_0 , P_1 , and return value F_{-1} , that are inputs to other procedures. It can also create new object instances, such as `obj1` and `obj2`, and use

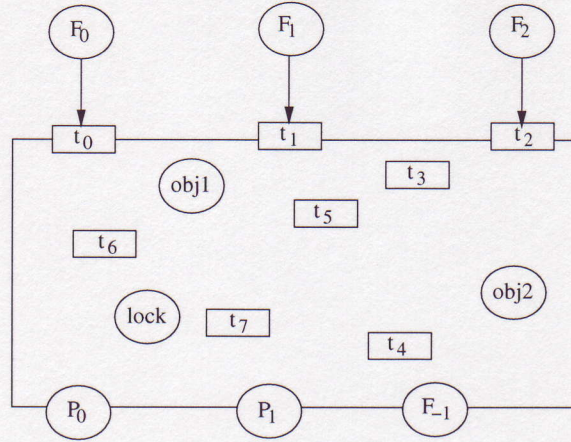


Figure 2.1 A conceptual view of a procedure.

properties associated with the object instances, such as `lock`. The procedure also contains a set of temporary locations where any of the object instances can reside while in use.

However, what is missing from this view is how these locations shown in Figure 2.1 interact. The purpose of intraprocedural analysis is to analyze each method in such a way as to discover the interactions and properties of these locations and then to distill this information to remove the internal temporary locations from the representation. This result is then representative of the method's effects on the unique object instances it comes in contact with. The set of interactions between unique object instances are then represented in such a way as to facilitate swift interprocedural propagation of the information.

For example, given the conceptual view of our method shown in Figure 2.1, the analysis first discovers the interaction shown in Figure 2.2(a). However, the goal of our analysis is not only to discover this interaction and dataflow but to also reduce

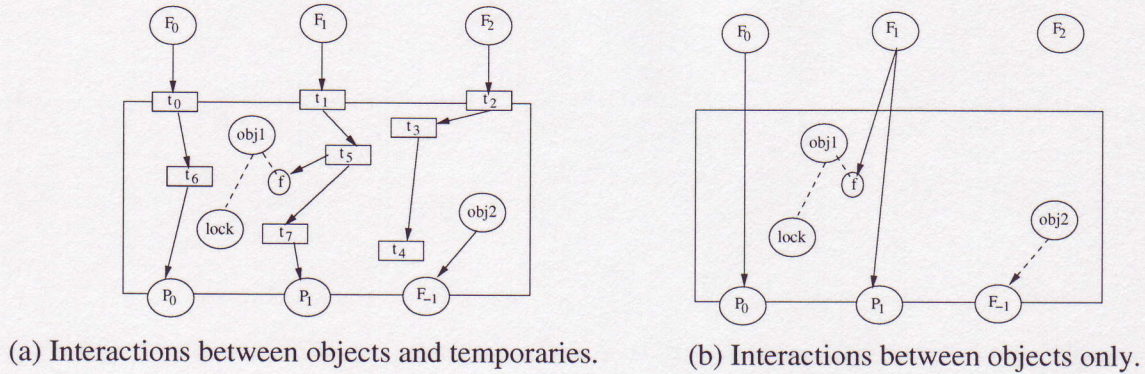


Figure 2.2 A conceptual view of intraprocedural analysis.

it such that only the object instances are left. Basically, we are not concerned with the temporary locations used, since these are internal to the method. They can be viewed as only temporary place-holders for the actual object instances. Instead, what interests us is how the actual memory locations representing the object instances are interconnected and used. This interaction and use is what defines not only the target of virtual method calls, but how these locations can be laid out and optimized. Furthermore, since temporaries are internal to the individual procedures, they are meaningless for interprocedural propagation of information. The reduced version contains the relations between the unique object instances used in the method, minus the temporary locations. This representation is shown in Figure 2.2(b). We call this reduced graph a *Compact Dataflow Graph (CDG)*.

Context independence is maintained during the intraprocedural analysis by not including any calling context or application-specific information during this phase. Its compact design and context independence allow persistence by utilizing the annotation

Table 2.1 Definition of entries in a Dataflow graph.

Node symbols	Definition
P_i	the i th formal of the method m
(P_{-1}, l)	the <i>return</i> value of the method m at line l
l	the object created at the line l
(v, l)	a definition of the variable v at line l
(f, l)	a field access of the field f at line l
(n, l)	the n th formal of the method invoked at line l
$(-1, l)$	the return variable of the method invoked at line l
(t, l)	the throwing (t) , of an exception at line l
(L, l)	the locking (L) , of an object at line l
(U, l)	the unlocking (U) , of an object at line l

There are two types of edges in the DG of the method m :

\rightarrow, \leftarrow	dataflow (strong) edge
$---$	association (weak) edge

mechanisms provided by the Java specifications [29]. Furthermore, the virtual machine independence is maintained by representing the CDG in terms of bytecode level information.

2.1.1 The Compact Dataflow Graph

We now examine the structure used to describe the intraprocedural relationships in more detail. The locations in Figure 2.1 contain reference values and can be divided into several fundamental types of nodes. These nodes in conjunction with their associated edges form the basis of the initial dataflow graph. Table 2.1 lists the types of nodes and edges present in a dataflow graph. They are defined in more detail as follows:

- **Local Variable Nodes:** Nodes representing a temporary variable name. These are represented by nodes of type (v, l) , where l is the bytecode line number that defines the local variable location.
- **Formal Value Nodes:** Nodes representing the formals to a method, these include any object instance returned by the method. These are represented by nodes of type P_i and (P_{-1}, l) , where P_{-1} represents a return value and l is the line number it occurred at.
- **Parameter Value Node:** Nodes representing reference values passed to callee methods. These are represented by nodes of type (n, l) and $(-1, l)$, where $(-1, l)$ is a reference value returned from a callee method at line number l .
- **Allocation Node:** Nodes representing new object instances being allocated within the method. These are represented by nodes of type l , where l is the line number the allocation occurred at.
- **Field Nodes:** Nodes representing a field associated with another object instance within the graph. These are represented by nodes of type (f, l) , where f is the constant pool identifier for the field and l is the line number the field was accessed at.
- **Property Nodes:** Nodes that represent a property associated with an access of a reference value. These are represented by nodes of type (t, l) , (L, l) , and (U, l) , where t is the use of a reference value to *throw* an exception, L is the locking of

the monitor associated with a reference value, U is the unlocking on the monitor associated with a reference value, and l is the line number at which the event occurred.

- **Global Node:** Nodes that represent an object instance that is globally visible to all threads running within the application. In Java, these object instances are associated with a class file instead of the particular object instances and are declared using the `static` key word. These nodes are represented in the graph by nodes (g, l) where g is the identifier for the global and l is the line number the access occurs at.

Under these definitions, the locations $t_0, t_1, t_2, t_3, t_4, t_5, t_6$, and t_7 , shown in Figures 2.1 and 2.2(a), become *local variable nodes*. Locations F_0, F_1, F_2 , and F_{-1} become *formal value nodes*. Locations P_0 and P_1 become *parameter value nodes*. Locations $obj1$ and $obj2$ become *allocation nodes*. Location f becomes a *field node* and the location `lock` becomes a property node. Note that property nodes differ from the other forms of nodes in that they do not represent the flow of data or a connection between the nodes. Instead, they represent a property that is associated with a given access to a node that may affect the state of the reference value when the graph is used to perform analysis. Since our goal is to provide an intraprocedural representation that accurately represents the usage and interconnections between the unique object instances within a method, the property nodes are necessary to correctly represent the usage of the object instances. For example, knowing where and when an object instance is used to obtain a *lock* as well as conveying

the fact that the *lock* is associated with a given object instance is necessary to identify unnecessary synchronization operations dynamically.

Although the analysis used to generate a CDG is performed on a bytecode representation of the method, m , it can be conceptually viewed as performing the following steps.

1. Execute a reaching definition algorithm.
2. Construct an initial graph from the reaching definition.
3. Complete the initial graph by extending edges around temporary nodes, forming a transitive closure on the initial graph.
4. Remove temporary nodes and edges from the extended graph.

Two types of edges are used in the graphs, a dataflow edge and an association edge. The dataflow edge represents the flow of data between two locations, while the association edge is a means of attaching field locations and properties to their parent objects. Data does not flow along an association edge, and the edge does not contain direction. Table 2.2 gives the edges used to connect the nodes within the graph for a set of source level style expressions. To better explain an association edge, refer to the third entry in Table 2.2, $l : v.f := w$. This expression at line number l stores the value of w in f , of object v . Therefore, we denote the data flowing from w to f with the dataflow edge, but denote the relationship between v and f with an association edge. Association edges are also used for properties associated with a reference value, such as the locking/unlocking of

Table 2.2 Rules to add edges in initial DGs. The l 's represent line numbers.

expression	edges added
$l : v := newC()$	add a dataflow edge $l \rightarrow (v, l)$
$l : v := w$	for each definition (w, l') reaching line l add a dataflow edge $(w, l') \rightarrow (v, l)$
$l : v.f := w$	for each definition (v, l') reaching line l add an association edge $(v, l') \dashrightarrow (f, l)$ for each definition (w, l') reaching line l add a dataflow edge $(w, l') \rightarrow (f, l)$
$l : v := w.f$	for each definition (w, l') reaching line l add an association edge $(w, l') \dashrightarrow (f, l)$ add a dataflow edge $(f, l) \rightarrow (v, l)$
$l : v := p(w_0, \dots, w_k)$	add a dataflow edge $(-1, l) \rightarrow (v, l)$ for each $i = 0 \dots k$ for each definition (w_i, l') reaching line l , add a dataflow edge $(w_i, l') \rightarrow (i, l)$
$l : synchronize(w) \{$ \dots $l'' : \}$	for each definition (w, l') reaching line l , add an association edge $(w, l') \dashrightarrow (L, l)$ add an association edge $(w, l') \dashrightarrow (U, l'')$
$l : throw(w)$	for each definition (w, l') reaching line l , add an association edge $(w, l') \dashrightarrow (t, l)$ where t represents the state <i>thrown</i>

the reference's monitor or the use of the object to throw an exception. These are shown by the last two entries in Table 2.2.

The initial graph is then expanded into an extended graph by extending edges around local variable or intermediate locations, thus forming a transitive closure. The rules for extending these edges are shown in Table 2.3. They simply allow the graph to bypass any local variables used within the initial graph while still accurately representing the relationship and usage of the object instances.

Table 2.3 Rules to add edges to the initial graph to form the extended graph.

if $n_1 \rightarrow n'$ and $n' \rightarrow n_2$	add a dataflow edge $n_1 \rightarrow n_2$
if $n_1 \rightarrow n'$ and $n' \dashrightarrow n_2$	add an association edge $n_1 \dashrightarrow n_2$

where n' is a local variable node.

The first rule allows the data flowing from one object instance to another object instance to be represented directly without the local variable node. The dataflow edge between n_1 and n' states that they can be considered direct aliases for each other, and likewise for the edge between n' and n_2 . Therefore, since $n_1 = n'$ and $n' = n_2$, we know $n_1 = n_2$. The next rule simply states that if an association edge exists on a local variable node, it is extended with an association edge to any node with a dataflow edge entering the local variable node. Therefore, a relation between the two objects is maintained even though no data flows between them. This extension of the association edge enables relations that are only associations to local variable nodes to be associated with the object instance nodes while maintaining the read/write direction of the access. Note that if no edges are leaving n' , then the relation $n_1 \rightarrow n' \leftarrow n_2$ has no effect on the object instances n_1 and n_2 . Therefore the node n' and its associated edges can be safely removed from the graph without loss of information.

After applying the rules in Table 2.3, the CDG is formed by removing the extraneous temporary nodes and their related edges from the graph. The edges added then capture the relationships between the unique object instances used in the method.


```

1:  classClazz {
2:     Clazz f;
3:      staticClazz g;
4:
5:      <Clazz>(Clazz o){
6:          <object>(o);
7:      }
8:
9:
10:     voidHoe(Clazz o){
11:         Clazz a = newClazz();
12:         <Clazz>(a);
13:
14:         Clazz b = newClazz();
15:         <Clazz>(b);
16:
17:         o.Foo(a, b);
18:     }
19:
20:     voidFoo(Clazz o, Clazz p, Clazz q){
21:         Clazz r = newClazz();
22:         <Clazz>(r);
23:
24:         o.Bar(p, q);
25:     }
26:
27:     voidBar(Clazz o, Clazz x, Clazz y){
28:         Clazz z = newClazz();
29:         <Clazz>(z);
30:
31:         z.f = x;
32:         g = z;
33:     }
34: }

```

Figure 2.3 Example for illustrating construction of the CDGs.

2.2 CDG Formation

To better illustrate the formation of a CDG, we use the example in Figure 2.3. The class in this example contains four methods: an initializer, `<Clazz>`, and three other methods, `Hoe`, `Foo`, and `Bar`. In Java, the default method type is *virtual*, meaning that an object instance is used to locate the correct definition of the procedure. In the example class in Figure 2.3, we have shown this object instance explicitly as the first parameter, *Clazz o*, in each of the four procedures in the class. Therefore, for the call shown on line

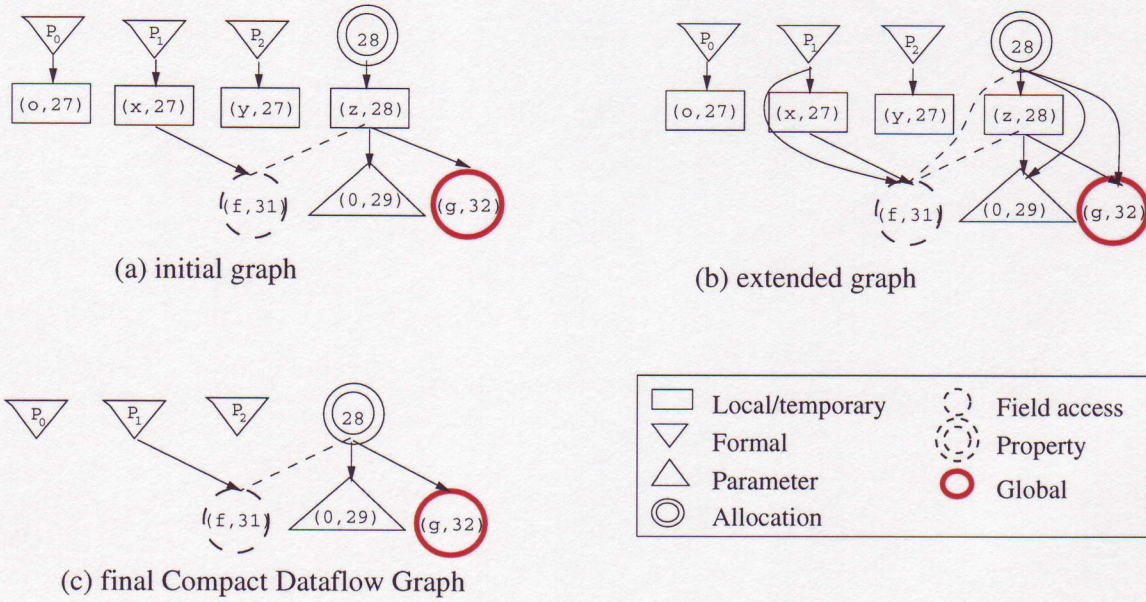


Figure 2.4 CDG construction steps for method Bar.

17, `o.Foo(a, b)`, the object instance, `o`, maps to the first parameter in `Foo`'s parameter list, `Clazz o`. To illustrate the construction of a CDG, we use the method `Bar`. We also show, but do not discuss, CDGs for the procedures `Hoe`, `Foo`, and `<Clazz>`, which are used in subsequent chapters.

Figure 2.4 constructs the CDG from the method `Bar` shown in Figure 2.3. The bottom right side gives a legend for the different node types defined in this chapter. We walk through the construction of the CDG in a forward progression although the actual implementation uses a backward flow algorithm. The forward algorithm follows the normal execution progression and is therefore easier for readers to follow. We present the actual backwards algorithm performed on the bytecode representation of the method in Appendix A.

In Figure 2.4(a), the formals coming into the method `Bar` add three formal nodes to the graph, P_0 , P_1 , and P_2 . These are assigned to temporary locations, o , x , and y , adding three local variable nodes to the graph. Following the rule for formals given in the fifth row of Table 2.2, $(l:v := p(w_0, \dots, w_k))$, we attach solid dataflow edges between the appropriate pairs. The creation of the new object at line 28 of Figure 2.3 adds an allocation node labeled 28, shown in the upper right side of the graph. Its assignment to temporary location z adds a local variable node to the graph, labeled $(z, 28)$. They are then connected via a dataflow edge as shown in Figure 2.4. The call to the initializer at line 29 of Figure 2.3 is shown by the addition of the parameter node labeled $(0, 29)$ and the dataflow edge attaching the local variable node $(z, 28)$ to $(0, 29)$. The field assignment at line 31 adds the field access node labeled $(f, 31)$ to the graph, with two edges attached to it. The dashed edge associates it with the local variable node labeled $(z, 28)$, and the solid edge expresses the dataflow from the local variable node labeled $(x, 27)$. Finally, the assignment into static field location g adds the global node labeled $(g, 32)$ and the dataflow edge attaching the local variable node, $(z, 28)$, to it.

To form the extended graph in Figure 2.4(b), we apply the rules for edges given in Table 2.3, extending around the local variable nodes. We next remove all local variable nodes from the graph along with any edges incident on them, bringing us to the final CDG given in Figure 2.4(c).

The construction of the CDGs for the remaining three methods shown in Figure 2.3 are simpler. Figures 2.5 - 2.7 show the construction of these graphs.

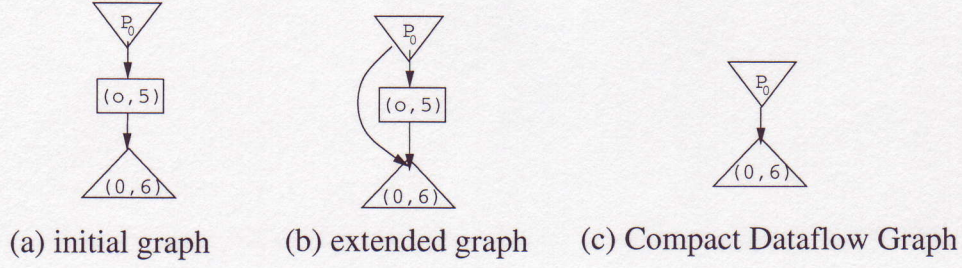


Figure 2.5 CDG construction for <Clazz> in Figure 2.3.

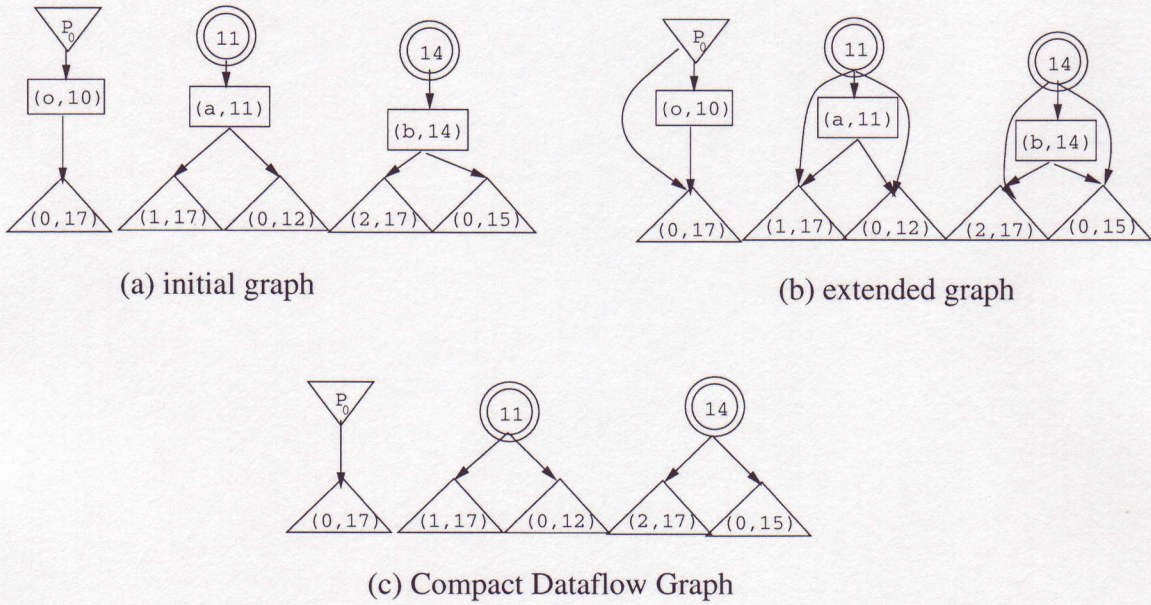


Figure 2.6 CDG construction for Hoe in Figure 2.3.

2.3 File Annotations

The final step is the persistence of the information via the annotation mechanism in the bytecode file format specifications given in [29]. Although the analysis can be conducted at load time, it is flow sensitive with a worst case time complexity of $O(n^3)$,

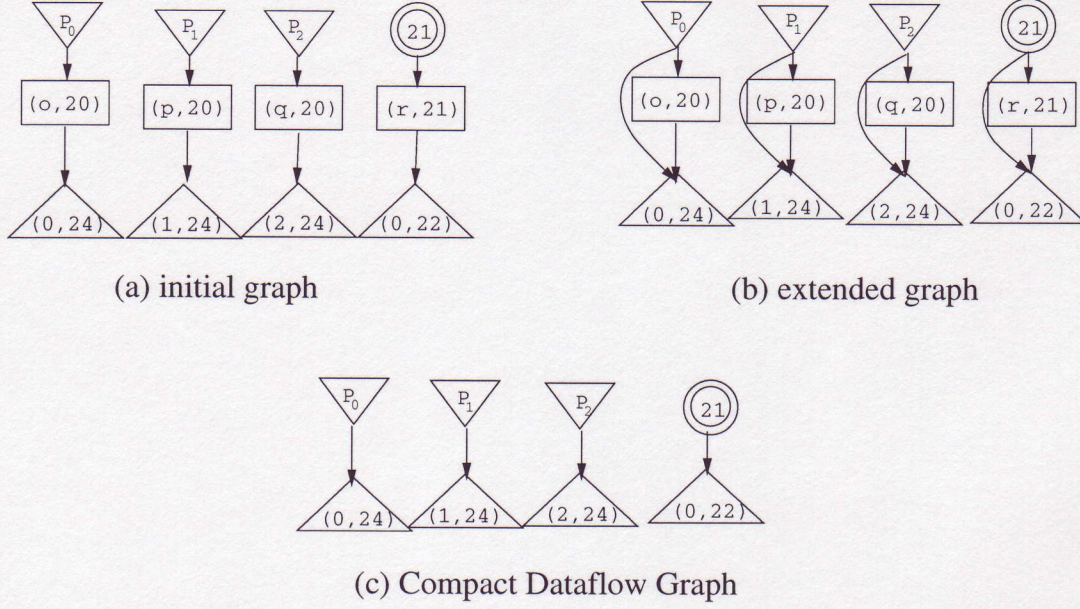


Figure 2.7 CDG construction for Foo in Figure 2.3.

where n is the number of basic blocks in the CFG. Note that when a method is a single basic block, the complexity is linear. The flow sensitivity allows the dynamic interprocedural analysis to be either flow-sensitive or flow-insensitive. For example, the synchronization removal algorithm presented in [16] uses flow sensitive analysis to remove extraneous synchronization operations even from *thread escaping* references. Although the work presented in [16] is based on a static, closed-world view of the Java application, by including the flow sensitivity in the CDG we can enable similar techniques to be employed dynamically.

However, Java is dynamically linked and loaded implying that, aside from the library files that are part of the runtime, the class files used by the application may arrive from outside sources at any time during the applications execution. Files obtained at runtime

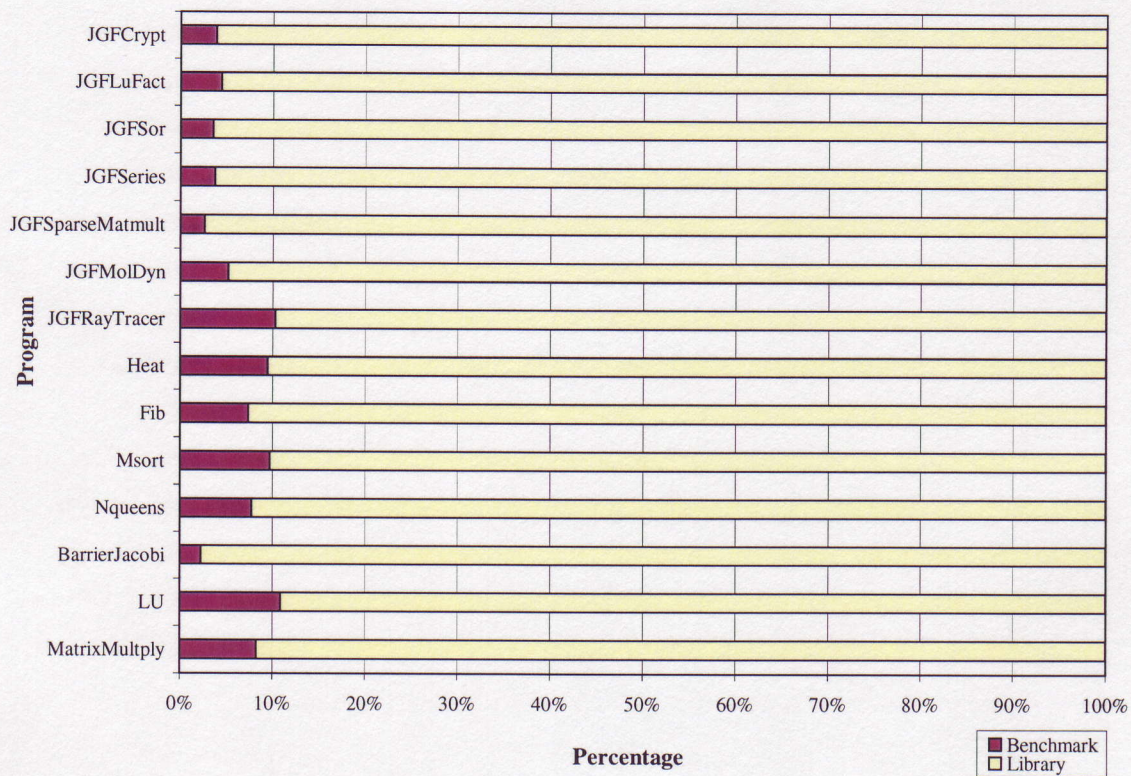


Figure 2.8 The percentage of unique methods in each benchmark that are from the benchmark versus library class files.

may not contain the necessary annotations. Even if the files do contain the annotations, the cost of verifying their correctness may approach the cost of creating them.

Not all of the intraprocedural analysis needs to be conducted dynamically. A significant portion of the unique methods used by an application are from the runtime library. Figure 2.8 shows, for the unique methods used in each of the examined set of programs, what percentage came from the program class files and what percentage were from the library class files. As can be seen from the graph, the percentage of methods unique to the application specific portion of the program versus the percentage of methods coming

Table 2.4 Percentage of actual methods used that were single basic block.

Programs	Percentage Programs	Percentage Library
JGFCrypt	47.6%	52.0%
JGFLUFact	41.7%	52.3%
JGFSOR	52.6%	52.0%
JGFSeries	55.0%	52.1%
JGFSparseMatmult	64.3%	52.2%
JGFMolDyn	53.6%	56.0%
JGFRayTracer	69.5%	52.1%
Heat	49.1%	52.2%
Fib	48.7%	54.0%
MSort	44.8%	52.5%
NQueens	51.1%	47.7%
BarrierJacobi	33.3%	50.1%
LU	45.0%	54.1%
MatrixMultiply	50.0%	54.0%
Total	51.0%	52.3%

from the library class file portion is relatively small. Furthermore, even for those methods that are analyzed dynamically, a significant portion of them are single basic block, meaning the analysis for them is linear. Table 2.4 shows, for the programs investigated, what percentage of the unique methods invoked by the application were single basic block. This is divided in Table 2.4 into those that were program specific and those that were part of the standard library files.

One solution to the security issue is to only statically persist the CDG in the library files. Since the library files are under the control of the runtime and considered part of the runtime, standard security measures such as sealed packages and signatures can be used. The file size expansion from annotating these files in a Java 1.2 library implementation was measured at 11.28%. Any new program files loaded into the system can have the CDG

constructed at either load time, or first invocation. For commonly used applications, some of the annotations created by the runtime could be made persistent on the deployment machine. Techniques similar to those we developed in [30], [31] could be used to recognize version changes within these persistent files both at a coarse and fine grain level, thus discarding and updating their CDGs only when necessary.

For the interested reader, Appendix A presents the actual analysis performed on the bytecode representation of the method. The final result is compared to the conceptual view presented in this chapter. Also presented is the actual format of the annotations within the bytecode files. These sections are not necessary for understanding the remainder of the thesis, which requires only the conceptual view of the CDG.

CHAPTER 3

DYNAMIC ANALYSIS: INTERPROCEDURAL ANALYSIS

The key to optimizing a dynamically linked object oriented style application is the ability to perform the interprocedural analysis assuming incomplete information. When the application is dynamically linked, the call graph can only be assumed to contain a partial set of the calls used within the running lifetime of the application. At any point during the execution, the system can load a new class file and increase the number of potential targets for one or more points within the call graph. The dynamic analysis framework must have a way to represent uncertainty and to perform analyses based on only partial information.

In addition to this uncertainty are concerns of time and space. With a dynamically linked and adaptively compiled application, the analysis engine is competing with the actual application for system resources. Therefore, the need for efficient use of memory and processor resources limits the use of some static analysis techniques. Furthermore, since the application can change behavior at different phases of execution, it is important for the analysis engine that guides the optimizer to have the analysis results in a timely fashion. Otherwise, an optimization may become obsolete before it can be applied, due to newly loaded class files or changes in user behavior.

In this chapter, we address the structures necessary for efficient dynamic interprocedural analysis. They include the class hierarchy representation already part of most VMs, the dynamically adaptive call graph, and the connecting of intraprocedural analysis results to arrive at an interprocedural solution. We present our designs and interfaces for each of these. Additionally, we discuss how our analysis framework can be used to provide the information necessary to perform a set of optimizations shown to be beneficial for Java style applications.

3.1 Class Hierarchy Representations

A *Class Hierarchy (CH)* representation is a structure used to represent the inheritance relationships between class files. Java requires that all ancestors of a new class file be initialized before the class file is initialized [29]. To facilitate this ordering, most Java virtual machines contain some form of representation for an application's currently known class hierarchy. We do not assume any particular structure for this representation, but instead identify the types of information necessary to implement our framework. The primary information needed by the analysis engine is the ability of the VM's CH to return results from two queries, parent of a given class file, and children of the given class file.

Note that although Java has a single inheritance structure for class files, it does not impose that constraint on interfaces. In fact, the number of interfaces that a class file can implement is limited only by the size of the 16-bit `interface_count` field within the

class file [29]. The part of the framework presented in this thesis covers only the single inheritance structure used for class files but can be easily extended to include interfaces.

3.2 Adaptive Call Graphs

One important tool for performing interprocedural analysis is a graph of the caller-callee structure within the application commonly called a call graph. This structure usually encompasses the entire application and contains multiple potential targets for virtual call sites. For our framework, we adapted the traditional definition of a call graph to better suit the dynamic application analysis problem. Instead of creating a single call graph for the entire application, we create a partial call graph for a procedure under analysis. Furthermore, based on context information known about the procedure being analyzed, we represent call sites within the call graph as either single target or unknown. We call our modified call graph an *Adaptive Call Graph* (ACG) because it adapts to a given procedure and context. We define an ACG as follows:

Definition 1 *An Adaptive Call Graph (ACG) is a call graph that extends from a root procedure, m , to include the potential callees of m such that given the calling context of m , the nodes in the ACG are of the following types.*

- **Known**, the callee has only one potential target within the given context.
- **Speculative**, the callee has two or more potential targets within the given context.

To explain the ACG, we again use the example introduced in Figure 2.3, and subclass it with the two new classes shown in Figure 3.1, `ClazzA` and `ClazzB`. Since we have


```

35: class ClazzA extends Clazz {
36:     void Foo(Clazz o, Clazz p, Clazz q){
37:         ClazzA r = new ClazzA();
38:         r.<Clazz>();
39:         Clazz P = r;
40:         o.f.Bar(P, q);
41:     }
42: }
43:
44: class ClazzB extends Clazz {
45:     void Bar(Clazz o, Clazz x, Clazz y){
46:         g = y;
47:     }
48: }
49:

```

Figure 3.1 Example subclasses for the class in Figure 2.3.

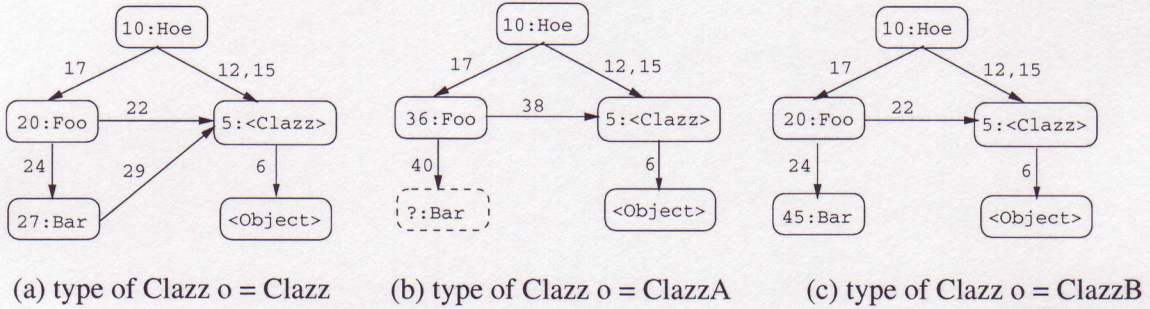


Figure 3.2 ACGs for the three potential types of Clazz o in Figure 2.3.

these classes in our CH, and the CH only contains loaded and initialized classes, all three definitions for an object of type Clazz are available. Figure 3.2 shows three ACGs for the root method Hoe for the three potential types of Clazz o. In the figure, solid nodes denoted *known* and dashed nodes denote *speculative* nodes. The edges specify the line number at which each method was called.

Note that even though the class Clazz has several subclasses that have been loaded and initialized into the runtime, at profile time we know the exact type of Clazz o, the

object used to call `Hoe`, as the Java runtime uses the runtime object instance to locate the correct method table and correct method resolution for `Hoe`. Therefore, since the object instance value, `Clazz o`, shown explicitly in this example as the first parameter to `Hoe`, is known, and this is also the object instance used to call `Foo`, this method is also definitively known. This fact is reflected by the use of a known node for `Foo` in all three ACGs in Figure 3.2 .

Not all nodes in a first mode, profile time ACG can be classified as *known*. The resolution of `Bar` is not *known* in the ACG in Figure 3.2(b). If the object `o` has the runtime type `ClazzA`, the resolution of `Bar` depends on the object instance type of its field, `f`, as the implementation of `Foo` in `ClazzA` uses `o.f` to locate `Bar` (line 36 of Figure 3.1). When performing and consuming analysis results during execution, resolution of the field type when constructing the ACG can require several accesses to memory to retrieve the type. Additionally, the exact field must be tracked through the iCDG built in conjunction with the ACG to be certain that it remains that type until the invocation site. Therefore, since overhead is a factor during the first mode, this level of resolution is not viewed as practical. Therefore, the ACG in Figure 3.2(b) shows `Bar` as a *speculative* node labeled `? : Bar`.

The type of node in the ACG does not necessarily remain the same if the interprocedural analysis is performed when the runtime optimizer runs (second mode). At this point, the calling context including the exact runtime type of the object is missing, making type determination difficult. The only information is on potential types, which can be determined from the CH. Using the same set of class files with the same CH defined at

profile time, no assumptions can be made about which of the three class files the object instance `Clazz o` will belong to. Therefore, both the instance of `20:Foo` shown in the first mode, profile time ACGs in Figure 3.2(a) and (c), and the instance of `36:Foo` shown in Figure 3.2(b), could be the target of the call to `Foo`. Additionally, potentially not yet loaded subclasses could introduce additional targets. This uncertainty makes the node for `Foo` in the second mode ACG speculative. However, optimization is still possible and we address this further in Chapter 5.

There are invocation targets that are known single targets and remain monomorphic even in the presence of dynamic class loading. These include *initializers* and methods declared as `final` or `static`. For *initializers*, the object is being created of a known type with a known descriptor. Therefore, there can be only one resolution for the *initializer* call. For the case of methods declared using the `final` key word, no subclass can override the method by definition of the use of the `final` modifier [29]. Methods declared with the `static` key word are resolved via the class object versus a given object instance. Like fields declared as `static`, there is only one implementation of them available, and therefore the target is monomorphic.

3.3 Analysis Information

The interprocedural analysis assumes the availability of the CH, ACG, and CDG. Based on these three structures, the following types of analyses can be made.

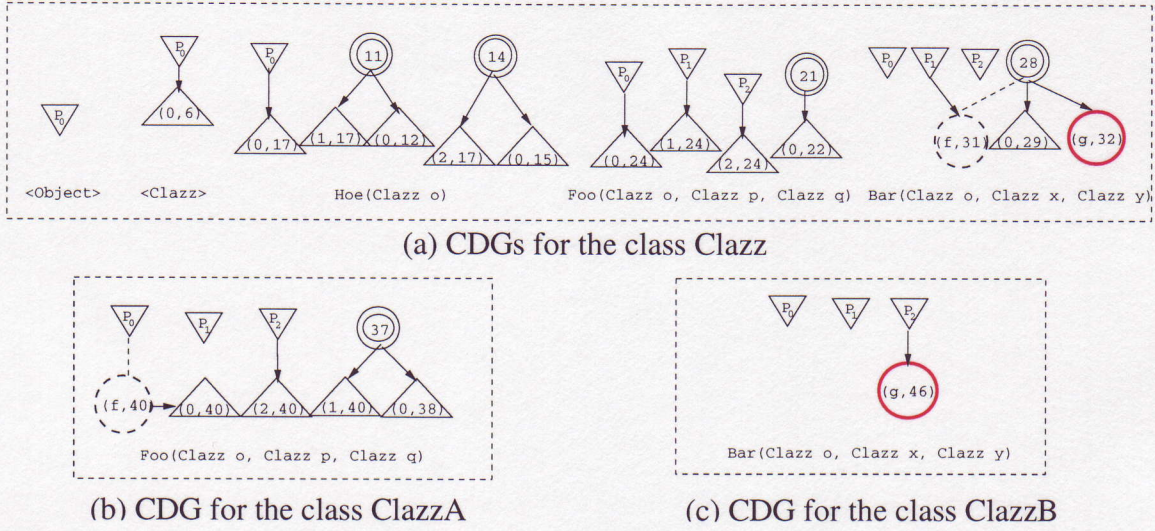


Figure 3.3 Final abstract source-level CDGs for the methods in Figures 2.3 and 3.1.

- *Access information*: When and how an object instance is accessed. For example, whether it is *read* or *written* as well as where within the method the access occurred.
- *Escape information*: Whether or not a given object instance escapes a given scope of control. At the thread level, this determines whether or not the object instance becomes visible to other threads. At the method level, this involves whether or not the object instance lifetime extends beyond its allocating method.
- *Property information*: Whether or not a particular object instance is used for a locking operation or to throw an explicit exception.

The formation of the ACG is an iterative process that connects the CDGs for each *known* node to form an *interprocedural CDG* (iCDG). *Super nodes* are used to connect formal and actual parameters at call sites, and are later replaced with edges. Figure 3.3

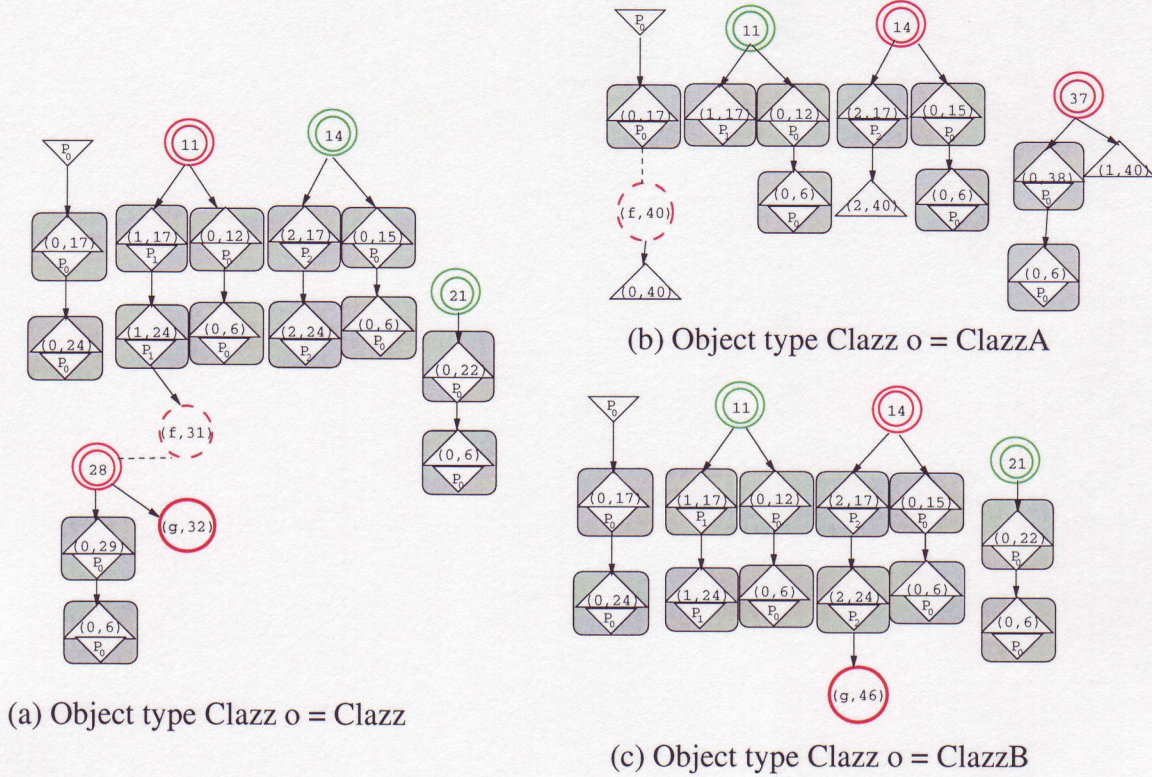


Figure 3.4 Results from interprocedural propagation of the three types ofClazz in Figure 2.3.

shows the CDGs for the class files introduced in Figure 3.1. Figure 3.4 shows the iCDGs constructed in conjunction with their respective ACGs shown in Figure 3.2. The *super nodes* are explicitly shown in Figure 3.4 as the larger, rounded-edged nodes containing both a caller parameter node and its corresponding callee formal node. The speculative node shown in Figure 3.2(b), is reflected in Figure 3.4(b) by the three parameter nodes $((0, 40), (1, 40), (2, 40))$ left open (not contained in super nodes). Recursive procedures are also connected via super nodes. Depending on the full type context information passed with recursive calls, the CDG connected within the iCDG can either

be duplicated for a partially overlapping type context, or directly connected to the existing one for a fully overlapping type context. We constructed these ACG/iCDG, and in general all ACG/iCDGs, as follows:

- Create root node in the ACG for the procedure being analyzed.
- For each procedure called by the root node:
 - Add a *known* node for any provably monomorphic procedures
 - Add a *speculative* node for all others.
- Examine the CDG for the root node and promote any *speculative* nodes to *known* nodes based on iCDG propagated context information.
- Form *super nodes* by connecting actual and formal parameters from the CDGs corresponding to the *known* nodes in the ACG, into the iCDG.
- For each *known* node in the ACG not yet resolved, treat it as a *root* node and repeat the previous steps.

There are two types of information that need propagation across the interprocedural boundary: state and edges. A reference value crossing an interprocedural boundary can exist in one or more of the following states:

- Thread Escaping (T Esc.) A reference to it is accessible from other threads.
- Method Escaping (M Esc.) The object out lives its allocating method.
- Thrown (Thr.) The object instance was thrown as an exception.

- Used in Locking (Lck.) The object's monitor is currently locked by the current thread.
- Read (R) The object instance or one of its fields was read.
- Written (W) The object instance or one of its fields was written.

A reference value crossing an interprocedural boundary can become linked to other object instances via field assignments within the callee. This then adds edges to the iCDG.

Edges are used to propagate state information between reference values and therefore new edges forged by a callee procedure should be propagated back to the caller. The level of refinement needed in the edge representation is dependent on the type of analysis as well as the phase at which the analysis is conducted. The interprocedural phase propagates these new edges such that all state information is correctly propagated. We refer to these new edges as *Links*.

3.4 Optimizations Enabled

The analysis can be used to drive optimization decisions. In this section we discuss the generation of analysis results and which optimizations would consume the results. We illustrate this analysis result generation by demonstrating the construction of the ACG/iCDG for each of the optimization types listed in Table 3.1. We defer discussion of the actual optimization models until Chapter 5. In that chapter, we focus on the optimization models that consume the analysis results presented. The aggressiveness of

Table 3.1 Analysis needed based on optimization.

optimization	information			propagation					
	acc.	esc.	prop.	Link	T Esc.	M Esc.	Thr.	Lck.	R/W
stack alloc.		X		X		X			
sync. removal		X	X	X	X			X	
result caching	X	X	X	X	X				X
lifetime	X	X		X	X				
memory layout	X	X		X	X				X
code motion	X	X	X	X	X	X	X	X	X
race detect.	X	X	X	X	X			X	X

the optimization determines the level of validation necessary with the dynamic loading of new subclasses; therefore, we also leave discussion of the validation type and level for each optimization until that chapter.

The following subsections correlate directly to the rows in Table 3.1. The columns in this table are broken into two main sets, information and propagation. The information set refers to the types of information that are contained within the CDG that is pertinent to generating the analysis results for the given optimization. The propagation set refers to the propagation of the information interprocedurally. For each optimization, it lists what level of information propagation is needed to produce the correct analysis results.

3.4.1 Stack allocation of objects

Stack allocation of dynamically allocated objects is an optimization that reduces overhead of not only access delays going through the main heap, but also reduces the number of short-lived object instances within the heap. This reduction of heap allocated object instances reduces the number of garbage collection epochs required by an application.

Therefore, for applications containing a large number of method local allocations, this optimization has been shown to contribute significantly to performance [15], [18].

In Table 3.1, we identified one type of “information” and two types of properties as necessary for the interprocedural analysis result generation for this optimization. The necessity to include escape information stems from the need to determine if a reference to the object instance being considered for stack allocation will exceed the lifespan of its allocating method. Therefore, the need for both method escaping (**M. Esp.**) and new edge information (**Link**) to perform the interprocedural analysis. In order to accurately make this determination of escaping state of an object, we need to propagate not only the escaping state of the references within the iCDG, but also any new links forged. For example, if an object being considered for stack allocation gets written into the field of another object instance and that object instance exceeds the lifetime of the allocating method, then the information retained by the new link is necessary for the correct determination of the method escaping state of the original object instance.

Referring back to the two mode of analysis generation we described in Chapter 1, identifying object instances that can be stack allocated can occur in either of these modes. The first mode of analysis result generation has the advantage of the exact context being known, but the disadvantage of the analysis potentially delaying the execution of the code segment substantially. Additionally, first mode analysis has the added potential to swiftly identify object instances that are candidates for stack allocation at or prior to the point of allocation even when the method is being executed for the very first time. This enables the system to catch and reduce the overhead of short-lived object instances

that occur infrequently as well as identify allocations whose state depends on calling context information. However, the cost of performing the analysis during the profiling stage needs to be low or it could outweigh any performance gained from the consumption of its results.

The second mode occurs after a profile has been collected and the runtime optimizer is invoked. Although the application is executing at the same time that the optimizer is running, the optimizer is considered a separate and distinct process and the execution is not stalled waiting for the analysis results in order to continue. Rather, the second mode has a snapshot of the application as input but no exact context information. Therefore, analysis performed during the second mode can alleviate some of the cost concerns; however, it is missing the exact calling context information that was present during the first mode. With the cost alleviation, the analysis can become more aggressive, and consider items such as the escaping state for all potential targets known to be present in the class hierarchy and make a general aggressive optimized version of the code. It can identify object instances that may exist across several interprocedural boundaries and even have extended lifetimes as still meeting the criteria for stack allocation. The loss of calling context information can cause the analysis results to add a level of conservatism due to the consideration of multiple potential targets for call site; however, this can be mitigated by some forms of the optimization models covered in Chapter 5.

In both cases, we define a stack allocatable object instance as follows.

Definition 2 *An object instance is defined to be stack allocatable if and only if the full lifetime of the object instance can be analyzed and no reference to the object instance survives the lifetime of its allocating method.*

Based on Definition 2, we then define the criteria for which an object instance is considered to survive its allocating method.

Definition 3 *An object instance is said to survive the lifetime of its allocating method, or to be method escaping, if one of the following events occurs during its lifetime.*

- *Global Escaping: A reference to the object instance becomes accessible via a global variable.*
- *Reference Escaping: A reference to the object instance is stored in the address of another object instance, either through an assignment or as a field value within the other object instance, and the other object instance's lifetime exceeds the allocating method.*
- *Return Escaping: A reference to the object instance is returned from its allocating method.*
- *Unknown Path Escaping: A reference to the object instance crosses an interprocedural boundary for which the current thread does not have access (native methods, methods without CDGs, passed in a call to another thread).*

The first three - *global escaping*, *reference escaping*, and *return escaping* - concern an object instance that has a reference to it existing beyond the allocating method. The fourth

item, *unknown path escaping*, concerns an object instance whose full lifetime cannot be analyzed because portions of it exist in code not accessible to the analyzer.

The analysis varies based on whether it is conducted during the first or second mode. To illustrate the information collected in each mode, we again use the example class files given in Figures 2.3 and 3.1. The abstract, source-level view of the CDGs from these three class files is shown in Figure 3.3. We assume that our Class Hierarchy (CH) contains all three classes: `Clazz`, `ClazzA`, and `ClazzB`. Note that the CDG for the `<object>` initializer called at line 6 of Figure 2.3 is simply the node (P_0), which is included in the figure.

In the first mode of analysis gathering, we have one of the three potential ACGs with a root node of `Hoe`. The choice among the three is based on the calling context of `Hoe`, mainly the type of the parameter P_0 . In the first mode of analysis, the exact type is known and the ACG will be one and only one of the three ACGs shown in Figure 3.2.

The corresponding iCDGs for the ACGs shown in Figure 3.2 are shown in Figure 3.4. We reduce the iCDGs in Figure 3.4 by removing the *super nodes* from the graphs and replacing them with their corresponding edges and the resulting graphs are in Figure 3.5(a)-(c).

In Figure 3.5(a), the object instances created at line 14 and 21 are *method local* and stack allocatable. The object instance created at line 11, however, is found to be *method escaping* due to the *global write* node, ($g, 32$), now attached to it. This property was propagated through the iCDG by first forming the new edge between the object instance

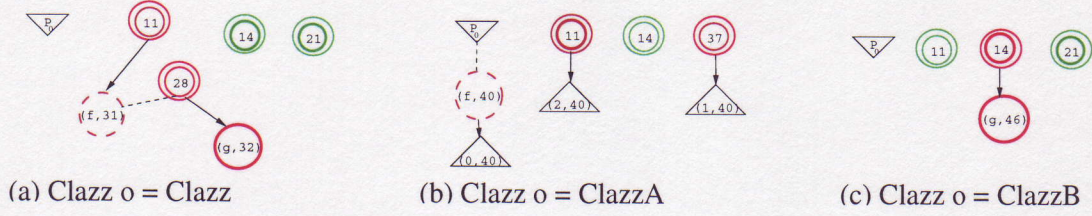


Figure 3.5 Results from removing the *super nodes* from the iCDGs in Figure 3.4.

created at line 11 and the field access occurring at line 31, then propagating the write to a global at line 32 through all edges reachable from the global node.

When the runtime type of `Clazz o` resolves to `ClazzA` in Figure 3.5(b) the solution changes. The speculative nodes at line 40 in Figure 3.2(b), cause only the object instance at line 14 to be found as method local and stack allocatable. The other object instances allocated within the iCDG are *unknown path* escaping due to the presence of the speculative node in the ACG. This presence is reflected in the iCDG in Figure 3.5(b) by the parameter nodes remaining in the graph.

The allocation decision changes again when the runtime type of the object instance is `ClazzB`, as can be seen in Figure 3.5(c). Here the objects created at lines 11 and 21 are found to be *method local* and stack allocatable.

We present an actual implementation of our framework for first mode, profile time method local analysis in Chapter 4.

3.4.2 Synchronization removal

In Java, synchronization is based on acquiring and releasing the monitor associated with a particular object instance or the monitor associated with a class instance. The monitors are single threaded, meaning that only one thread can own a monitor at any given moment. A monitor must be acquired upon entry to a procedure if the procedure is declared with the `synchronized` keyword. It can also be acquired explicitly in the code by use of the `synchronized` block structure. To avoid cluttering the libraries and confusing programmers with synchronized and unsynchronized versions of the files, the Java runtime libraries made several library calls thread safe. In making the libraries thread safe, they include the safest level of synchronization necessary to limit accessibility by multiple threads in any potentially critical region. Additionally, Java packages produced by *Independent Software Vendors* (ISV), may also be identified as thread safe and include the additional synchronization operations.

Although alleviating the programmer of some of the responsibility when writing threaded applications, the thread safe guarantees can add unnecessary overhead to an application in the form of unneeded or redundant synchronization. Redundant synchronization occurs from attempts to reacquire an already owned monitor for an object instance, through several nested layers of invocations. Although Java specifies that if a given thread already owns a given particular monitor and executes another synchronization block using the same monitor, then it does not reacquire the lock and must not block in the attempt, the nested attempts still add overhead. Therefore, since only one

thread is allowed to synchronize on (i.e., own the monitor for) a given object instance at any given point in the execution, the reacquisition of an already owned monitor can be considered superfluous. We also refer to these redundant synchronization blocks as *nested* synchronization. Additionally, it is possible for a thread to synchronize on a thread local object instance, meaning an object instance visible to only one thread. Since no other thread has access to the object instance, no other thread can contend for the monitor associated with it, and the synchronization is unneeded.

It is important to note that the identification of the locking operations as superfluous (unneeded or redundant) is based on recent developments within the Java specifications. In the original Java memory model, the acquisition of a monitor was tied to a memory barrier forcing a thread to synchronize its view of memory with that of the main memory. Therefore, the above synchronizations, although superfluous, had side effects that made their removal unsafe. This model is currently being revised and the current proposal only requires that the memory changes be visible to any thread that locks the *same* monitor [32] - [34]. Under the proposed model, the side effects originally associated with the *nested* and *thread local* synchronization operations are gone and the locking actions are indeed superfluous. In the old model, if there were thread escaping variables used by the thread with the identified superfluous locking operations, the synchronization operations could not be fully eliminated, still requiring a memory barrier and main memory update.

The identification and elimination of superfluous synchronization can occur at either the two modes. During the first mode, the ability to know dynamically that a monitor associated with an object is already held by a given thread, can be used to skip *nested*

synchronization operations. This property is tractable with the information contained within the CDG. The locking information is contained within the CDG in the form of the property nodes attached via association edges to the entries. This information can also help the analyzer to determine swiftly that an object instance is *thread local* and will remain *thread local* within a given synchronization region. Based on this, these synchronization operation can be eliminated.

For example, Figure 3.6 shows a synchronized version of the class file introduced in Figure 2.3. The CDGs for the synchronized methods in this class are shown in Figure 3.7. The difference between these CDGs and the ones for the unsynchronized version of the class file shown in Figure 3.3(a) is the addition of the *locking* and *unlocking* property nodes. These nodes are connected via association edges. Figure 3.8 shows the iCDGs both before and after super node removal. The synchronization operations identified as superfluous are the green highlighted property nodes in Figure 3.8(b), which are the four center property nodes attached to the formal node P_0 and the two property nodes attached to the allocation node 21.

The synchronization pair, $\{(L, 10), (U, 18)\}$, acquire and release the monitor associated with the object instance represented by node P_0 (Figure 3.8(a)). Therefore, the monitor pairs $\{(L, 20), (U, 26)\}$ and $\{(L, 27), (U, 33)\}$ are nested and superfluous. Referring back to the source code for the method `Hoe` in Figure 3.6, these pairs correspond to the calls to `Foo` and `Hoe`, respectively. For the superfluous *thread local* synchronization pair, $\{(L, 23), (U, 25)\}$, the determination is based on the fact that the object instance for


```

1: class syncClazz {
2:   syncClazz f;
3:   static syncClazz g;
4:
5:   <syncClazz>(syncClazz o){
6:     o.<object>();
7:   }
8:
9:
10:  synchronized void Hoe(syncClazz o){
11:    syncClazz a = new syncClazz();
12:    a.<syncClazz>();
13:
14:    syncClazz b = new syncClazz();
15:    b.<syncClazz>();
16:
17:    o.Foo(a, b);
18:  }
19:
20:  synchronized void Foo(syncClazz o, syncClazz p, syncClazz q){
21:    syncClazz r = new syncClazz();
22:    r.<syncClazz>();
23:    synchronized(r){
24:      o.Bar(p, q);
25:    }
26:  }
27:  synchronized void Bar(syncClazz o, syncClazz x, syncClazz y){
28:    syncClazz z = new syncClazz();
29:    z.<syncClazz>();
30:
31:    z.f = x;
32:    g = z;
33:  }
34: }

```

Figure 3.6 Synchronized version of the class file from Figure 2.3.

which the monitor is acquired and then released, is not thread escaping and is therefore inaccessible by another thread.

It is important to note that in general, the iCDG alone is not sufficient to determine *nested* synchronization information. Although the iCDG contains all the *locking* and *unlocking* information attached to the object instance associated with it, control flow information is still needed to definitively determine that a synchronization pair is

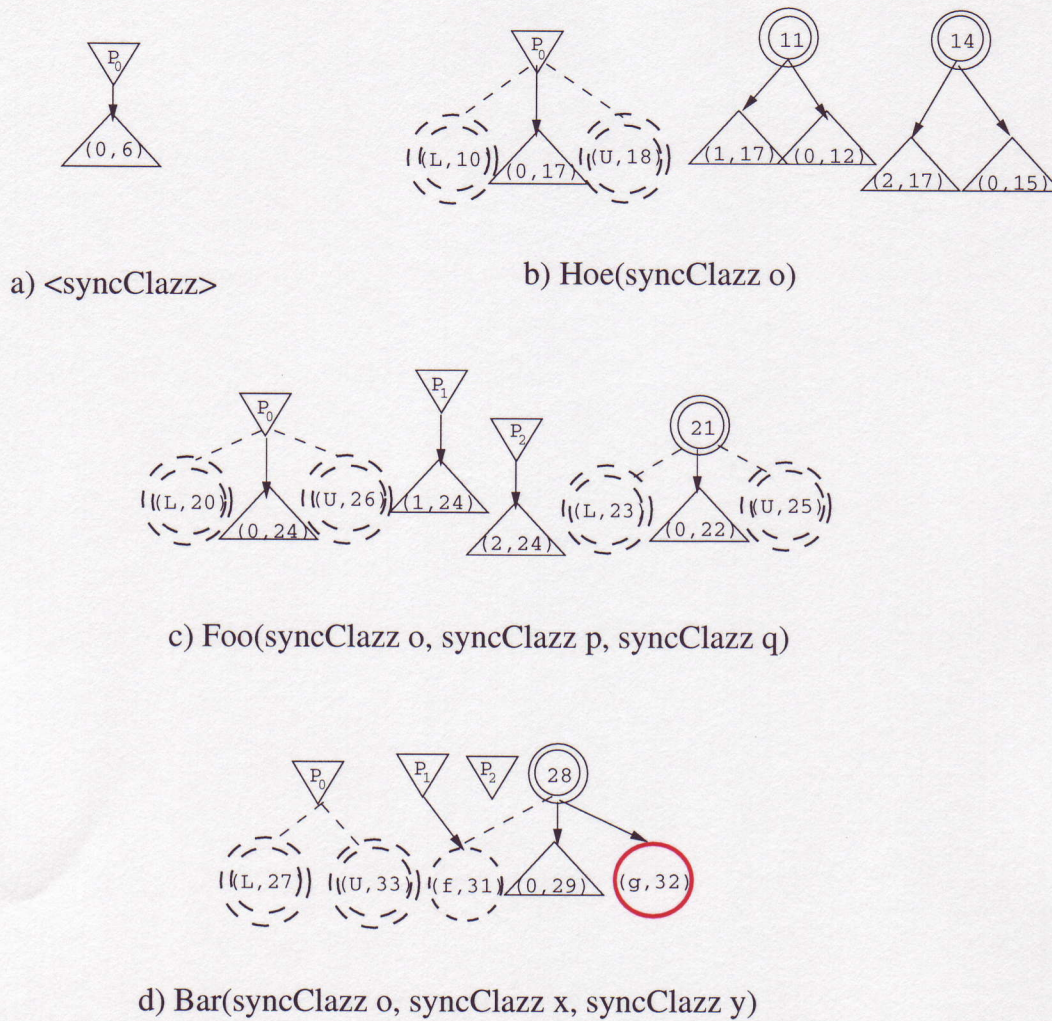


Figure 3.7 CDGs for the class file shown in Figure 3.6.

fully enclosed within another pair. Therefore, the iCDG is an enabling tool for this determination and not the only tool necessary.

3.4.3 Caching of redundant callee results and call elimination

In object oriented languages, sometimes procedures are used simply to obtain the value of a field of an object instance. For example, the call to `size` in the library class


```

1:  class ListClazz{
2:      ClazzNode C;
3:      int s;
4:
5:      <ListClazz>(ListClazz o){
6:          o.s = 0;
7:      }
8:
9:      void add(ListClazz o, ClazzNode N){
10:         if(o.C == null)
11:             o.C = N;
12:             o.s = 1;
13:         }else{
14:             o.C.add(N);
15:             o.s = o.s + 1;
16:         }
17:     }
18:     int size(ListClazz o){
19:         return o.s;
20:     }
21:     ClazzNode getElement(ListClazz o, int i){
22:         if(i < 0 || i >= o.s){
23:             return null;
24:         }else if(i == 0)
25:             return o.C;
26:         else
27:             return o.C.getElement(i);
28:     }
29: }
30:
31: class ClazzNode{
32:     object D;
33:     ClazzNode n;
34:
35:     <ClazzNode>(ClazzNode o){
36:         o.<object>();
37:     }
38:
39:     void add(ClazzNode o, ClazzNode x){
40:         if(o.n == null)
41:             o.n = x;
42:         else
43:             o.n.add(x);
44:     }
45:
46:     ClazzNode getElement(ClazzNode o, int i){
47:         i = i - 1;
48:         if(i == 0)
49:             return o.n;
50:         else
51:             return o.n.getElement(i);
52:     }
53: }

```

Figure 3.9 Example list class.


```

55:  class ListClazzUser{
56:      void append(ListClazzUser o,
57:                  ListClazz A, ListClazz N){
58:          for(int i = 0; i < A.size(); i++){
59:              ListClazz X = A.getElement(i);
60:              N.add(X);
61:          }
62:      }
63:
64:      ListClazz ListCopier(ListClazzUser o, ListClazz A){
65:          ListClazz B = new ListClazz;
66:          <ListClazz>(B);
67:          for(int i = 0; i < A.size(); i++){
68:              ClazzNode N = new ClazzNode;
69:              <ClazzNode>(N);
70:              ClazzNode M = A.getElement(i);
71:              N.D = M.D;
72:              B.add(N);
73:          }
74:          return B;
75:      }
76:
77:      ListClazz ListFlattener(ListClazzUser o, ListClazz C[]){
78:          ListClazz R = new ListClazz;
79:          <ListClazz>(R);
80:
81:          for(int i = 0; i < C.length; i++){
82:              ListClazz T = o.ListCopier(C[i]);
83:              o.append(T,R);
84:          }
85:          return R;
86:      }
87:  }

```

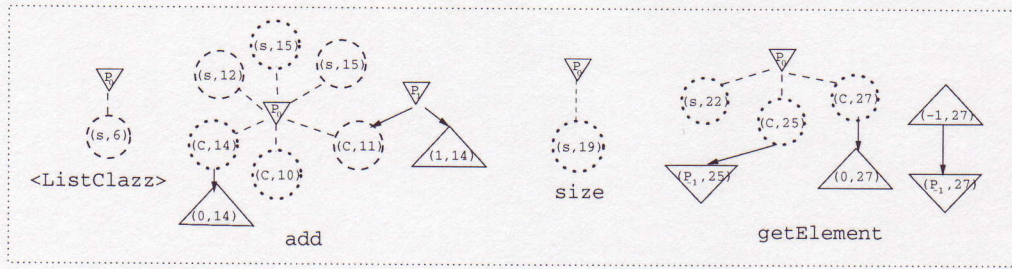
Figure 3.10 User class for the list class shown in Figure 3.9.

The `ListClazz` contains elements of type `ClazzNode` also shown in Figure 3.9. `ClazzNode` is a simple linked list node containing a data field, `D`, and a pointer to the next node, `n`. Both class files have definitions for the same two methods, `add` and `getElement`. When the method `add` is called with an object of type `ListClazz`, it proceeds to call the method `add` with an object of type `ClazzNode`. The method `add` in `ClazzNode` is recursive, calling itself with subsequent elements of the list until the last element is found, adding the new element to the end of the list. The method `getElement` is similar in its calling pattern with the version in `ListClazz` calling the version in `ClazzNode`.

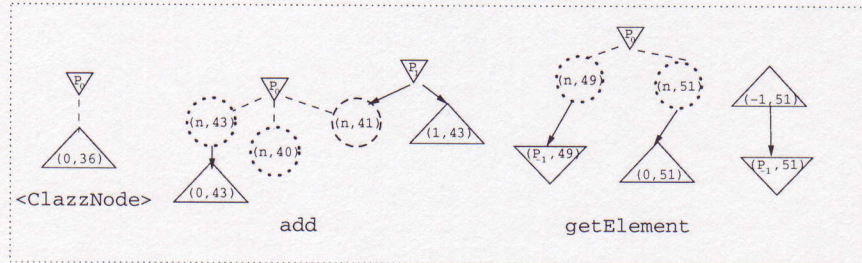
To illustrate the use of the iCDG/ACG for determining whether or not the value returned by `size` can be cached and reused, we introduce a user class for the list class. This user class, shown in Figure 3.10, contains three methods: the method `append` which appends the elements in one list onto another list, the method `ListCopier` which creates a new list then copies the elements of the original list onto the new list, and the method `ListFlattener`, which takes an array of lists and creates one long list out of them by calling the other two methods. The CDGs from the intraprocedural analysis results for these three class files, `ListClazz`, `ClazzNode`, and `ListClazzUser`, are shown in Figure 3.11.

The interprocedural analysis builds the ACG/iCDG iteratively following the steps described in Section 3.3. We walk through this process starting with `ListFlattener` as our root procedure for the ACG and adding the CDGs for the known nodes at each step. Figure 3.12 shows the ACG and corresponding iCDG starting from the root procedure `ListFlattener`. Note that in this figure, only the procedure `ListFlattener` and the initializer `<ListClazz>` are known nodes in the ACG and therefore only these two CDGs are incorporated into the iCDG. Note also that we have assumed that P_0 , used to resolve `ListFlattener`, is the exact type `ListClassUser` and not a subclass. This is important since from the iCDG we notice that P_0 is used for the resolution of `append` and `ListCopier`. In order to promote these nodes to known within the ACG, the type of P_0 must be definitively known.

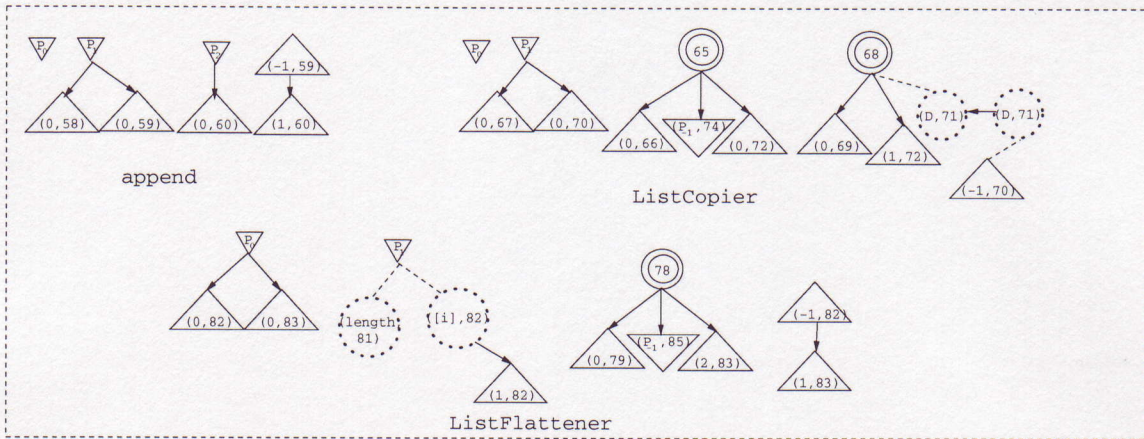
Making the assumption that P_0 is definitively known, we promote `append` and `ListCopier` to known nodes in the ACG and incorporate their CDGs into the iCDG. This new



(a) CDGs for class ListClazz



(b) CDGs for class ClazzNode



(c) CDGs for class ListClazzUser

Figure 3.11 CDG formation from the class files in Figures 3.9 and 3.10.

version of the ACG/iCDG is shown in Figure 3.13. Note that we have also incorporated the CDGs for the known nodes in the ACG corresponding to the initializer calls.

The speculative node add in the ACG of Figure 3.13 is resolved using the object allocation node (65) and therefore definitively known by definition. To continue pro-

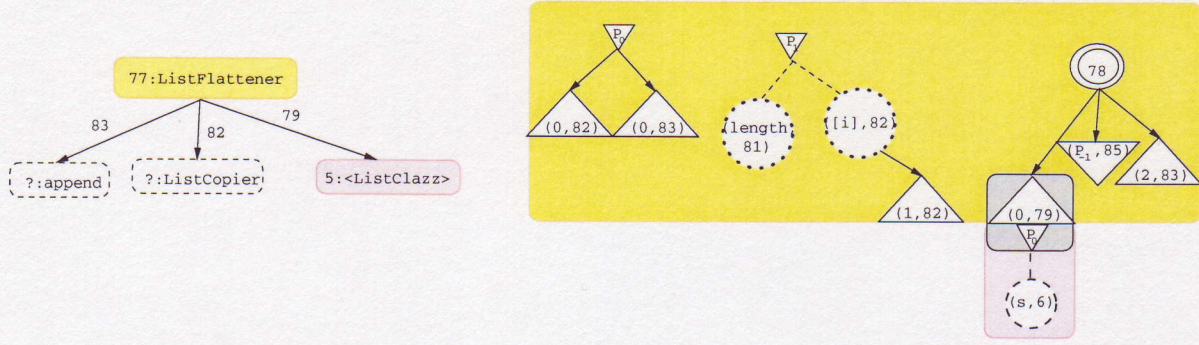


Figure 3.12 Initial ACG and corresponding iCDG for ListFlattener.

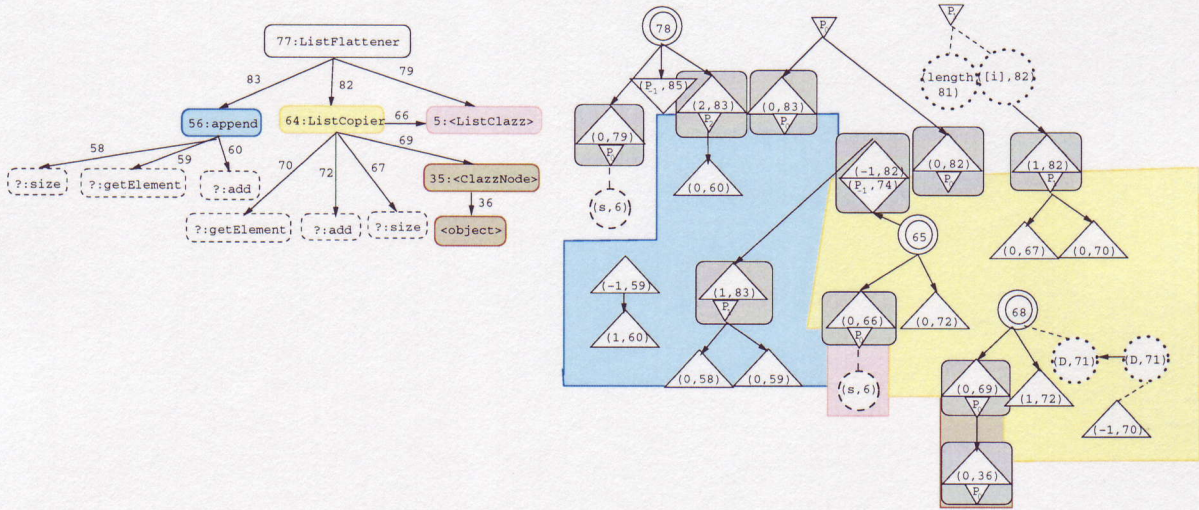


Figure 3.13 ACG and corresponding iCDG after resolving ListCopier and append.

moting nodes, we make an additional assumption that the type of parameter P_1 of ListFlattener is also definitively known. This is important since from the iCDG in Figure 3.13, we observe that P_1 is used to resolve the speculative nodes, `size` and `getElement`. With the assumption concerning node P_1 and the observation concerning node (65), we promote the nodes `getElement`, `add`, and `size` connected to ListCopier,

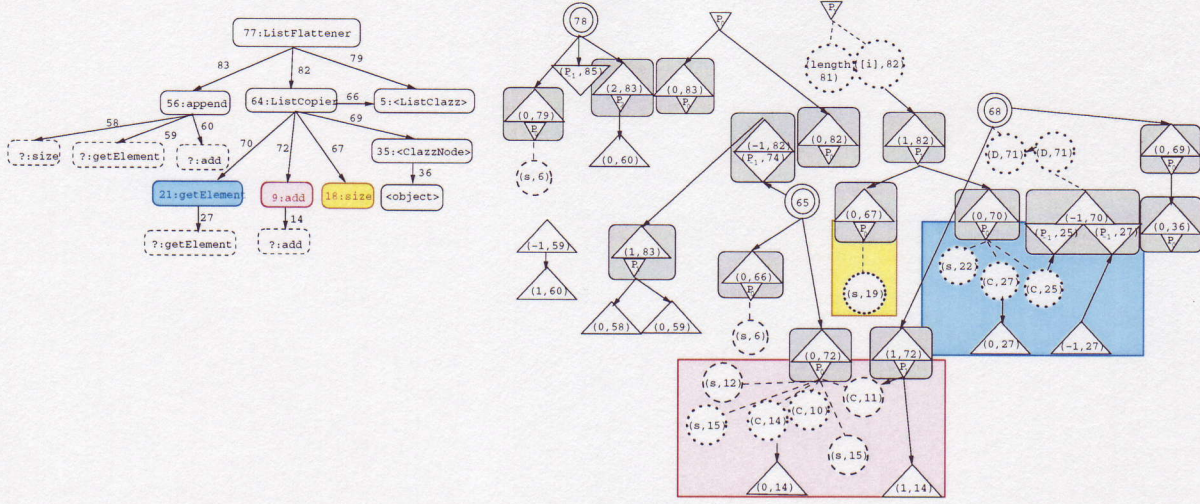


Figure 3.14 ACG and corresponding iCDG after resolving `getElement` and `add`.

to known nodes. We then incorporate their CDGs into the iCDG. The new ACG/iCDG is shown in Figure 3.14.

We next observe from the iCDG in Figure 3.14 that the node used to resolve `size` and `getElement` in `append` is the allocation node (65). Furthermore, the node used to resolve `add` from `append` is the allocation node (78). Since these are both allocation nodes, the types are definitively known and the speculative nodes, `size`, `add`, and `getElement` can be upgraded to known in the ACG. We have performed this upgrade and incorporated their CDGs into the iCDG in Figure 3.15.

To continue promoting nodes, we again make a type assumption. We assume the type of the field `C` accessed from `P0` is the exact type `ClazzNode`. This allows us to promote the speculative nodes `getElement` and `add` to known and incorporate their respective CDGs into the iCDG. This new iCDG is shown in Figure 3.16. This then leaves two speculative

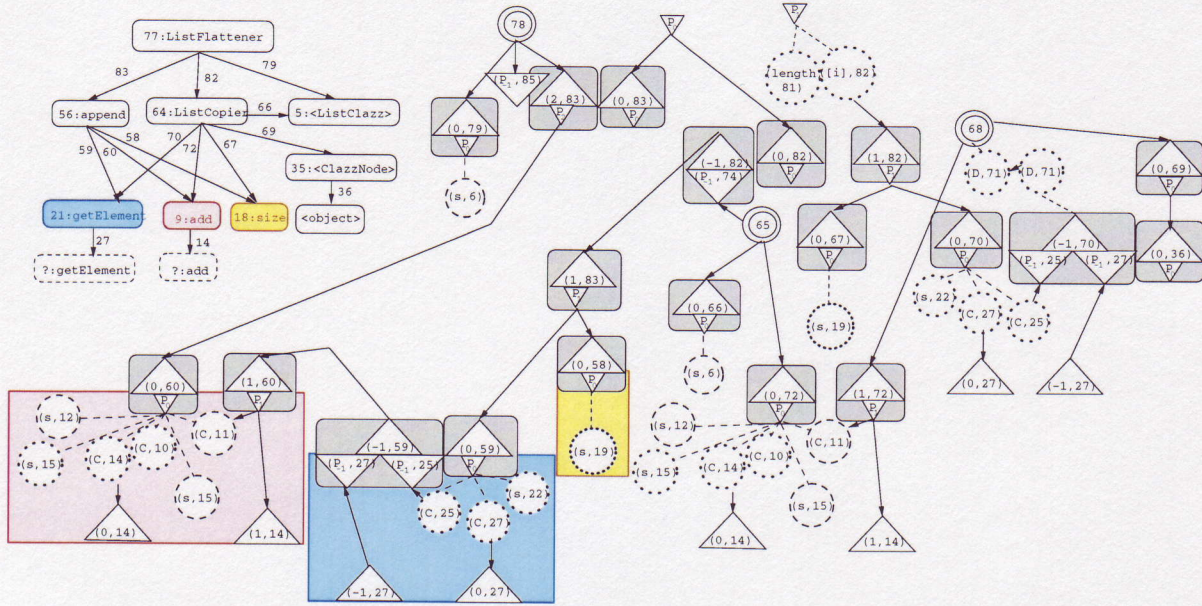


Figure 3.15 ACG and corresponding iCDG after resolving the next tier `getElement` and `add`.

nodes in the ACG. For these two nodes, again we must make a type assumption, i.e., the field `n` of parameter node P_0 is the exact type `ClazzNode`. With this assumption, we can connect these two nodes as recursive nodes in the ACG and corresponding iCDG, as shown in Figure 3.17. Since there are no more speculative nodes within the ACG and all of the known nodes have been incorporated into the iCDG, this is also the final graph.

For the caching of the result return from `size` for the call to `append` made in the method `ListFlattener` at line 83, we need to determine where the object associated with parameter node P_1 is originating from. We have highlighted this node in the final iCDG in Figure 3.17. From this graph, we determine that the object passed as parameter P_1 is from the allocation node (65). Once the object definition is established, we then need to determine two items about it, its escaping state and when the field `s` is written.

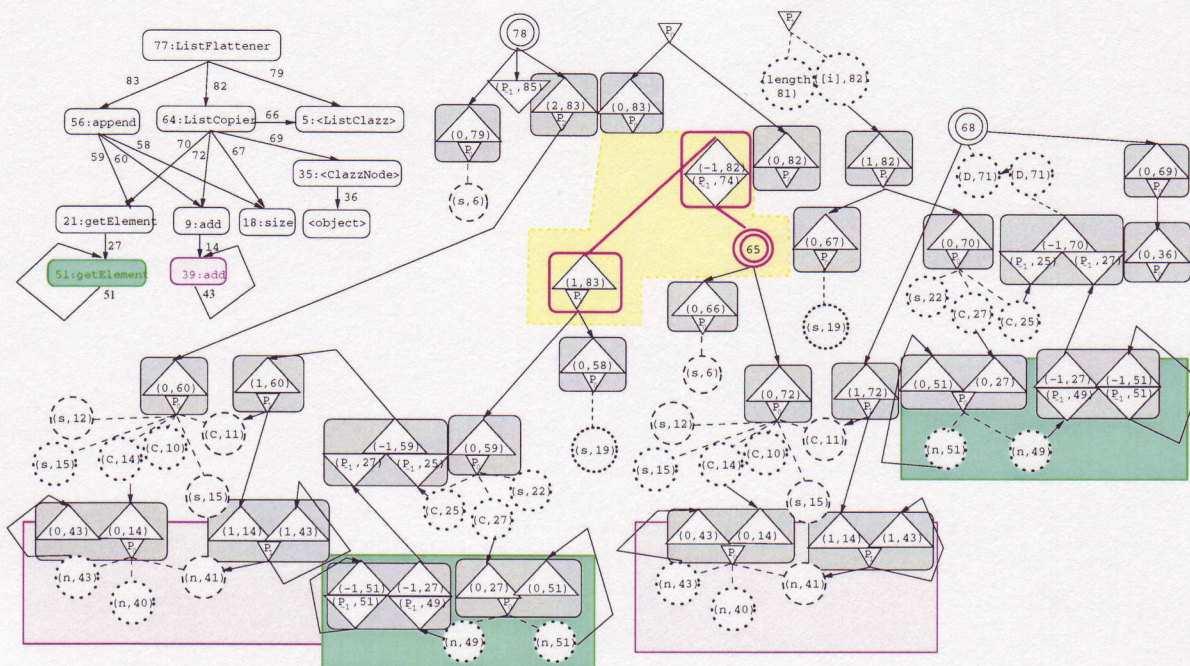


Figure 3.17 Final ACG and corresponding iCDG.

From Figure 3.19 we discover that the write nodes attached to the node (65), are (s,6), (s,12), and (s,15). Next we need to determine if any of these writes to *s* occur in the area where the cache result is of interest. The write nodes correspond to the writes in the initializer and the method `add`, neither of which is called from within the body of the for loop in `append`. From the escaping information and the change information, we can determine it is safe to cache the value returned by `size`.

There are several important items to note with this example. First, the promotion of several of the nodes in the ACG was based on type assumptions. Although in general type assumptions can prove unsafe, there are some cases where we can make these assumptions and validate that they do indeed remain valid as new class files get loaded. For example,

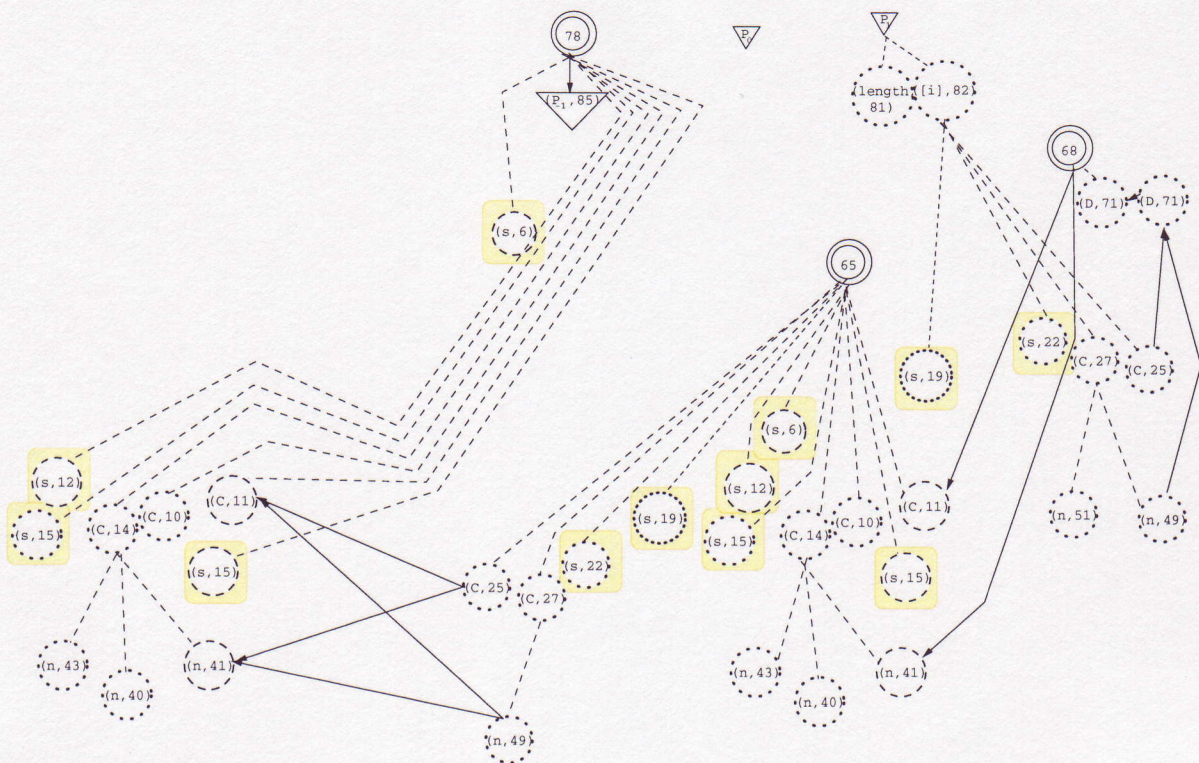


Figure 3.18 Final iCDG with super nodes removed.

if our class hierarchy only contains the class files used in this example and not any subclasses for them, then we could promote based on only one copy. However, if a new subclass should get introduced into the system, then the class hierarchy for this example could change and this assumption may no longer be valid. In Chapter 5, we address type assumption optimizations and the types of validation rollback and recovery necessary when such assumption invalidations occur. Additionally, the optimization we discussed as being enabled by the analysis requires that an optimized version of the entry method be created which inlines the callee methods. Therefore, the optimization of caching the

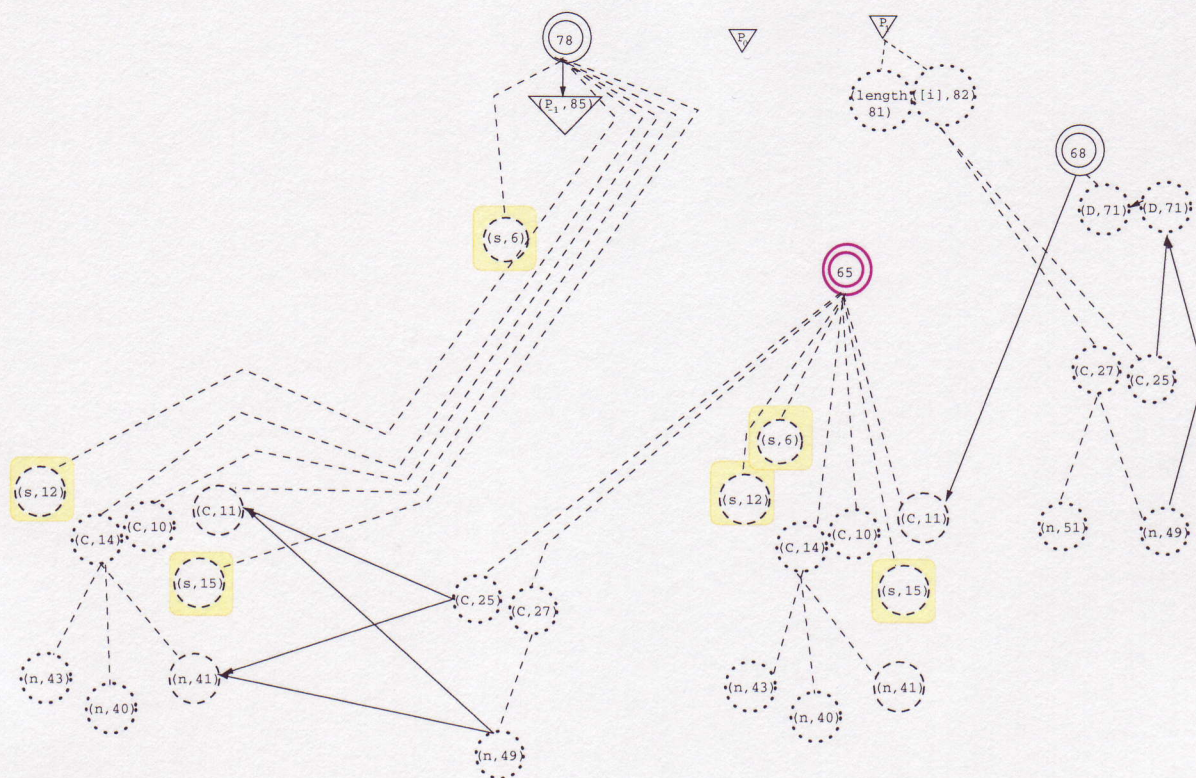


Figure 3.19 Final iCDG with field `s` reads removed.

value returned by `size` is only valid in the inlined version. We discuss these types of optimizations and several optimization models further in Chapter 5.

3.4.4 Lifetime-based optimizations

Lifetime-based optimizations are optimizations that recognize that even though an object instance is no longer thread local, there is a portion of its lifetime prior to it escaping in which thread local optimizations are still valid. This form of optimization uses the iCDG information to augment flow-sensitive information provided by the *Control Flow Graph* (CFG) or other optimization structures.

The primary difference between the analysis results needed for this type of optimization versus the analysis results used for the *method local* and *thread local* determination, is its use of flow sensitive information requiring more than just the iCDG. For example, in *thread local* or *method local* determination, the analyzer may just be interested in a “safe” solution. The solution is therefore based on whether or not the object instance in question ever escapes during its lifetime. The answer therefore can use bidirectional propagation and propagate an escaping property along the reverse direction of a data flow edge. The solution is then “safe,” but not flow sensitive. For lifetime-based optimizations, the analyzer needs to consider not only the escaping state but when within the code stream the property changed. Therefore, information such as a *global write*, which changes the state of an object instance, should only be propagated along forward data flow links to preserve the temporal information implicitly contained within the iCDG.

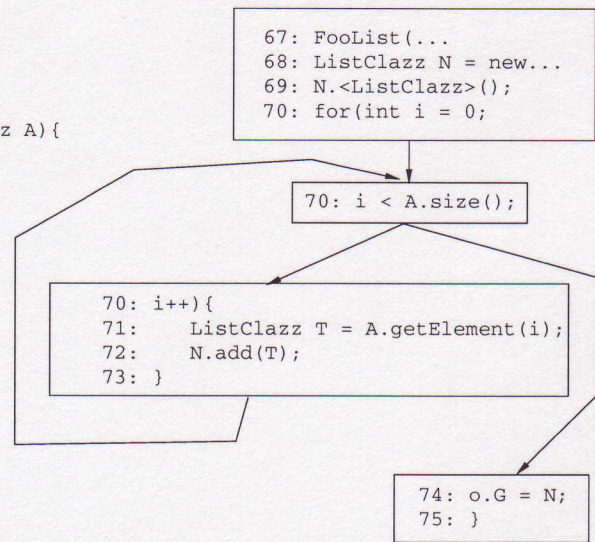
As an example, we introduce another version of the class `ListClazzUser` in Figure 3.20. This new class, `ListClazzUser2`, uses the other two class files shown in Figure 3.9, `ListClazz`, and `ClazzNode`. It varies from the other version in that it contains a global field `G` on line 65. The ACG for the procedure `FooList` is shown in Figure 3.21. The only difference between this ACG and the one for `append` shown in Figure 3.11 besides the line numbers and procedure name is the inclusion of call to the initializer, `<ListClazz>`. The CDG for `FooList` is shown in Figure 3.22, the *super node* connected iCDG is shown in Figure 3.23, and the *super node* removed version of the iCDG is shown in Figure 3.24.

Again we would like to determine if it is safe to cache the value returned by the call to `size` within `append`. To determine the escaping state of the allocation node (68)


```

64: class ListClazzUser2{
65:     static ListClazz G;
66:
67:     void FooList(ListClazzUser2 o, ListClazz A){
68:         ListClazz N = new ListClazz();
69:         N.<ListClazz>();
70:         for(int i = 0; i < A.size(); i++){
71:             ListClazz T = A.getElement(i);
72:             N.add(T);
73:         }
74:         o.G = N;
75:     }
76: }

```



CFG for FooList

Figure 3.20 Another version of the class ListClazzUser that uses the classes ListClazz and ClazzNode shown in Figure 3.9.

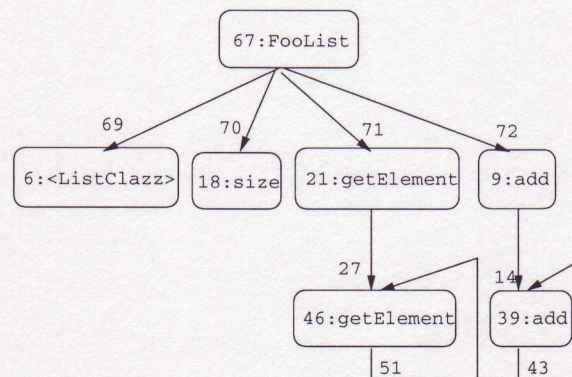


Figure 3.21 The ACG for the procedure FooList shown in Figure 3.20.

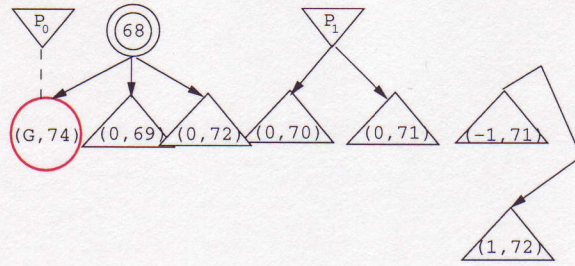


Figure 3.22 The CDG for the procedure `FooList` shown in Figure 3.20.

and whether it changes the value of its field `s` within the region under analysis, we need the iCDG and control flow. The region under analysis is first determined from the CFG for the method `FooList`, shown on the right of Figure 3.20. From the CFG, it can be determined that the region in which the value returned by `size` would need to remain unchanged is lines 70-73. This determination is made similarly to the one in the previous section, with the only writes to `s` occurring in the initializer and the call to `add`. Neither of these writes are within the bounds of the current analysis. The determination of the escaping state of the allocation node (68) is again based on the region under analysis. Although this node does indeed escape as observed by the strong edge connecting it to the global node `(G, 74)`, this event does not occur within the analysis region. However, exclusion from the region is not sufficient. We also must determine if the event occurs either prior to or after the analysis region. From the CFG, we can determine that the strong connection to node `(G, 74)` occurs after the region in question, and therefore it is safe to still cache the value returned by `size`.

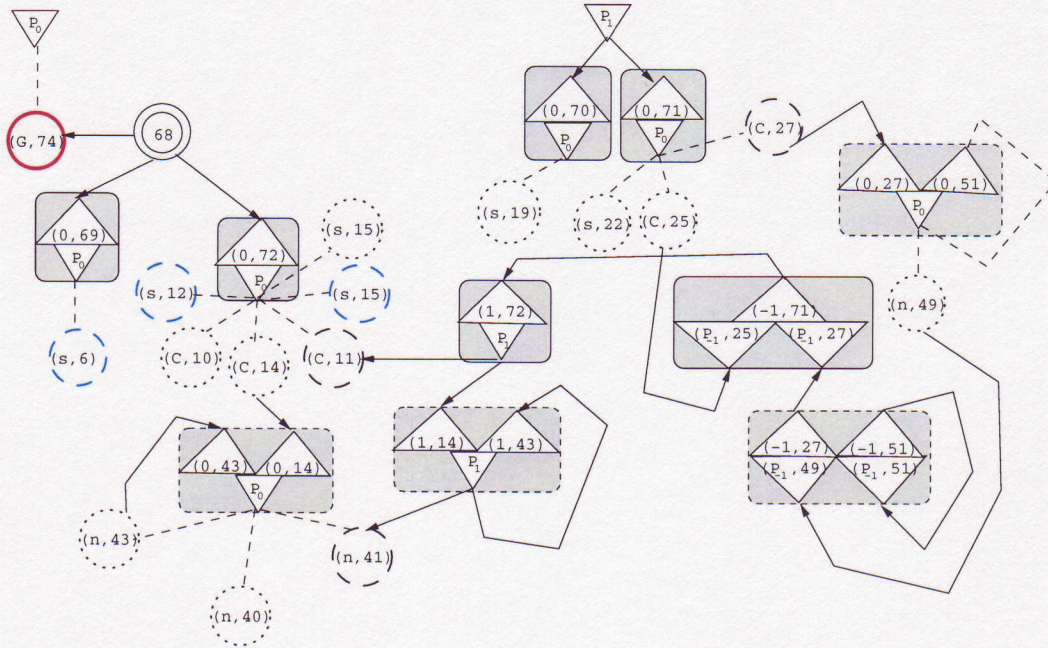


Figure 3.23 The resulting iCDG from the interprocedural propagation in FooList.

In general, the iCDG/ACG is a helping tool for this form of analysis. Note that the traditional CFG and flow information was still needed. Furthermore, we have made assumptions similar to those made in the prior analysis concerning known types which also require additional validation of these assumption when new class files are loaded.

3.4.5 Memory layout for better data locality

The relationship between object instances can be used to provide hints to the memory manager for potentially better memory layout. If two object instances are shown to be connected in some fashion, and one object instance contains a reference to the other, it may become desirable to place the two object instances next to each other in memory.

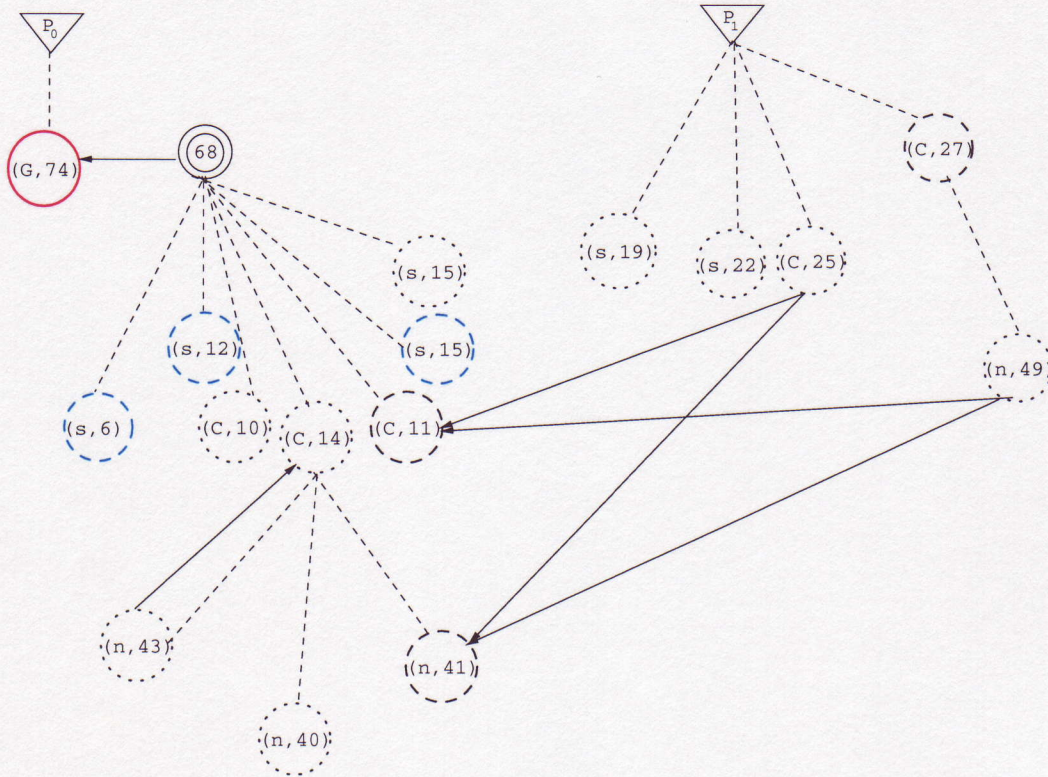


Figure 3.24 The resulting iCDG after *super node* removal.

This information is contained within the CDG as the relation between object instances via the association edges.

This analysis can be conducted during both of the two modes. For example, referring back to the class `Clazz` shown in Figure 2.3, the new object created at line 28 has a field connection to the object created at line 11. This information could be used by the memory manager to allocate these two object instances next to each other. Normally this level of information would not be available to the memory manager. However, looking at the interprocedural graph for `Hoe` shown in Figure 3.5(a), the connection between the two allocations is clear. Even though these object instances were found to be method

escaping, the layout information can be used to guide the heap layout and improve memory performance.

3.4.6 Code motion after inlining

Often, after inlining callees within a method being optimized, it is desirable to move instructions above other instructions, thus reordering the original execution order. This movement can facilitate several optimizations such as eliminating duplication of work on multiple iterations through a loop, or allowing other instructions to execute sooner, thus reducing or eliminating potential stalls. To determine if it is safe to move an instruction, the optimizer must have results from several forms of analysis, including flow information and points-to information. The information contained in the CDG is helpful to the optimizer since a points-to relation can be extracted from it and escaping information is also present. However, using the CDG to determine the connections between objects and their escaping states is not a separate form of analysis but rather an application of analysis results during the optimization of the code. Therefore, this particular application of CDG information is covered in Chapter 5.

3.4.7 Unsafe sharing, potential race determination

The information in the CDG can be viewed as the first step in implementing a race detection system. The CDG contains all the necessary information for the propagation of locking properties, escaping states, and flow information. The state information contained within the CDG is a first step in the solution for full race detection. The detection

of races also needs temporal information such as when the locks are held and by which thread. This form of detection in a dynamic application may need to be conducted as the application is running since newly loaded classes can change the results of previous detections, thus making a safe access become a data race. The detection process itself may interfere with the application in such a way as to change an assumed timing and cause incorrect results. Even though the assumed timing is actually a masked data race, for server-side applications where remaining up is the most critical component, the exposure and potential premature termination of such a race could be problematic. The problem of race detection and its correction is complex, and therefore the iCDG/ACG is only part of the solution, the rest remaining part of future work.

CHAPTER 4

SPECIFIC APPLICATION: OBJECT CONNECTION GRAPHS

In Chapter 2 we presented the Compact Dataflow Graph (CDG), which summarizes the use of reference values within a procedure. In Chapter 3 we presented the design of the dynamic interprocedural analysis engine, which employs an Adaptive Call Graph (ACG) in combination with the individual CDGs to create an interprocedural CDG (iCDG). We showed how the iCDG can be used to provide a wide range of analysis results. In this chapter, we present an implementation of our framework to perform a subset of analysis outside the range of other dynamic analysis systems.

4.1 Problem Description

Several researchers have shown that if the worldview of a Java application is assumed closed and no further class files will be loaded, full program analysis can identify as much as 94% of the object instances used in the application as being stack allocatable [15], [25]. Furthermore, by stack allocating these object instances, these researchers have shown that speedups as high as 44% can be achieved [15]. The analysis, referred to as static analysis since it assumes a closed-world, static view of the application, must

determine the safe set of method local object instances based on traditional analysis techniques. The determination of a safe set of method local object instances requires the identification of all object instances that have the potential to exceed the lifetime of their allocating method or are method escaping. Although other researchers have demonstrated through static analysis that a large portion of the object instances used within an application are indeed method local, Java is dynamically loaded and linked. Therefore, static full application analysis is not available.

Some researchers have suggested that the static analysis techniques could be employed in a dynamic application during the runtime optimization phase. Even if a partial, safe solution is determined during the optimization phase (second mode), this solution can be invalidated when new class files are loaded. On top of the invalidation problem, delaying method local determination and this form of optimization until the optimizing phase could miss some potentially beneficial stack allocation opportunities. The loading and initializing of new class files can involve the use of temporary object instances that can be stack allocated. Initialization code tends to make liberal use of short-lived object instances. The same liberal use of short-lived object instances occurs in the early portion of an application's execution, prior to the first run of the optimizer. During this segment, a large portion of the application code is initialization code. Therefore, identification of stack allocatable object instances at the point of first allocation can help the runtime capture these optimization opportunities. Performance improvements from stack allocating these object instances include reducing if not eliminating the memory

manager overhead. Additionally, the stack allocation can delay garbage collection events by reserving the memory for potentially longer-lived method escaping object instances.

There are several obstacles with the identification of stack allocatable object instances using first mode analysis. First, all new object instances in Java call an initializer, meaning the analysis is interprocedural. Furthermore, Java does not restrict the actions performed by an initializer, making the determination of escaping state dependent on interprocedural analysis. Second, we cannot assume control flow information is available or even within a reasonable cost bound to compute. The computation of control flow information can become costly when factoring in the multiple paths of control and potential targets in polymorphic languages. Third, the determination of the escaping state of a particular reference value depends on points-to information concerning the references used within the region under analysis. Points-to information normally requires control flow information for its derivation. Fourth, in order to reap any benefit from the determination of stack allocatable object instances, the benefit of the optimization must outweigh the cost of the analysis. Since the maximum determined static analysis benefit was shown to be a speedup of 45% with an average of 24%, the analysis cost should not exceed the average benefit. All of these factors have made the determination of stack allocatable object instances using first mode analysis difficult and unattractive to other analysis systems. However, our framework makes this form of analysis efficient and attractive.

4.2 Our Solution

Our framework provides a very efficient solution to this analysis challenge. The CDG contains all the information concerning the use of reference values within a given method. This enables swift extraction of points-to information for each method. Second, our design of the ACG/iCDG system enables swift, efficient propagation of interprocedural analysis results. We can adapt this design to accommodate interprocedural propagation of points-to information as well. Although, control flow information is missing at the time of analysis, the CDG was constructed using flow-sensitive information. Therefore, we are able to still capture a large percentage of the potentially stack allocatable object instances using just the extracted points-to information and the efficient interprocedural propagation. Finally, using our design, we are able to keep the costs below 22% for the benchmarks we used, which is within the bounds set by the potential benefits of a 45% performance improvement.

4.2.1 Definition

In Chapter 3, we defined the properties used to determine if an object instance is method escaping as follows.

Definition 4 *An object instance is said to survive the lifetime of its allocating method, or to be method escaping, if one of the following events occurs during its lifetime:*

- *Global Escaping: A reference to the object instance becomes accessible for any portion of its lifetime, via a global field.*

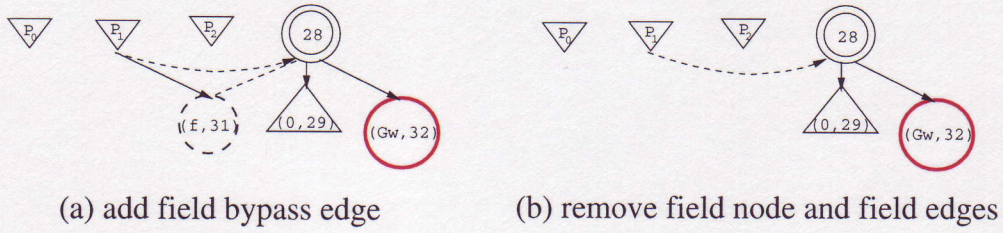


Figure 4.1 Reduced CDG example for the method Bar.

- *Reference Escaping:* A reference to the object instance is stored in the field of another object instance whose lifetime exceeds the allocating method.
- *Return Escaping:* A reference to the object instance is returned from its allocating method.
- *Unknown Path Escaping:* A reference to the object instance crosses an interprocedural boundary for which the current thread does not have access (native methods, methods without intraprocedural analysis results, passed in a call to another thread).

4.3 Extracting Points-To Information

Extracting a points-to relation from the CDG is straightforward. First, property nodes can be removed from the CDG since these do not represent dataflow. Second, field nodes can be eliminated by making the relationship between the two reference values a weak link, meaning they are not direct aliases for each other but one is reachable via information contained within the other. To illustrate, Figure 4.1 contains the resulting reduced CDG for the method Bar. This reduced CDG is formed by first adding a weak

edge between nodes P_1 and 28 and then removing the field node and its associated edges. Figure 4.1(b) shows the resulting CDG from this process. The final graph in Figure 4.1 is equivalent to a reverse points-to relation with the addition of the weak edges.

The weak edges are important when we later use the edge type to propagate class types for swift method resolution when forming the ACG. Type information does not propagate across weak edges. Therefore, if a weak edge is used to connect a parameter node that is used for the resolution of a callee, the callee cannot be promoted to a known node status in the ACG, and subsequently its intraprocedural information is missing. The inclusion of speculative nodes causes the object instances to be marked as method escaping due to the unknown path, and this affects the analysis results.

The next simplification to our points-to relations is intended to reduce the cost of the analysis. We choose in our representation to eliminate the direction from the edges. This simplification gives a conservative analysis result in the sense that it overestimated the number of escaping object instances. However, by eliminating direction, we are able to propagate information more efficiently while identifying between 47% and 61% of the the oracle method local object instances.

Figure 4.2 illustrates the conservative aspect of direction elimination. In this figure, we consider only strong edges, but analogous results occur with weak edges. The partial reduced CDG in Figure 4.2(a) shows the CDG from an object allocation site. This object allocation is passed interprocedurally as two different callee parameters. The potential callee graphs are shown in Figures 4.2(b) and (c). For the first interprocedural graph, Figure 4.2(d), if parameter P4 escapes, then we also mark F1 as escaping since these

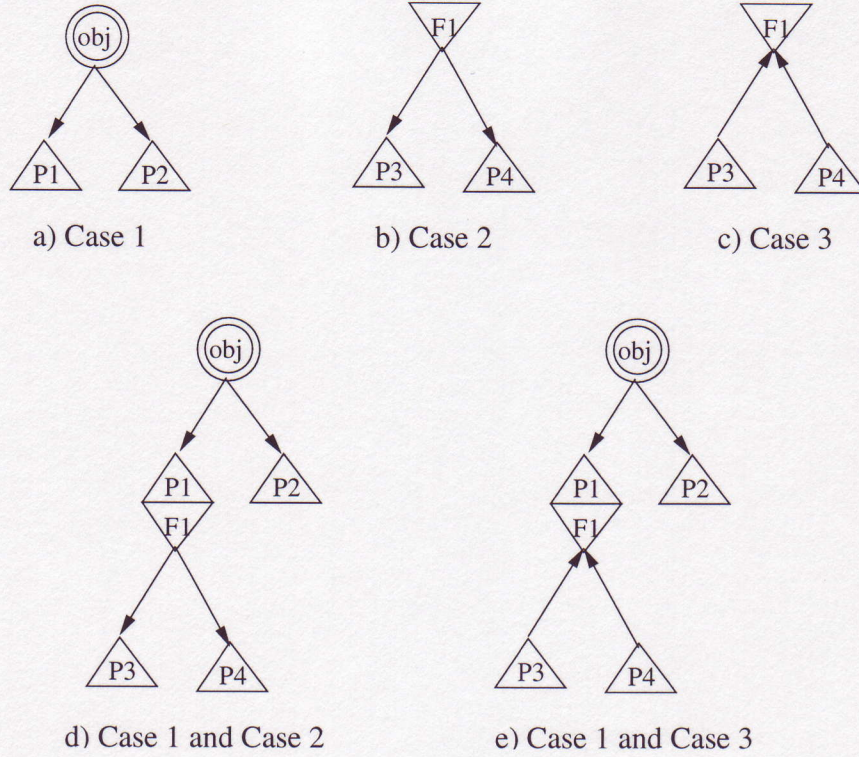


Figure 4.2 OCG propagation through the ACG for the type resolution `ClazzB`.

two are equivalent. This escaping state then propagates interprocedurally to P1 and obj. However, since we do not have control flow information, we do not know if the assignment to P3 occurred before or after F1 escaped. Therefore, P3 is also marked as escaping. Likewise for the assignment to P2. This case does not hold in Figure 4.2(e). In this case, if P4 escapes, then F1 also escapes. However, since P4 overwrote the reference originally in F1, this escaping information does not propagate any further. Likewise, since P3 overwrites F1, the lack of control flow information does not affect the propagation as in the previous example, and this object instance also remains unaffected. By eliminating the direction from the edges, we arrived at the correct solution for Figure 4.2(d) when

OCG	$:= (n^*, e^*)$
n	$:= (j, l) (Gw) (Gr) (l) (P_i) (P_{-1}, l)$
e	$:= \text{an undirected edge connecting node } n_i \text{ with node } n_j$ <i>either strong: $(n_i) - (n_j)$, or weak: $(n_i) - - (n_j)$</i>
(j, l)	$:= \text{the } j\text{th parameter to the method called at line } l$
$(-1, l)$	$:= \text{the return value from the parameter called at line } l$
(Gw)	$:= \text{a write access of a global variable}$
(Gr)	$:= \text{a read access of a global variable}$
(l)	$:= \text{an allocation occurring at line } l$
(P_i)	$:= \text{the } i\text{th formal parameter to the method } m.$
(P_{-1}, l)	$:= \text{the return value from the method } m \text{ at line } l.$

Figure 4.3 Definition of the Object Connection Graph.

propagating escaping information, but an overly conservative one for Figure 4.2(e), since we must mark all of the entries escaping. Although this solution is not as precise, it is still safe.

We call the new points-to relation that we extract from the CDG an *Object Connection Graph* (OCG) to distinguish it from a traditional points-to relation. Likewise, the corresponding interprocedural version is called an iOCG. The definition of an OCG is given in Figure 4.3. Note that line numbers have been removed from all but the allocation and parameter nodes. Although the line numbers aid in precision when used in combination with control flow information, the absence of this information makes their inclusion superfluous. The line number information retained in the allocation and parameter nodes aids in the identification of the exact allocation operation within the method as well as the appropriate call site for the ACG construction.

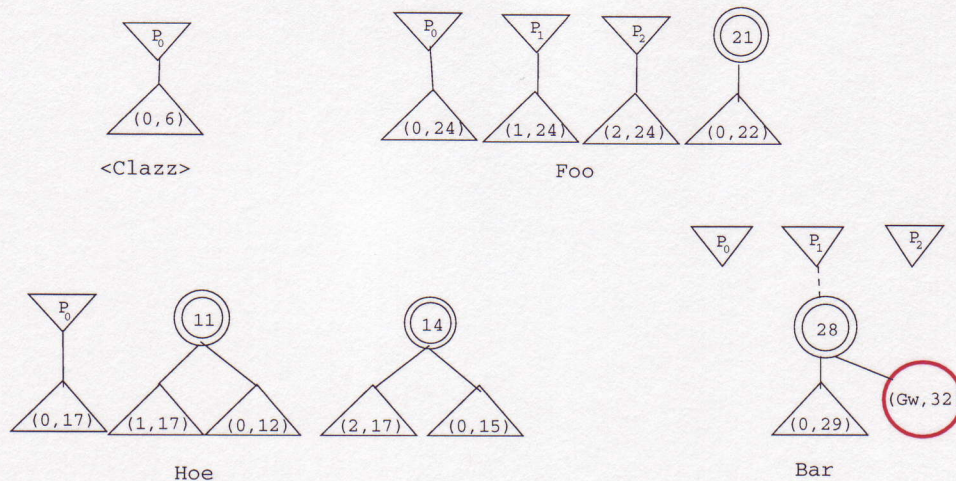


Figure 4.4 OCGs for Figure 2.3.

Figure 4.4 shows the OCGs for the CDGs in Figures 2.5 - 2.7. Note that for three of these OCGs – <Clazz>, Hoe, and Foo – their OCGs are almost identical to their corresponding CDGs with the only noticeable difference being the transformation of the directed edges into undirected edges. For the method **Bar**, the OCG is equivalent to an undirected form of the reduced CDG shown in Figure 4.1(b).

4.4 Interprocedural Propagation

The next step in the determination of stack allocatable object instances is the propagation of information interprocedurally. There are four basic determinations we can make based on the interprocedural propagation of the information contained within the iOCG:

- *method escaping* based on intraprocedural information,

- *method escaping* based on interprocedural information,
- *method local* based on interprocedural information,
- cannot be decided due to the presence of speculative nodes in the ACG.

All allocations call an initializer: thus, no object instance can be classified as method local based solely on intraprocedural information. Also note that weak edges or multiple strong edges in the OCG can cause speculative nodes to remain in the ACG. Object instances under analysis that pass into a speculative ACG node cannot be definitively determined based on the information present. Therefore, by Definition 4, we define these object instances conservatively as being unknown path escaping, making them method escaping.

The steps for forming the iOCG are almost identical to the steps for forming the iCDG. They are as follows:

1. Create root node in the ACG for the procedure being analyzed.
2. For each procedure called by the root node:
 - Add a *known* node for any provably monomorphic procedures.
 - Add a *speculative* node for all others.
3. Create the iOCG with the OCG for the root node and promote any *speculative* nodes to *known* nodes based on iOCG information and current context.
4. Form *super nodes* by connecting actual and formal parameters from the OCGs corresponding to the *known* nodes in the ACG, into the iOCG.

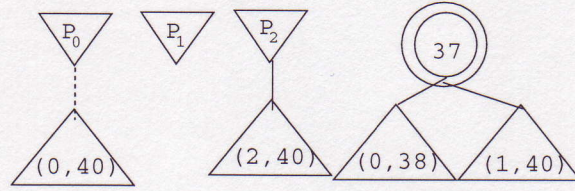
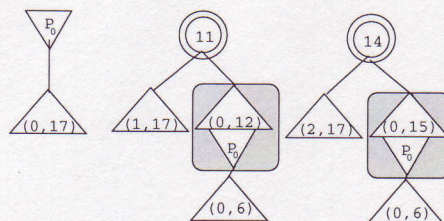


Figure 4.5 OCG for the subclass `ClazzA`.

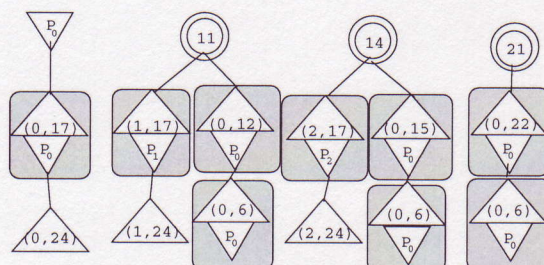
5. For each *known* node in the ACG not yet resolved, treat it as a *root* node and repeat the previous steps.

To illustrate the formation of the iOCG and the four resulting types of allocation decisions, we again use the example application introduced in Figure 2.3 with the added subclass, `ClazzA`, introduced in Figure 3.1. The OCG for the method `Foo` from the class file, `ClazzA`, from Figure 3.1 is shown in Figure 4.5. It was derived from its corresponding CDG shown in Figure 3.3 using the same method outlined in Section 4.3. Figure 4.6 walks through the formation of the ACG/iOCG for the case where type used to resolve `Hoe` is `Clazz`. Later, we present the results for type `ClazzA`. The ACG shown in Figure 4.6(a) begins with the node `Hoe`. Since `Hoe` calls an initializer and initializers are by definition also known, the ACG also contains a known node for `<Clazz>`. However, the node `Foo` is initially speculative. The iOCG is created by connecting the OCGs for `Hoe` and `<Clazz>`, the known nodes in the ACG. Examination of the iOCG reveals that a strong edge connects nodes P_0 and $(0, 17)$, meaning that the known type used to resolve `Hoe` is also used to resolve `Foo`, and likewise for the initializer `<Object>` called at line 6 in `<Clazz>`. In Figure 4.6(b), `Foo` has been promoted to known status and `<Object>`



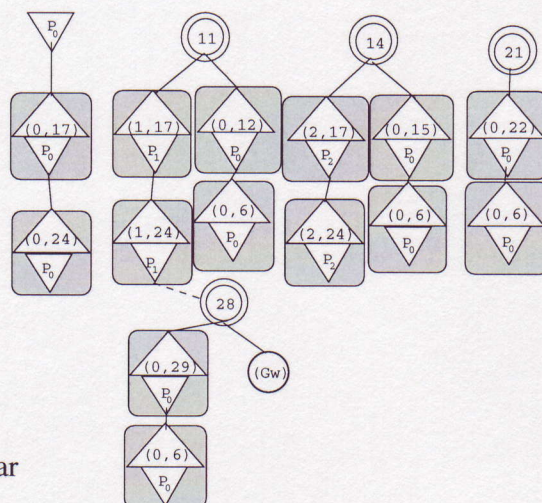
```

graph TD
    10Hoe["10:Hoe"] -- 17 --> 20Foo["20:Foo"]
    10Hoe -- "12, 15" --> 5Clazz["5:<Clazz>"]
    20Foo -- 24 --> Bar["?:Bar"]
    20Foo -- 22 --> 5Clazz
    5Clazz -- 6 --> Object["<Object>"]
    style Bar stroke-dasharray: 5 5
  
```

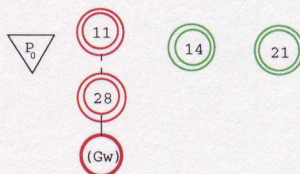


```

graph TD
    10Hoe["10:Hoe"] -- 17 --> 20Foo["20:Foo"]
    10Hoe -- "12, 15" --> 5Clazz["5:<Clazz>"]
    20Foo -- 24 --> 27Bar["27:Bar"]
    20Foo -- 22 --> 5Clazz
    27Bar -- 29 --> 5Clazz
    5Clazz -- 6 --> Object["<Object>"]
  
```



(c) After resolving Bar



(d) Results after super node bypassing and property propagation

85

has been added as a known node. The resolution of `Foo` has added another node to the ACG, `Bar`, which is speculative until the iOCG is examined. The OCGs for `Foo` and `<Clazz>` are then added to the iOCG shown in Figure 4.6(b). Note that there is only one copy of the node `<Clazz>` in the ACG since the ACG contains the unique methods called. However, the iOCG is context sensitive and represents each unique call to a method; therefore, a new copy of the OCG for `<Clazz>` is added to the graph. By examining the new iOCG, it is discovered that a strong edge connects P_0 from node (0, 24), meaning that the same known object instance type used to resolve `Foo` is now used to resolve `Bar`. This allows `Bar` to be promoted to known status and its corresponding OCG added to the iOCG. This is shown in Figure 4.6(c). Since all known nodes in the ACG have been resolved (step 4 in the iOCG/ACG formation algorithm) the formation of the iOCG/ACG terminates.

Next, the super nodes are replaced with direct edges, and the escaping state produced by the write into the global node is propagated up the graph. The final result shows that the objects allocated at lines 14 (from `Hoe`) and 21 (from `Foo`) are indeed method local and stack allocatable, while the objects allocated at lines 11 (from `Hoe`) and 28 (from `Bar`) are not. Note that the escaping information also propagates along the weak edges. Referring back to the four potential determinations, the escaping states of the object instances created at lines 14, 21, and 11, were determined by using interprocedural information. However, the escaping state of the object instance allocated at line 28 was known prior to the incorporation of any interprocedural information due to the edge connecting it with the global node, G_w .

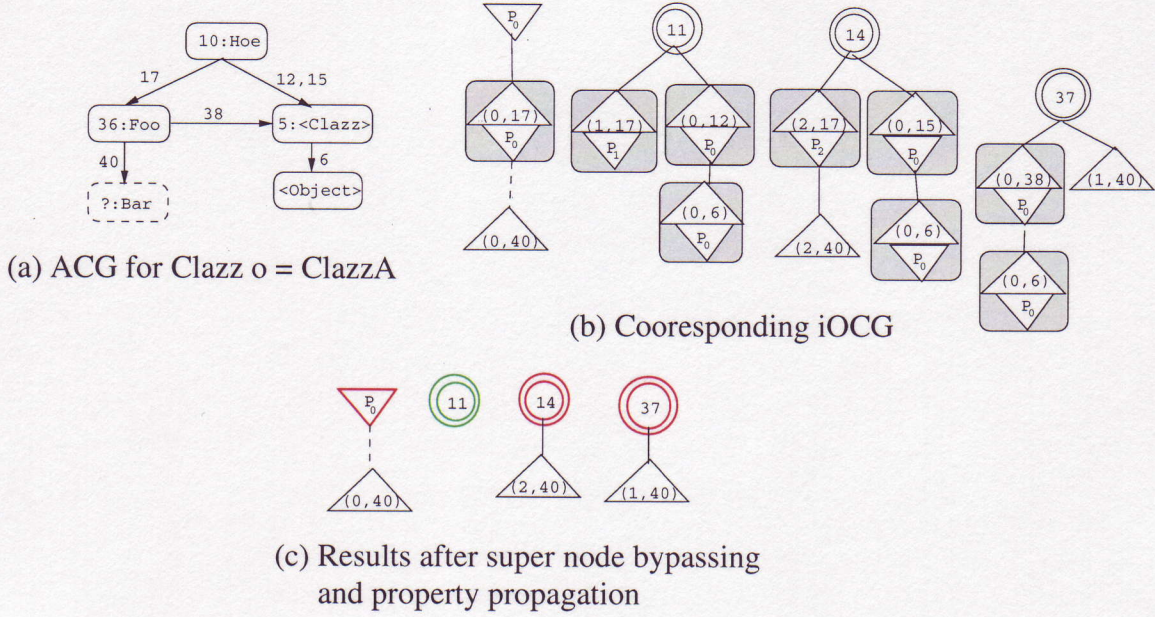


Figure 4.7 OCG propagation through the ACG for the type resolutionClazzA.

The next case to consider is when a weak edge is used to connect the parameter node, P_0 , for a virtual callee method. This case occurs when the runtime type of the object instance used to resolve Hoe is the typeClazzA. We show the ACG along with the iOCG before and after super node removal in Figure 4.7. The construction process is similar to that of the previous case. The key difference is that in Figure 4.7(b), a weak edge connects the formal node P_0 with the node $(0, 40)$. Therefore, the node for Bar in the ACG of Figure 4.7(a) remains speculative. The inclusion of a speculative node in the final ACG means that when the escaping state of the object instances is determined from the iOCG in Figure 4.7(c), only the escaping state of the object instance allocated at line 11 is definitively known. The other three object instances used in Hoe – P_0 and the allocations at lines 14 and 37 – are not decidable based on the iOCG information and

are therefore method escaping because they are passed into an unknown path the call to Bar.

Weak edges are not the only case where during ACG/iOCG construction the type of an object instance may no longer be considered definitively known. It is possible for more than one execution path to reach a given call. Therefore, it is possible for more than one type to be associated with a node. If that node is the parameter node, P_0 for a virtual call, then the call node cannot be promoted from speculative to known in the ACG. There are three types of nodes in an iOCG – formal, parameter, and allocation nodes – and two types of edges – strong and weak. Based on these node and edge types, we define the cases where a node type is no longer definitively known within the iOCG as follows.

Definition 5 *A node type is no longer considered definitively known within an iOCG if one of the following conditions is true:*

- *The node is directly connected to another node in the iOCG via a weak edge.*
- *The node is directly connected to another node that is already of unknown type.*
- *The node is directly connected via two or more strong edges to nodes of type formal and/or allocation.*

4.5 Runtime Structures

In order to improve runtime efficiency, a compact internal representation is used for the OCG and iOCG. Rather than representing them as graphs with pointers, a table is used, with each node represented as an entry in the table.

There are several steps when converting an OCG into the table representation. First, the global nodes only convey a change in escaping state of all nodes connected to them. Therefore, they can be removed from the graph and the attached nodes marked as method escaping. The removal of global nodes then leaves only three types of nodes remaining in the graph: parameter, formal, and allocation nodes. Second, in order to facilitate swift propagation, a transitive closure of the edges is formed. However, the graphs contain two types of edges, strong and weak. The rule for forming the transitive closure when it comes to the two types of edges is simply this: If a strong edge exists along any branch of a path traversed when forming the transitive closer, then the edge type of the edge added is a strong edge. This way, weak edges originally incident on parameter nodes remain incident on those nodes, and nodes that did not have at least one weak edge incident upon them prior to forming the closure do not acquire a weak edge. Since the primary purpose of the weak edges is to convey during interprocedural analysis that the exact type of an object instance is not definitively known, then the original edge still retains this information and allows for correct propagation.

Figure 4.8 shows the mapping for the nodes in the OCG into the table entries and bit vector format. Paths between the nodes are represented using bit vectors (*links*). Each

bit location represents an entry in the table, with the leftmost bit corresponding to the top entry. A bit is set for each node reachable from the current entry (through one or more edges). There are three additional bits for each entry in the bit vector field. The first of these, W (weak), is set only for parameter nodes and indicates that the node has a weak edge incident upon it. The second bit, C (changed), is set during interprocedural propagation to indicate that one or more of the bits within the bit vector has been set and requires propagation. The final bit, E (escaped), holds the escaping state of the entry.

For example, for the transitive closure formed from the OCG for the method `Bar`, shown in Figure 4.8(a), there are three formals coming into this procedure: P_0 , P_1 , and P_2 . These are the first three entries in the table. The fourth entry is the allocation node labeled 28. A parameter node labeled (0,29) is the fifth entry in the table. The edges between the nodes are defined by setting the corresponding bits in the links bit vector. For example, the edge between nodes P_1 and 28 is shown by the setting of the second and fourth bits (counting from the left) for both entries. The edge between nodes 20 and (0,29) is shown as the setting of the fourth and fifth bits for those two entries. The edges added by the transitive closure are represented by also setting the fifth bit for entry 2 and the second bit for entries 4 and 5. The remaining tables for the class `Clazz` are created in a similar fashion and are also shown in Figure 4.8.

Two properties of the OCG table representation enable swift interprocedural propagation of the escaping information. First, all interprocedural connections between the graphs for each method occur at the parameter entries, and second, since all paths from

each node are fully represented, the formal parameter entries for the callee method contain all the information that needs to be propagated to the caller. This information consists of either state bits, (W,C,E), or new connections between the actual parameters of the caller. The interprocedural propagation is performed in two passes: a forward pass to resolve speculative nodes in the ACG, and a backward pass to propagate state changes from the forward pass. Weak edges are propagated during the forward pass, caller-callee, allowing an edge to be downgraded from strong to weak. The rule for edge downgrading follows.

Definition 6 *If a node of type parameter in the iOCG has a weak edge directly incident upon it, then when the OCG for the callee is added to the iOCG, all edges directly incident upon any and all formal nodes within the super node are downgraded to weak. If a node of type F_{-1} (formal node of type return) has a weak edge directly incident upon it, then all edges connected to any and all parameter nodes within the super node are downgraded to weak.*

Since weak edges used to identify P_0 nodes where the type is unknown are critical for virtual ACG nodes and all ACG node resolution occurs on the forward pass, downgrading of edges is only implemented during the forward pass. The downgrading of edges only during the forward pass allows for the propagation of known type information and enables safe, accurate method resolution based on known types. Formals representing returns from the procedure are marked as escaping only when the OCG table is the root

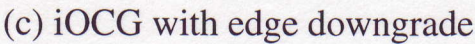


table for an ACG, thus capturing return escaping events but not introducing additional conservation by also marking callee return events.

93

edge to a weak edge because $P1_0$ has a weak edge incident upon it. However, from examination of the original OCG, it can be seen that $P2_0$ is actually from Obj_2 . Since allocations are always known types, the parameter $P2_0$ should have been a known type and the resolution of the virtual method called with it, likewise known. However, because of the downgrading of the new edge added from the transitive closure, the parameter $P2_0$ is no longer considered definitively known.

The propagation of information across the parameter connections is accomplished with a parameter map, which maps the actual parameters of the callee to the formal parameters of the caller. Its purpose is to perform the necessary masking and shifting of the links vector of the callee to the appropriate location of the corresponding actual parameter node in the caller. These vectors can then be propagated from the callee to the caller using a simple OR instruction. Any changes are then propagated within the caller's OCG, also using simple OR instructions.

Figure 4.10 shows the propagation across the call to *Foo* and its subsequent call to *Bar*. After all interprocedural interfaces occurring in *Bar* have been resolved: the *E* bit on the second formal parameter of *Bar* is propagated to the table for *Foo*. This change sets the *E* and *C* bits for the second to last entry in the table for *Foo*. The *C* bit indicates state changes that require propagation. This change propagates to all of the nodes connected to the changed node in *Foo* through the use of the link bit vector, causing the *E* and *C* bits to be set on the second entry (the second formal parameter into *Foo*). This information then propagates through the parameter mapping for the

Line Num.	Type	Param. Num.	W	C	E	Bit vector Links
0	0	0	0	0	0	1 0 0 0 0 1 0 0
11	1	0	0	1	1	0 1 1 0 0 0 1 0
12	2	0	0	1	1	0 1 1 0 0 0 1 0
14	1	0	0	0	0	0 0 0 1 1 0 0 1
15	2	0	0	0	0	0 0 0 1 1 0 0 1
17	2	0	0	0	0	1 0 0 0 0 1 0 0
17	2	1	0	1	1	0 1 1 0 0 0 1 0
17	2	2	0	0	0	0 0 0 1 1 0 0 1

Hoe

Line Num.	Type	Param. Num.	W	C	E	Bit vector Links
0	0	0	0	0	0	1 0 0 0 0 1 0 0
0	0	1	0	1	1	0 1 1 0 0 0 1 0
0	0	2	0	0	0	0 0 0 1 1 0 0 1
21	1	0	0	0	0	0 0 0 1 1 0 0 0
22	2	0	0	0	0	0 0 0 0 1 1 0 0 0
24	2	0	0	0	0	1 0 0 0 0 1 0 0
24	2	1	0	1	1	0 1 1 0 0 0 1 0
24	2	2	0	0	0	0 0 1 1 0 0 0 1

Foo

Line Num.	Type	Param. Num.	W	C	E	Bit vector Links
0	0	0	0	0	0	1 0 0 0 0 0
0	0	1	0	0	1	0 1 0 1 1 1
0	0	2	0	0	0	0 0 0 1 0 0
28	1	0	0	0	1	0 1 0 1 1
29	2	0	0	0	1	0 1 0 1 1

Bar

Figure 4.10 State propagation through the Parameter Maps (circles indicate changes during iOCG formation).

interprocedural interface between Hoe and Foo, eventually setting the E and C bits on the second and third entries in Hoe’s table.

After the E bit has been fully propagated, a quick look at the two allocations in Hoe’s table show that the object instance at line 11 must be heap allocated since it is escaping, but the object instance at line 14 can be stack allocated.

4.6 Experimental Setup and Results

To evaluate the effectiveness of our techniques, we used several benchmarks from the Java Grande Threaded Benchmark Suite [35] and several small threaded Java applications from Doug Lea’s book [36]. A description of the programs used can be found in Table 4.1. The first seven of these are from [35] and the last seven are from [36].

Table 4.1 Description of benchmarks and applications used.

Program	Threads	Description
JGFCrypt	4	Performs the International Data Encryption Algorithm (IDEA) on an array of N bytes.
JGFLUFact	4	Solves an NxN linear system using LU factorization. Same as Linpack.
JGFSor	4	Perform 100 iterations of successive over-relaxation on a NxN grid.
JGFSeries	4	Computes the first N Fourier coefficients of the function: $f(x) = (x + 1)^x$.
JGFSparseMatmult	4	Performs a sparse matrix multiplication algorithm using a compressed row format.
JGFMolDyn	4	Models partial interactions with boundary conditions.
JGFRayTracer	4	Renders a 3D scene containing 64 spheres.
Heat	4	Simulates heat diffusion across a mesh.
Fib	4	Computes a Fibonacci number using a specified number of threads.
Msort	4	Parallel merge/quick sort on random numbers.
NQueens	4	Positions N queens on an NxN board so that they can not attack each other.
BarrierJacobi	4	Performs a cyclic barrier version of Jacobi iteration on a mesh of the given size.
LU	4	Decomposes a randomly filled matrix.
MatrixMult	4	Matrix multiplication using parallel divide-and-conquer.

Evaluating the effectiveness of an optimization on threaded applications poses several problems. The timing and performance of a threaded application depends on the timing of the threads. Slight changes in timing between runs of the same application can cause significant changes in performance by causing one or more threads to stall waiting on resources that during previous runs had been available at the optimal time. Additionally, virtual machine overheads, such as garbage collection, can affect performance of the

application by occurring at different times. Complicating this data collection dilemma further is the desire to evaluate the effectiveness of our techniques in a state-of-the-art Java runtime rather than a VM used solely for research.

To address these issues, we chose to collect traces of the benchmarks using the production level HP Hotspot 1.0 VM [37] in interpreted mode. The use of the tracing mechanism allowed us to capture the dynamic execution trace for these programs on a production level runtime. We then use this trace to guarantee that the execution order remains the same for every run of the applications and evaluate the costs of our analysis based on the same execution order. Furthermore, we guarantee that garbage collection occurs at the same point in the execution and evaluate the effects on memory of our system.

To handle the simulation of a virtual machine executing the trace lines, we built a simulation of a Java runtime environment. Figure 4.11 shows an overview of the Simulated Runtime. The key components of our Simulated Runtime are the Simulated Execution Engine, the Thread States, the Memory Manager, and the Method Invoker, with its associated class loader. A more thorough description of the simulator can be found in Appendix B.

To explain the data collection process we describe briefly the basic units of the simulated runtime that are pertinent to the collection of the data. The simulated execution environment simulates the effects of each bytecode instruction encountered by reading the trace file line in execution order. It then applies the appropriate state changes to the specified thread, loading and storing thread states as necessary.

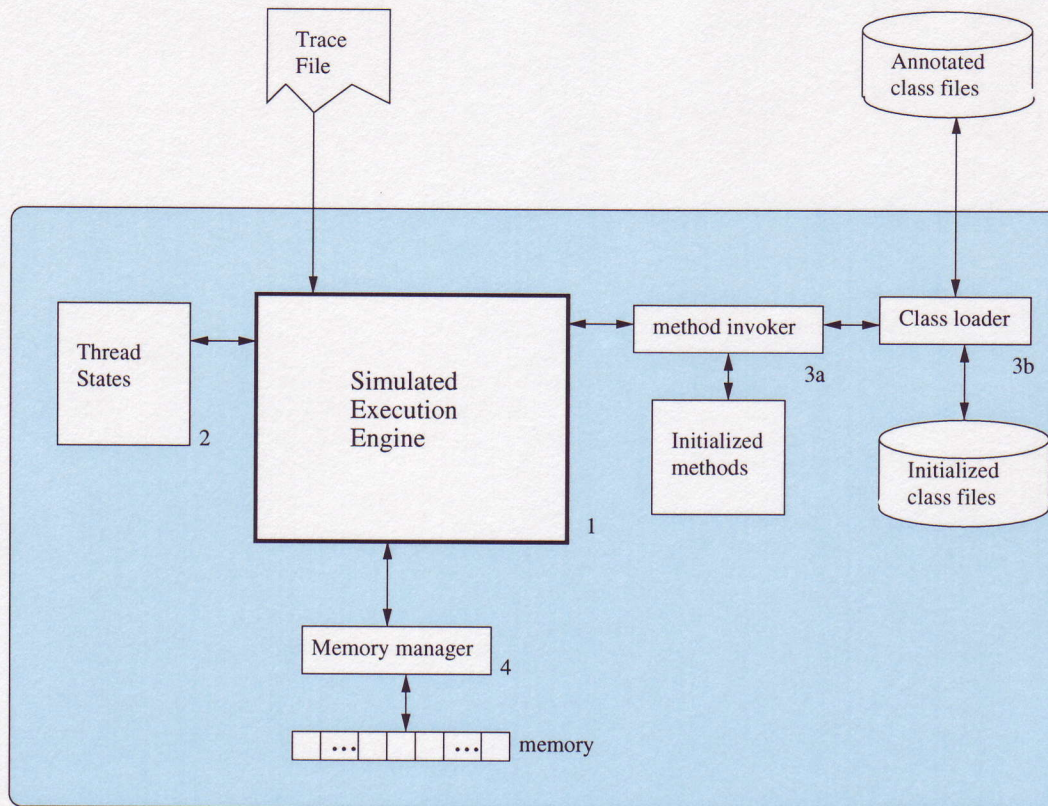


Figure 4.11 An abstract overview of the simulated runtime environment.

Invocations are handled as requests to the method invoker. When a request is received by the method invoker, it first checks to see if it has an initialized version of the method available. If it does not have one, it places a request to the class loader. The class loader first checks its initialized class files to see if the requested class is available. If not, the class loader loads the class file from a set of preannotated class files. During simulation, these class files always exist. The class loader locates the requested method and returns it to the method invoker. When the method invoker receives a new method from the class loader, it computes the OCG for the method based on its CDG and stores the initialized

method (the bytecode for the method, the clean CDG, and the clean OCG) in its table. A copy is made of the initialized method and passed back to the simulated execution environment.

The OCG of a newly invoked method returned from the method invoker is checked by the simulated execution environment to see if it contains allocation nodes. A Boolean field is used to indicate whether or not a given OCG contains an allocation node. This field is set at the time the OCG is constructed. If it does contain allocation nodes, based on the runtime object type used to locate the method and any known object types in the thread's state for this invocation, the simulated execution engine constructs the ACG/iOCG with this method as the root node. If the OCG does not contain allocation nodes, nothing is done, and the only overhead is the check.

If the simulated execution environment encounters a bytecode instruction involving a memory location, it processes the request by also involving the memory manager. The memory manager handles allocation requests as well as field assignments. The simulated memory representation contains, for each simulated memory location, a table representing referenced fields as well as the state of the memory location, indicating whether or not a given memory location is method local. Each field is identified with its unique field ID and contains a reference to the simulated memory location it is referencing.

4.6.1 Experimental results: potential benefits

In addition to the introduced conservativeness already discussed, there is one more level of conservation occurring in our implementation from our self-imposed bounding

of the ACG/iOCG resolution. We stop the method resolution process in the forward pass when we reach a speculative node that cannot be promoted or when we reach a predetermined maximum resolution depth. We set the maximum resolution depth to six based on observed call stack depths within the simulated runtime. While this constraint allows us to bound the size of the ACG and guarantee termination of the ACG/iOCG construction process, we do not always resolve the graph as precisely as we could have. Any node in the ACG exceeding the maximum resolution depth remains speculative and therefore by Definition 4, unknown path escaping.

To evaluate the impact of level of added conservation as well as determine the effectiveness of the ACG/iOCG analysis technique, we also implemented an oracle version of the algorithm. The oracle version used the full CDG and the actual executed calls within the trace to determine *method escaping*. The full CDG still contains the direction on the arcs and does not contain a transitive closure. Furthermore, we only updated information within the iCDG as a given bytecode instruction is executed. This allowed us to eliminate any conservation introduced by the absence of control flow information since we followed the exact execution flow of the application. In addition, we computed the results at the point of execution, or after the object instance’s full lifetime had been explored, therefore making all nodes in the ACG *known* and the full call depth explored. This eliminated the final loss of precision caused by the speculative nodes and the maximum resolution depth in the ACG/iOCG analysis. We consider the use of CDGs in forming this version of the iCDG as an oracle result that is not achievable with either

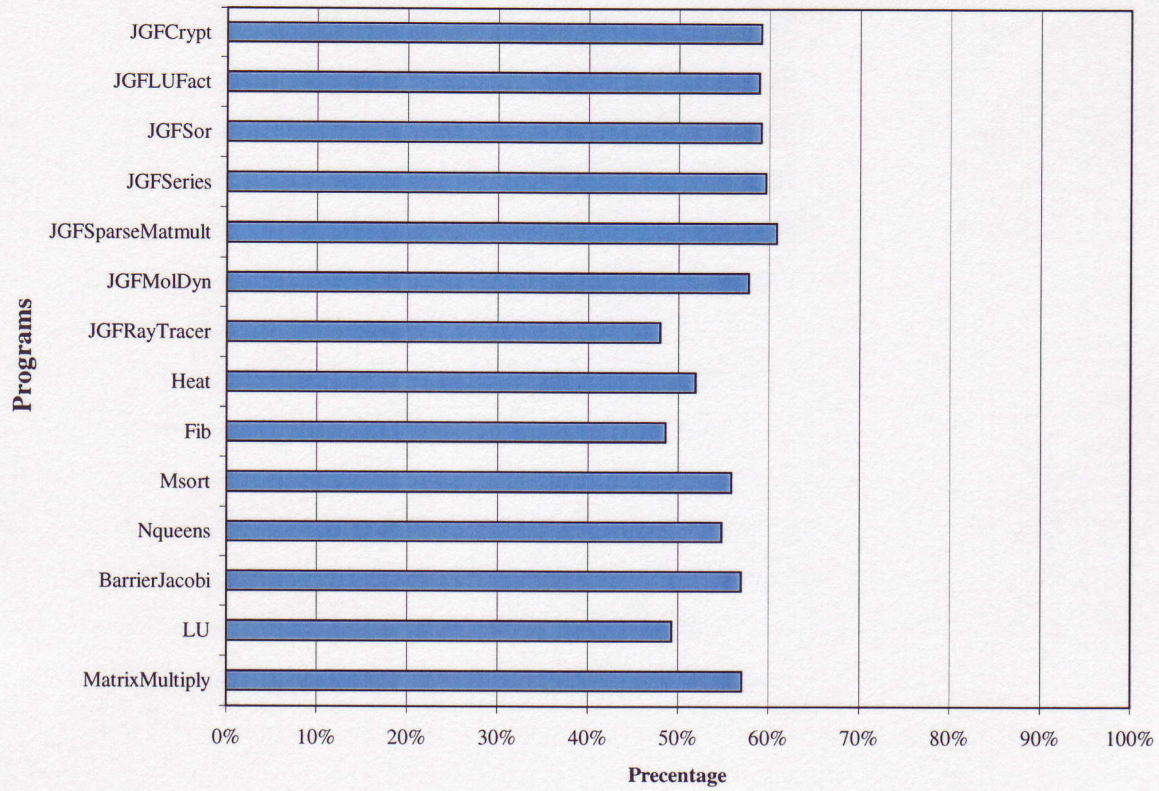


Figure 4.12 Percentage of *method local* using iOCG compared to oracle method.

runtime or static analysis. However, it does provide an upper bound for the evaluation of our simplification.

In Figure 4.12 we compare the determination of *method local* allocations using our iOCG method to the best possible results from the oracle method. The graph shows the percentage of *method local* allocations found using the iOCG compared with those found with oracle's perfect knowledge. To compute the information, we tracked the decision as fields in the simulated memory locations used by the memory manager, and the values were collected by analyzing the memory locations collected at the first GC event. Note

that, on average over the benchmarks, the quicker iOCG method is able to identify 53.9% of the *method local* allocations identified by the oracle method. This result indicates that although we introduced additional conservatism, we were able to identify a significant portion of the actual method local memory locations.

Additionally, we tracked what percentage of the collected memory locations were identified as method local based on each of the analysis methods. Figure 4.13 shows for each of the analysis methods, iOCG and oracle, what percentage of the total collected memory can be stack allocated. An average of 68.8% of the memory collected was identified as stack allocatable by the oracle analysis, compared to 36.5% by the ACG/iOCG technique. These results imply that the use of our technique within our framework could reduce garbage collection events by as much as 30% by stack allocating these identified object instances.

To understand how far the benchmark had progressed at the first GC event, we also recorded the number of dynamic bytecode instructions execution by each of the benchmarks at the time of the GC event. Table 4.2 contains these numbers. Note that `Heat`, `Fib`, `Msort`, and `NQueens` completed execution entirely, and they also showed a large percentage of the memory locations as method local.

4.6.2 Experimental results: estimated costs

To estimate the costs of our implementation, we compare the cost of the analysis to the execution time of the application. The propagation of information through the ACG/iOCG uses logical OR operations. We recorded the number of logical OR operations

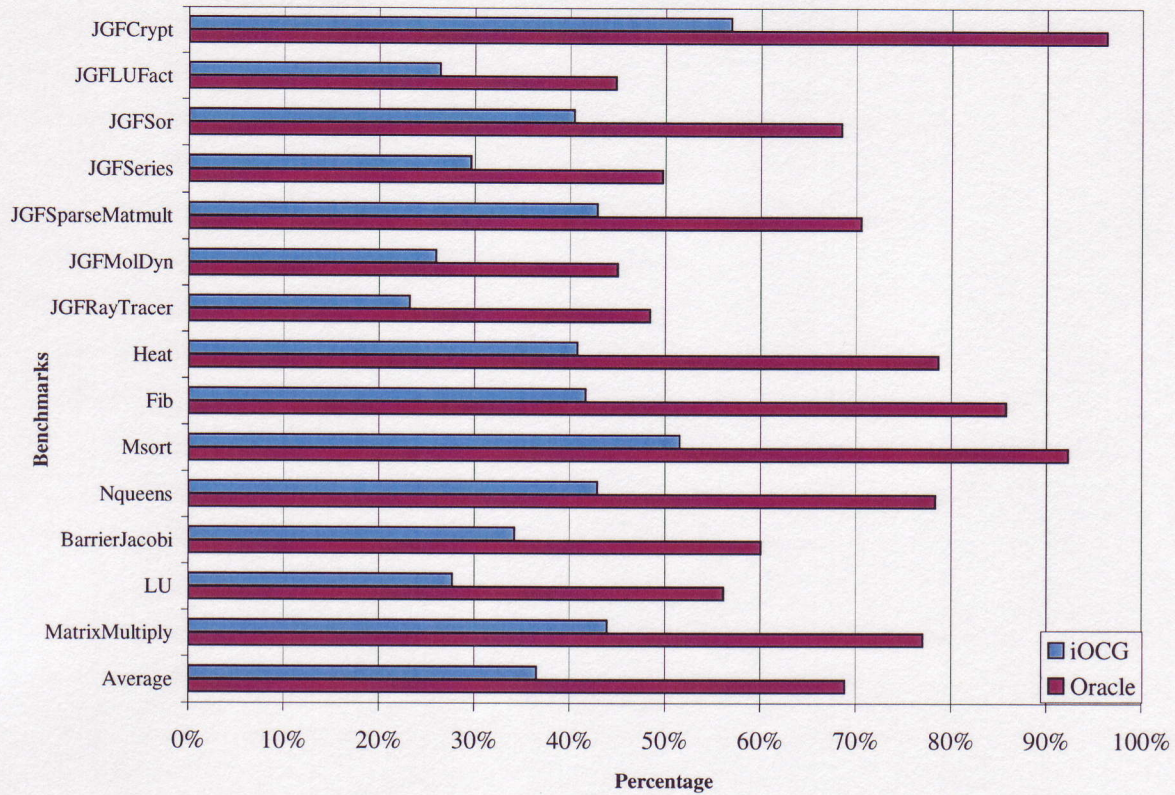


Figure 4.13 Percentage of method local memory location collected using iOCG compared to oracle method.

performed in constructing the ACG/iOCG for each of the benchmarks, prior to the GC event. We assert that the cost of each logical OR operation with its associated loads is comparable to the execution of a single bytecode instruction and therefore consider this a fundamental measure. Note that in support of this assertion, most bytecode instructions translate into more than one simple assembly instruction, some requiring additional safety checks by the virtual machine. In Figure 4.14 we show the ratio of the fundamental OR operations to the dynamically executed bytecode instructions for each benchmarks. This cost averages around 20%, which is within the analysis cost bounds we had set.

Table 4.2 The actual number of dynamic bytecode instructions executed at the time of GC.

Program	Dynamic instructions
JGFCrypt	99,910,804
JGFLUFact	79,014,889
JGFSor	83,261,676
JGFSeries	67,872,390
JGFSparseMatmult	115,188,730
JGFMolDyn	92,423,979
JGFRayTracer	25,505,124
Heat	18,541,123
Fib	4,289,037
Msort	7,206,937
NQueens	6,189,075
BarrierJacobi	99,877,803
LU	64,816,366
MatrixMult	98,195,473

To understand the values in Figure 4.14, we also tracked the ratio of method OCGs that contained an allocation event versus the total number of methods used by the benchmarks. This method breakdown is shown in Figure 4.15. Because of the large number of methods coming from the runtime library files, we have broken the methods with allocation percentage into library methods and application/benchmark specific methods. Note that for benchmarks that allocate a large portion of temporary location but whose percentage of benchmark specific methods to library methods is small (Figure 2.8), the percentage of allocations is substantial. For example, in `JGFSparseMatmult`, which transforms sparse matrices into vector representations, less than 3% of the methods were from benchmarks but more than 57% of those contained allocations. This relationship also holds for the similar benchmark, `JGFSor`. For the `JGFRayTracer`, the benchmark meth-

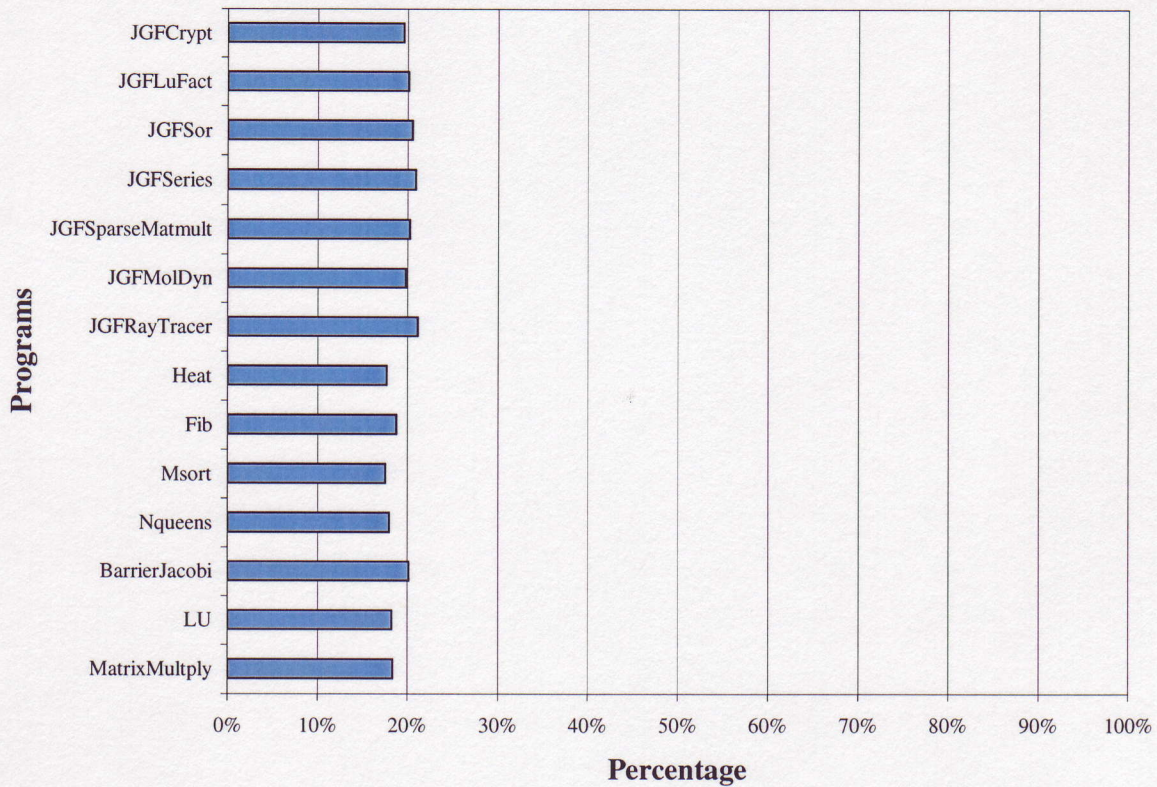


Figure 4.14 Percentage of OR operations to dynamic bytecode instructions.

ods also contain a large number of allocations, but not as many are *method local*: because it is computing the pixels for a scene and passing some of the results back, the objects tend to persist beyond their allocating methods. JGFCrypt, on the other hand, does not contain a particularly large percentage of methods containing allocations compared to the other benchmarks. However, these methods use a substantial number of temporary structures to hold intermediate results, which can be stack allocated.

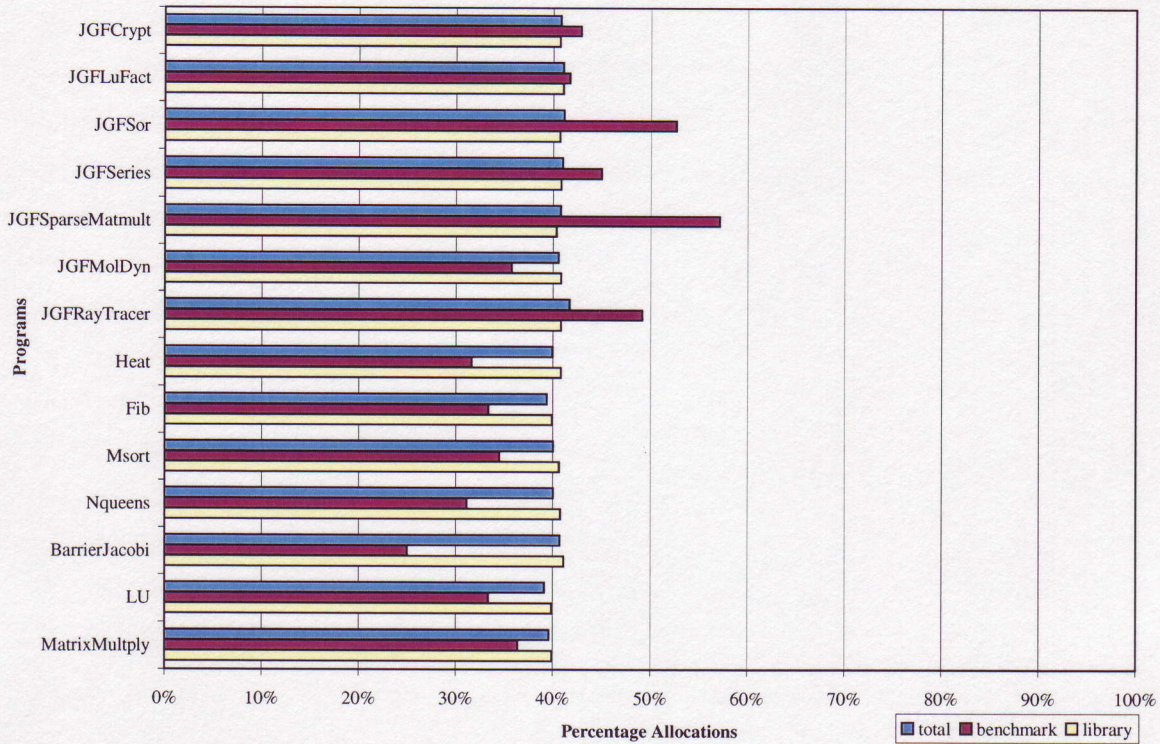


Figure 4.15 Percentage of unique methods containing allocations.

4.7 Conclusion on iOCG

In this chapter we have shown how our framework is able to capture a significant percentage of *method local* stack allocatable object instances that traditionally remain unidentified by most runtimes. Since most runtimes do not attempt to perform this level of analysis until the first optimization epoch, these initialization object instances are rarely if ever identified as stack allocatable and optimization opportunities are missed. If identified early with a swift, safe, analysis method, their promotion to stack allocatable objects could delay a GC event and potentially improve startup performance.

CHAPTER 5

DYNAMIC OPTIMIZATION VALIDATION AND ROLLBACK

Up until this point we have discussed how the Dynamic Application Analysis Framework can be used to generate analysis results in a dynamic application. However, the structures and design of this framework are also applicable to the validation of optimizations enabled by the analysis results. In this chapter we classify models for optimization and describe the types of validation that would be necessary to enable each. We also discuss the necessary mechanisms for rollback and recovery in the event of a validation failure. In several cases, we show how the information contained within our framework structures not only facilitates more aggressive optimizations but also enables a finer grain of validation. By enabling this finer grain validation, some forms of optimizations have the potential for longer dynamic lifespans.

5.1 Optimization Models and Validation

We start by first defining three types of optimization decisions based on how the optimizer decides to use the information provided by the analysis engine. They are as follows.

Definition 7 *Optimization decisions*

- ***always safe***: The optimization performed by the optimizer is restricted to a provable set of monomorphic classes and procedures. Execution of the optimizations enabled under this model remain safe even in the presence of dynamic class loading.
- ***sometimes safe***: The optimizer assumes the class hierarchy (CH) is open. The optimizer recognizes the “speculative” nodes in the ACG and optimizes based on these points of optimization-time target nondeterminism. Optimizations enabled under this model embed checks within the code to determine dynamically when it is safe to execute along the optimized code path.
- ***speculatively safe***: The optimizer assumes the class hierarchy (CH) is closed. The optimizer aggressively promotes speculative nodes within the ACG to known nodes based on information within the CH. It relies on validation and recovery mechanisms within the runtime if the closed CH assumption is violated. It provides the runtime with the necessary information to provide the desired level of validation and recovery.

The actual optimization of a given code region may use a combination of the three types noting that the aggressiveness of the optimization is correlated to the optimization type. Therefore, optimizations based on the always safe model are not as aggressive as those based on the speculatively safe model. However, the always safe model does not require validation since it is always safe to execute the code optimized under this

model even in the event of dynamic class loading. The sometimes safe and speculatively safe models do require validation, and we classify validation into two main categories as follows.

- *execution time*: The validation check is embedded in the executing code and performed as part of the execution.
- *load time*: The validation check is performed by the runtime as a new class file is loaded.

The two types of validation are related directly to the two optimization types requiring validation. The *sometime safe* model of optimization, which embeds validation checks in the optimized code, is an example of *execution time* validation, while the *speculatively safe* model, which relies on the runtime to handle validation, is an example of *load time* validation.

5.1.1 Always safe

As previously mentioned, the optimizations we focus on are based on interprocedural analysis and use optimization tools such as inlining. The determination of whether an optimization is *always safe* is based on the type of the procedure being considered for inlining and optimization. We restrict *always safe* optimizations to monomorphic procedures, which we introduced in Chapter 1. We define them more formally as follows:

Definition 8 *A monomorphic procedure is any procedure that, within a given context, can be proven **not** to be redefined in the presence of dynamic class loading during the ex-*

ecution of a dynamic application and is a single-target node in the ACG for the procedure being analyzed and optimized.

Therefore, in addition to procedures that were declared using the `final` key word or procedures that are *sealed* within a *sealed* package [22], other procedures can also qualify as monomorphic. Initializers called by an allocating method are considered monomorphic procedures because the type of the object instance being initialized is defined by its allocating procedure and therefore definitively known. Furthermore, method resolution using the same definitively known allocated object instance, are also single target and monomorphic within the iCDG/ACG with respect to the given context. For an object instance to be considered *definitively known within a given context*, it must meet all of the following criteria:

1. The object instance is allocated within the iCDG.
2. The object instance is only attached via a node or nodes of type *write*, to another node in the iCDG that are also *definitively known*.
3. The object instance is **not** attached to any node of type *read* that contains the bytecode line number of a `checkcast` bytecode instruction.
4. The object instance remains enclosed within the iCDG under analysis.

Qualification 1 is a requirement because only object instances actually created within the iCDG can be considered members of the given class and not one of its subclasses.

Qualification 2 addresses type changes. If an object instance is written to by another

object instance represented in the CDG that is also definitively known, then even though the type may change, it still remains definitively known. Therefore, qualification 2 is recursive in that in determining if a node is *definitively known* the optimizer may also need to resolve attached nodes in the iCDG. Qualification 3 uses the information that a `checkcast` bytecode instruction is actually represented in a CDG as a node of type *field read*, as specified in Appendix A. Since the node in the CDG contains the bytecode line number of the operation, a simple check for the bytecode type at that line number can be used to qualify this. By performing the secondary bytecode check, we can avoid eliminating all object instances connected to read nodes from the set of definitively known object instances. Qualification 4 simply addresses the fact that we can only know the definitive state of an object instance if we can analyze the whole region where the object instance is used. Therefore, any object instance escaping either via an interprocedural boundary into a speculative node within the ACG or to another thread, cannot be fully analyzed and therefore its state cannot be definitively known.

Figure 5.1 shows the optimizing time ACG which assumes the CH contains the three versions of `Clazz` shown in Figures 2.3 and 3.1. This ACG varies from the others in that we have completed two potential targets for `Foo` and two potential targets for `Bar` while still leaving a purely speculative node for each. The calls from `Hoe` to `Foo` and subsequently `Bar` are virtual calls located via a formal to `Hoe` which is not definitively known. The only monomorphic procedures shown in this ACG are the initializers. We have distinguished these from the known nodes by using a double line around the node.

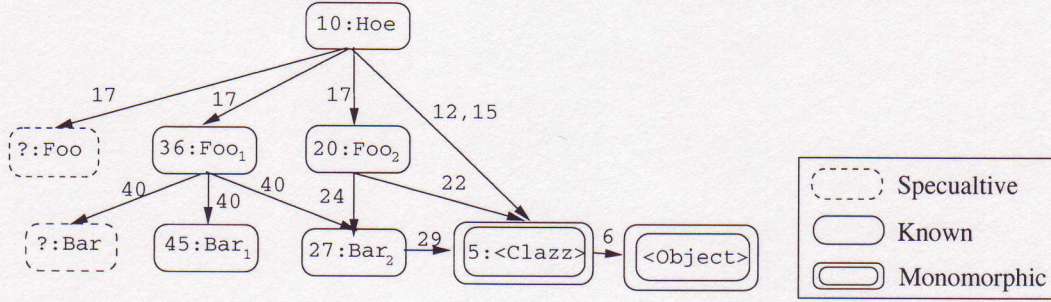
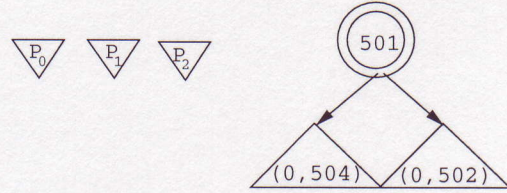


Figure 5.1 The optimizing time ACG for the versions of `Clazz` in Figures 2.3 and 3.1.

```

500: public void HoeUser(Clazz A, Clazz B){
501:     Clazz C = new Clazz();
502:     <Clazz>();
503:
504:     C.Hoe();
505: }

```



(a) Procedure using `Hoe`

(b) CDG for `HoeUser`

Figure 5.2 Example user for the class `Clazz` from Figure 2.3.

If we are optimizing `Hoe` based on the *always safe* premise, then only the initializer calls at lines 12 and 15 are safe for inlining and optimizations.

In contrast, consider the case where the type for the object instance used to call `Hoe` is definitively known by the optimizer. Figure 5.2(a) shows the procedure `HoeUser` that calls the procedure `Hoe`. If we are optimizing `HoeUser`, then the determination of inlining candidates is based on `HoeUser`'s iCDG/ACG.

We step through the construction of the iCDG/ACG for `HoeUser` and identification of ACG nodes as monomorphic, in Figure 5.3. First, starting with the CDG for `HoeUser`,

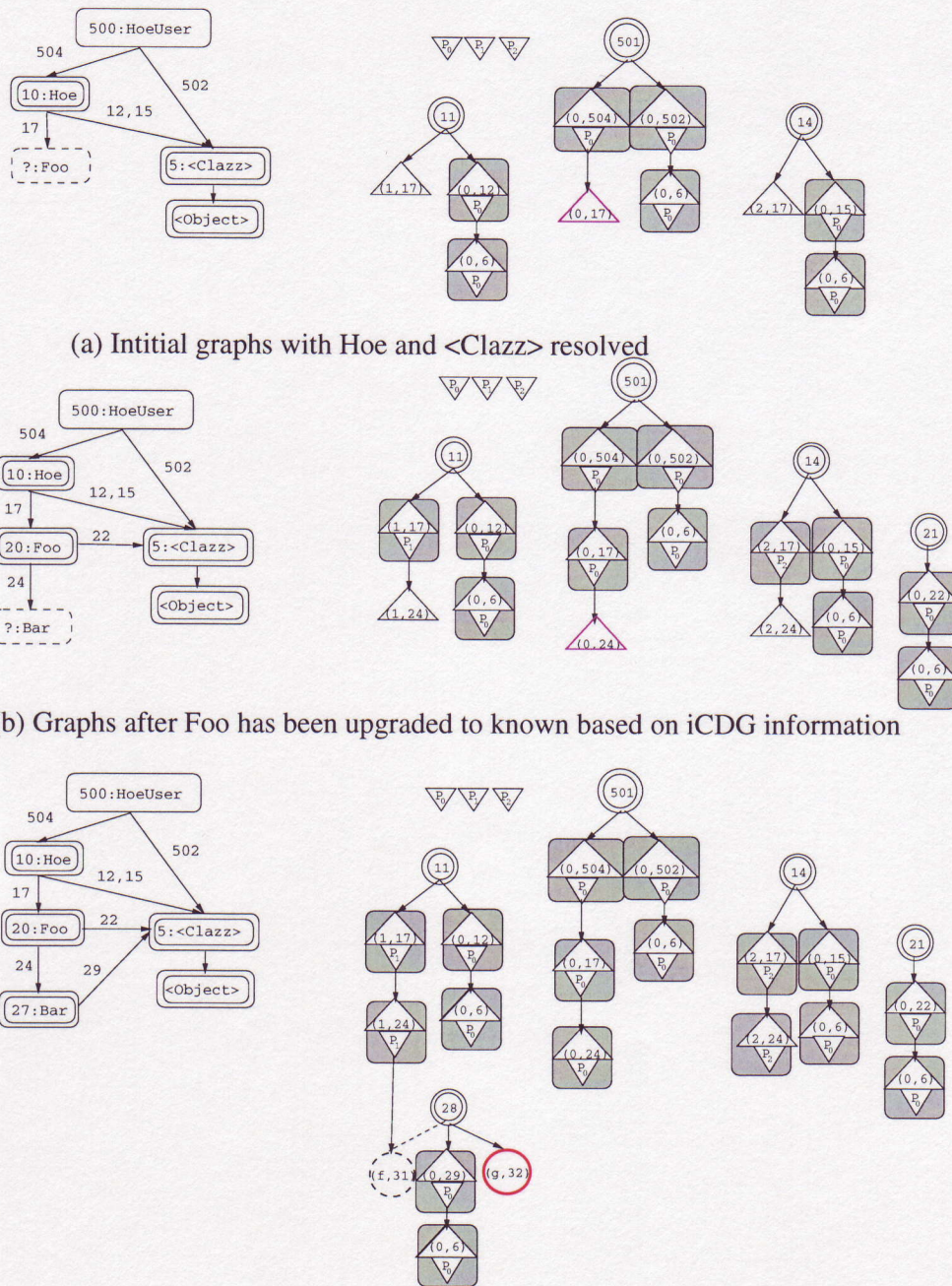


Figure 5.3 Interprocedural CDG for HoeUser in Figure 5.2.

Figure 5.2(b), we connect the initializer called at line 502 and resolve the virtual call to `Hoe` at line 504. Note that the call to `Hoe` and its subsequent node in the ACG is upgraded to monomorphic because the object instance used to resolve it is the same one allocated at line 501. The next step is to connect the CDG for `Hoe` into the iCDG which causes the speculative node `Foo` to be added to the ACG. These iCDG/ACG are the graphs shown in Figure 5.3(a). Upon analysis of the iCDG in Figure 5.3(a) it is concluded that the same object instance that was used to resolve the virtual call to `Hoe` is also used to resolve the virtual call to `Foo`. This determination is made by following the arc connected to the parameter node, (0,17), and treating super nodes as extensions of their attached arcs. The only node connected to this parameter node (excluding super nodes) is the allocation node, 501. This then causes the node `Foo` in the ACG to be upgraded to monomorphic and its corresponding CDG incorporated into the iCDG. This is shown in Figure 5.3(b). The incorporation of `Foo` causes a new node to be added to the ACG, `Bar`, which is added as a speculative node. By analyzing the iCDG in Figure 5.3(b) it is observed that the same object instance used to resolve `Hoe` and subsequently `Foo` is also used to resolve `Bar`. Therefore, `Bar` can be upgraded to monomorphic and its corresponding CDG incorporated into the iCDG. This is shown in Figure 5.3(c). The incorporation of `Bar` did not add any new nodes to the graph and the CDGs for all of the known nodes have been incorporated into the iCDG; therefore, the iCDG/ACG construction terminates.

The final ACG shown in Figure 5.3 contains all monomorphic nodes. This means that all procedures executed during the execution of `HoeUser` are candidates for inlining and

subsequent optimizations. The final optimized code would remain valid at all times since dynamic class loading cannot affect the monomorphic state of the nodes in this ACG.

In general, only a subset of the object instances in an iCDG can be classified as *definitively known* and as such only a subset of the nodes in the ACG can be promoted to monomorphic nodes and optimized using the *always safe* model. This makes the always safe model conservative, leaving many optimization opportunities unexploited.

5.1.2 Sometimes safe

The *sometimes safe* optimization model recognizes that the CH is not closed and can increase the number of class files in it during execution. This causes the inclusion of speculative nodes for any nodes that are not monomorphic within the ACG. This model is more aggressive than the always safe model, choosing to optimize one or more paths through the known nodes in the ACG. The optimizer also embeds the necessary validation checks and recovery mechanisms in the optimized code to handle the multiple potential execution paths.

For example, in Figure 5.1, if the profile showed that the execution had always resolved the call to `Foo` as `Foo2`, the optimizer under the *sometimes safe* optimization model would inline `Foo2`, but would embed a type check as validation as early as possible in the code to determine the type of the object used to locate `Foo`. If the type check failed, it would redirect execution to the original version of the code. The validation check under this model would be an *execution time check* since the validation is embedded in the optimized code. Note that for this example, there are actually two types that can result in the execution

of `Foo2`, `Clazz` and `ClazzB` as shown in the ACGs of Figures 3.2. Therefore, since the optimizer assumed the ACG for the unoptimized path could expand to yet unknown classes, it would need to embed a double type check for this inlining to direct only types of `Clazz` and `ClazzB` to the optimized segment.

However, checking the type at every execution of a method adds overhead even to the optimized path. If the loading of a new class of the given type is a rare event, then this overhead may be mitigated by a more aggressive optimization model.

5.1.3 Speculatively safe

Under the *speculatively safe* optimization model, the CH is assumed closed, meaning this model assumes no new class files will be loaded into the system. The assumption forces the registration of a validation check with the runtime. The actual registration of the validation check can be delayed until after the optimization, allowing the optimizer to potentially request a finer grain validation of specific properties of the newly loaded class files.

For example, in Figure 5.1, only the two known nodes for `Foo` would be included in the ACG. Although a type check would still be needed to determine the correct version to execute, the set of types would be considered closed. Unlike the double check used in the sometimes safe model, this optimization model would only need to check the single type, `ClazzA`, that can resolve to `Foo1`. This model registers a validation request with the runtime relying on the runtime to disqualify the optimized code should the closed CH assumption become invalid.


```

100: class ListClazzA extends ListClazz{
101:     ClazzNodeA C;
102:
103:     ClazzNode getElement(ListClazz o, int i){
104:         if(i < 0 || i >= o.s || i == 0)
105:             return o.C;
106:         ClazzNodeA X = o.C;
107:         for(int j = 0; j < i; j++){
108:             X = X.getNext();
109:         }
110:         return X;
111:     }
112: }
113:
114: class ClazzNodeA extends ClazzNode{
115:     ClazzNodeA n;
116:
117:     ClazzNodeA getNext(ClazzNodeA o){
118:         return o.n;
119:     }
120: }

```

Figure 5.4 Subclass for the list class in Figure 3.9.

5.1.4 Mixing optimization models

Sometimes *speculative* optimizations can be mixed with *sometimes safe* or *always safe* optimizations. To illustrate the combining of sometimes safe and speculative models, Figure 5.4 introduces two new subclasses, `ListClazzA` and `ClazzNodeA`, for the classes introduced in Figure 3.9. The new ACG for the call to `append` is shown in Figure 5.5. Additionally, we introduce a new user class for these two versions of `ListClazz`, `listBuilderDriver` shown in Figure 5.6. If the optimizer is optimizing `listBuilderDriver`, and the profile shows that the call to `getElement` always resolved to the version at line 103 of Figure 5.4, then the optimizer under the *sometimes safe* model could decide to inline this version of `getElement`. However, since this is a virtual call, the ACG would also have a speculative node for `getElement`. The optimizer upon deciding to inline the

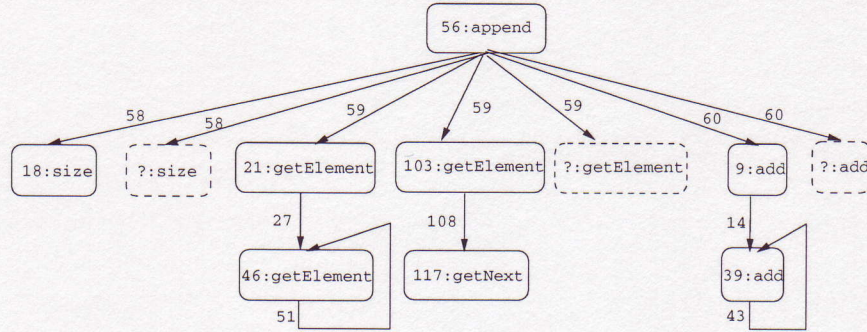


Figure 5.5 ACG for `append` in Figure 3.9 with the new subclasses in Figure 5.4.

```

100: void listBuilderDriver(listBuilder o, ListClazz[] lists){
101:     ListClazz L = new ListClazz();
102:     <ListClazz>(L);
103:
104:     ListClazzUser D = new ListClazzUser();
105:     <ListClazzUser>(D);
106:
107:     for(int i = 0; i < lists.length; i++){
108:         D.append(lists[i], L);
109:     }
110: }

```

Figure 5.6 Driver class for the subclasses of `ListClazz`.

version of `getElement` based on the profile information, would also place a type check for the `ListClazz` type when it is read from `A` at line 59 of Figure 3.9. The check would be used to redirect to a normal invocation if the type is not `ListClazzA`. This then classifies this portion of the optimization as *sometimes safe*.

Figure 5.7 highlights the sometimes safe portions of the optimized version of `listBuilderDriver`. Now the optimizer could make a *speculative* decision based on the loaded versions of `getElement` and `add`. It could decide this set is closed and to cache the value returned by the call to `size` in the `for` loop since neither of these causes the value returned


```

200: void listBuilderDriver(listBuilder o, ListClazz[] lists){
201:     //inline initializer for ListClazz L
202:     //inline initializer for ListClazzUser D
203:     //check escaping state of lists to determine which loop
204:     if(lists not escaping){
205:         int temp = lists.length;
206:         for(int i = 0; i < temp; i++){
207:             //inlined append from ListClazzUser
208:             //assumed that append formal 1 does not escape
209:             //in call to getElement();
210:             //cache size result
211:             int t_2 = lists[i].size();
212:             for(int j = 0; j < t_2; j++){
213:                 //optimized version of loop body
214:                 if(lists[i] is ListClazz){
215:                     //optimized version of getElement and add
216:                 }else{
217:                     //original code
218:                 }
219:             }
220:         }
221:     }
222:     else{
223:         //original code
224:     }
225: }

```

Sometime Safe checks

Figure 5.7 Sometimes safe regions of optimizations for listBuilderDriver.

by `size` to change across iterations of the loop. If a new subclass of `ListClazz` is loaded into the runtime, this optimization may no longer be valid. Therefore, the caching of the return value of `size` would be *speculative* and the optimizer would register a validation request with the runtime for the class loader.

Figure 5.8 highlights the speculative portions of the optimized version of `listBuilderDriver`. Note that both the speculative and sometimes safe optimizations are intertwined in this optimized code segment and the validation failure of the speculative optimization disqualifies both of them.


```

200: void listBuilderDriver(listBuilder o, ListClazz[] lists){
201:     //inline initializer for ListClazz L
202:     //inline initializer for ListClazzUser D
203:     //check escaping state of lists to determine which loop
204:     if(lists not escaping){
205:         int temp = lists.length;
206:         for(int i = 0; i < temp; i++){
207:             //inlined append from ListClazzUser
208:             //assumed that append formal 1 does not escape
209:             //in call to getElement();
210:             //cache size result
211:             int t_2 = lists[i].size(),
212:             for(int j = 0; j < t_2; j++){
213:                 //optimized version of loop body
214:                 if(lists[i] is ListClazz){
215:                     //optimized version of getElement and add
216:                 }else{
217:                     //original code
218:                 }
219:             }
220:         }
221:     }
222:     else{
223:         //original code
224:     }
225: }

```

Speculative optimization
 Requires runtime validation

Figure 5.8 Speculative optimization in listBuilderDriver.

5.2 Validation Failure, Rollback, and Recovery

Up until this point, we have discussed the need for validation and what types of validation are needed for each optimization model. However, validation could fail. In the event of a validation failure, rollback and recovery may be necessary. In this section, we address the types of rollback and recovery needed to handle validation failures, how these mechanisms are impacted by the state of the execution at the time of the validation failure, and what type of information is needed by the runtime to recover from validation failure events.

We start by identifying the following rollback mechanisms as facilitating the rollback of the state of the application in the event of a validation failure.

- *Code based stubs*: The replacement of entry instructions at the entry point of what was previously a valid optimization with an unconditional jump to rollback and recovery code.
- *On stack replacement*: The replacement on the call stack of return targets with unconditional jumps to rollback and recovery code.
- *Stack object instance flushing*: The transferring of a stack allocated object instance to the heap.

The first two mechanisms are general and may be necessary for handling validation failures across a wide range of optimizations. The third recovery mechanism is directly related to a particular type of optimization, namely stack allocation of object instances and therefore optimization type dependent. These rollback and recovery mechanisms are not exclusive to a particular type of validation and could require employment either with *execution time* or *load time* validation. For example, in an execution time validation check, the validation failure event normally redirects either directly to an unoptimized version of the code or first performs a rollback for some of the state prior to the redirect. The rollback may include the flushing of stack allocated object instances. In the event of a load time failure, not only could code stubs be used to redirect future execution of the optimized region, but it may also be necessary to flush stack allocated object instances and even in some cases to reset and re-execute portions of the affected region. In the remainder of this section we focus primarily on load time validation failure events since execution time validation failures are directly handled within the code.

Class file name
Type of validation
Coarse
Fine
Rollback/recovery needed
Call stack
Code
Memory

Figure 5.9 Abstract view of the necessary fields in a validation registration.

Load time validation is handled by the runtime as new class files are loaded into the system. To facilitate load time validation, we have developed an abstract structure we call a *validation registration*. The format of a *validation registration* is shown in Figure 5.9. The basic fields are the class file name for the class load event needing validation, the type of validation, and the type of rollback and recovery needed. The type of validation can either be a coarse grain validation such as the occurrence of the load event itself, or a fine grain validation such as particular property of a field or method of the class loaded by the load event.

For example, the caching of the return value for `size` described in Section 5.1.4, was based on the assumption that the CH was closed. Furthermore, it was observed that within the closed CH, none of the procedures called within the body of the loop changed the value of `size`. If a coarse grain validation is used, then any loading of a new subclass of `ListClazz` is sufficient to trigger a validation failure event. However, our framework is designed to do better than this. In our framework, fine grain validation can be specified, requiring that a validation failure event only occur if the newly loaded

subclass of `ListClazz` contain a new version of `getElement` and the new version affects in some way the value returned by `size`. The validation of this information can be derived from the CDGs for the newly loaded procedures. Most fine grain validation events use the information in the CDG/ACG to perform the validation, meaning that in most cases, without the CDG this type of fine grain validation would not be practical. Therefore, the CDG/ACG can be viewed as an enabling technology for fine grain validation.

Validation failure from a load event could occur while the execution of the affected region is in one of the following two states:

- *preregion*: Not yet entered the affected optimized region of the application.
- *in-region*: In the process of executing the affected optimized region of the application.

Note the absence of a postregion state from the necessity for rollback and recovery. Since the event affecting the validation failure occurred after the region had been executed, it could not have affected that region of the program. If the region is to be re-executed, then the state can be handled as *preregion*.

5.2.1 Preregion execution state

In the *preregion* execution state, rollback is not necessary since the region of code has not yet been executed. Instead, the recovery mechanism can simply replace the entry instruction for the image of the optimized code region with a stub redirecting future callees to the unoptimized version of the code. It incurs the added overhead of the

redirection; however, this overhead can be mitigated, if not completely eliminated, at the next optimization epic.

Figure 5.10 shows an abstract view of how this mechanism works. The original code after the optimizer runs is laid out in memory according to Figure 5.10(a). Here we have shown three optimized versions of the same region of code. Note that each optimized version is branching to a continuation point, `0x1dff`. Whether or not the code at the branch target for the continuation point is optimized is immaterial, so we have not specified it. Note also that the original code continues execution on through the branch target (continuation code) for the three optimized segments. When a validation failure event causes a stub insertion, the conceptual approach is to place a hard jump at the entry point for the region. This is shown in Figure 5.10(b) with the jump to the starting address of the block marked original. This jump instruction can overwrite the entry instruction(s) in the block since the remainder of the block's instructions are now dead code.

There are some additional assumptions made in this abstract representation. For example, we assume that the state of the program upon entry to each of the optimized versions is identical to the state of the program should we have entered the original code version. If it is not, then the stub must contain any necessary patch code to bring this state back to the state the program would have been in had the original code been entered. For example, if one or more of the optimized versions used stack allocated object instances and these object instances were assumed heap allocated in the original code, then the redirect may not be directly to the original version of the code. Rather, the

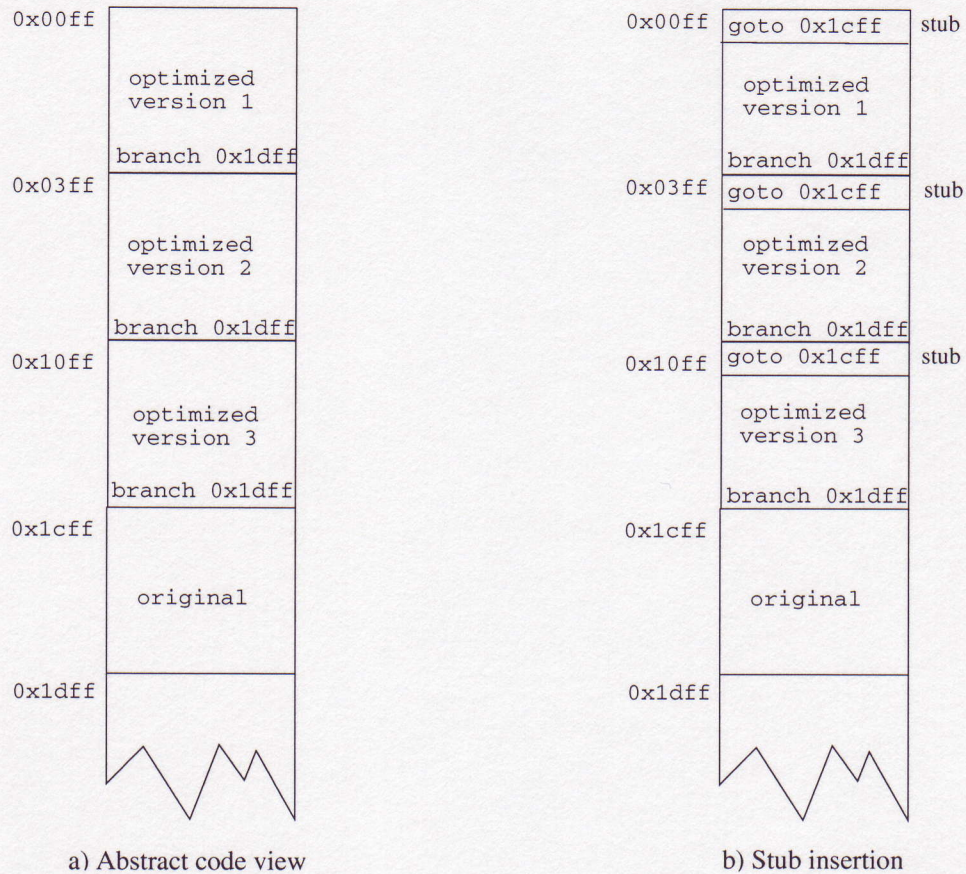


Figure 5.10 Abstract view of code space and insertion of redirection stub.

redirection stub would be to a patch code region that flushes the stack allocated object instances prior to continuing. This patch code could be unique for each of the three versions of optimized code shown in Figure 5.10. Since the redirection is in the target entry point for the optimized region, all other code which had this target is unaffected by the change.

Additionally, this model requires maintenance of the original code. However, most runtimes do keep the bytecode version of the code available at all times. Therefore, as

long as the optimized code and the unoptimized code share the same memory model, call stack and code space, this level of rollback is achievable in the system.

5.2.2 In-region execution state

The in-region execution state is a little more complex with the required rollback and recovery being directly related to the aggressiveness of the optimization. We have identified two approaches to handling executing regions of code when a newly loaded class file causes a validation failure event. The first approach, continue executing, contends that the event cannot affect the executing code and can only affect future entries into the region. Therefore, do nothing to the executing code and just insert stubs in the optimized region for new entries into the region. This is similar to the stub described in Section 5.2.1. However, we may be able to get more aggressive with the optimizations if we can roll back and re-execute. Therefore, the second approach, checkpoint, rollback, and re-execute, contends that validation failures are rare events; therefore, the cost of check-pointing, then rolling back the program's state and re-executing the affected region, is acceptable. This approach requires extra overhead and recovery code for executing methods.

5.2.2.1 Approach 1: continue executing

This model allows the region to complete and is only concerned with redirecting future entries into the region. Note that this model contends that procedures are located via a specific object or class instance. There is a limited means in which a reference to an

object instance can be obtained in an executing application. These limited forms are as follows.

1. *Allocation*: The actual creation of a new object.
2. *Field Access*: The obtaining of an object reference via the reference stored in another object.
3. *Global Access*: The obtaining of an object reference via the global field of a class object.
4. *Formal Parameter*: The reference was passed to the method via a formal parameter.
5. *Callee Return*: The reference was returned from a callee invocation.

Since this set is finite, and the behavior of the object instances is fully specified in the CDG for each procedure, we are able to track the state of the object instances involved in the optimization. The main restriction placed on optimizations under this model is that object instances involved in the optimization must be definitively known not to escape either prior to or during the region being optimized. This means object instances shared among threads or object instances crossing into speculative nodes in the ACG are excluded from optimization.

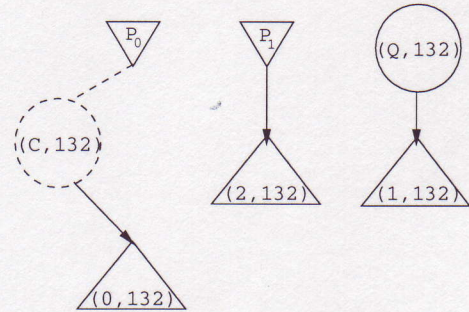
To illustrate why this restriction is necessary, consider again the example class in Figure 3.9 in which we introduce a user, `producerQueue` shown in Figure 5.11. At optimization time, the only implementation of `ListClazzUser` is the one shown in Figure 3.9. The field `Q` of `producerQueue` is declared as `static` and by definition *thread escaping*.


```

121: class producerQueue{
122:     ListClassUser C;
123:     static ListClazz Q;
124:     <producerQueue>(producerQueue o){
125:         o.C = new ListClassUser();
126:         o.C.<ListClazzUser>();
127:         o.Q = new ListClazz();
128:         o.Q.<ListClazz>();
129:     }
130:
131:     void addJobs(producerQueue o, ListClazz J){
132:         o.C.append(producerQueue.Q, J);
133:     }
134: }

```

(a) class file



(b) CDG for addJobs

Figure 5.11 A user class for the class file `ListClazzUser` in Figure 3.9.

Since the field `Q` is *thread escaping*, the items stored on it are not under the sole control of the executing thread. Under the aggressive speculative model, the optimizer may want to inline `append` and subsequently `getElement` then `add`. However, if while the application is running, a new subclass of `ListClazz` gets loaded, the results from executing the optimized code may vary from those of an unoptimized version. It is possible for a new object instance of the new subclass of `ListClazz` to be on the `Q` list. The new subclass may use different versions of `getElement` and/or `add`, meaning if the execution of `addJobs` is allowed to complete, the result of the execution may not match the unoptimized version. Therefore, under this model, the determination that `Q` is escaping would prevent the optimization.

In order to assure a reasonable level of reliability, optimizations based on this recovery model are limited to reference fields known to be *thread local* and in some cases, the stricter *definitively known* subset.

5.2.2.2 Approach 2: checkpoint, rollback, and re-execute

The second approach enables more aggressive optimizations. The concept is to checkpoint the state of the execution prior to entry to the region. If a validation failure event occurs, roll back to the check-pointed state and redirect execution to the unoptimized version of the code. This is similar in concept to the speculation mechanisms used in most compilers and processors. As long as state has not been committed, this model works. It does incur the cost of the checkpoint operation and may involve several checkpoints and rollbacks for a given region.

To illustrate, we use again the `producerQueue` class in Figure 5.11. Again we assume the CH is closed. This time we perform the aggressive speculative optimizations described in Section 5.2.2.1 that were prohibited under that model. Now the determination that `Q` is escaping is not an automatic disqualification of the optimization. Instead, further analysis is used to determine how involved the check pointing would need to be should a rollback and recovery be necessary.

In the case of `producerQueue`, we can observe from the code that we are only making a copy of `Q` and not affecting `Q`. This observation can be extracted from examination of the iCDG/ACG for `addJobs`. The iCDG is shown in Figure 5.12. To observe that `Q` is only read, we can reduce the iCDG by removing all property nodes, then bypassing and removing the field nodes. The steps and the final reduced version are shown in Figure 5.13. From observation of Figure 5.13(e), it is seen that there are only arcs leaving the node `Q`. Therefore, we can accurately conclude that we only read `Q`. Additionally, `Q`

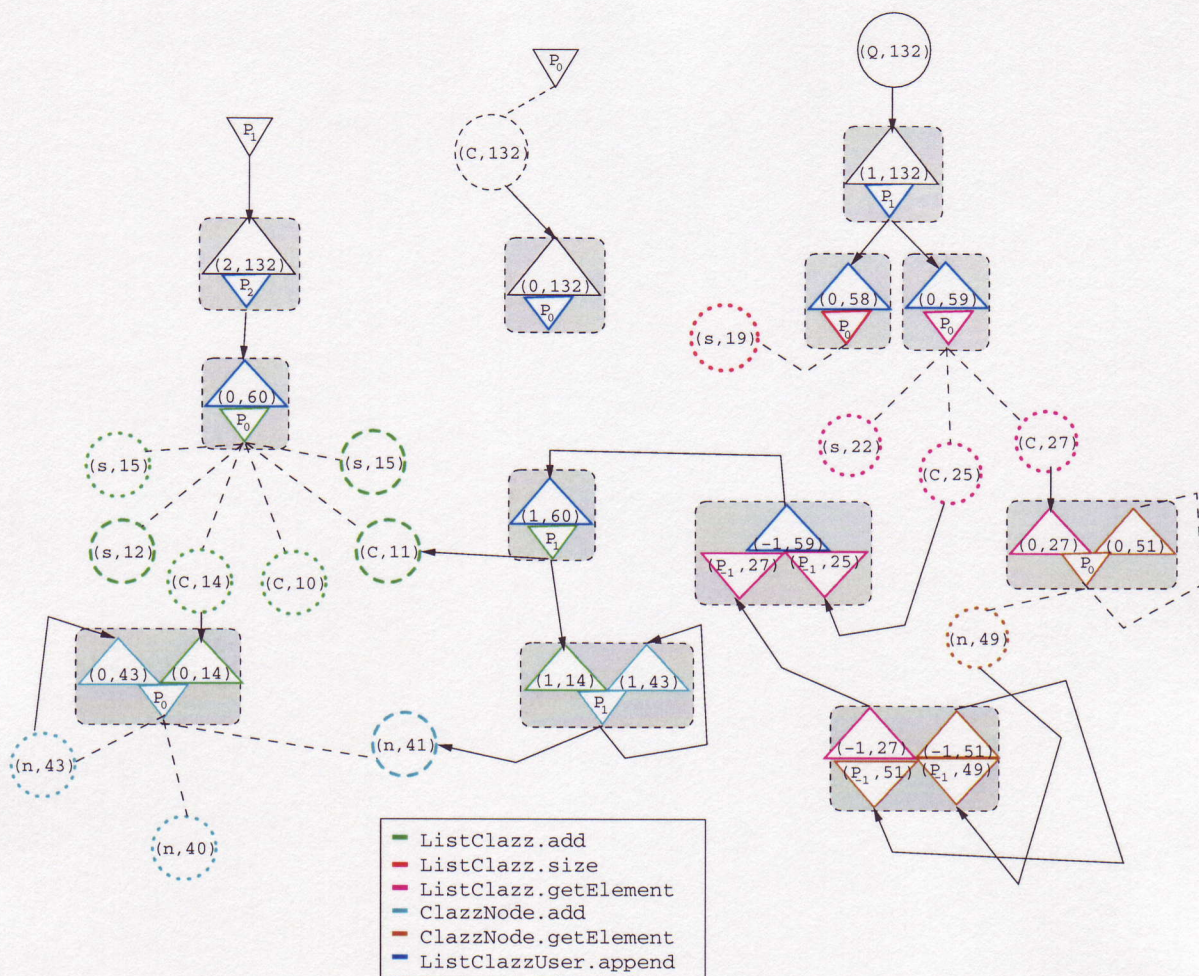
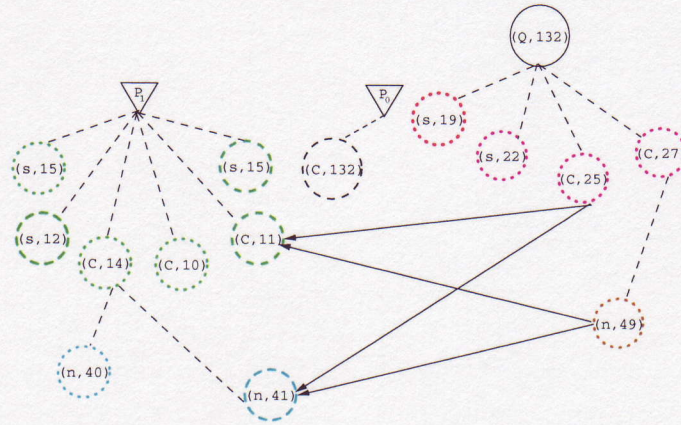
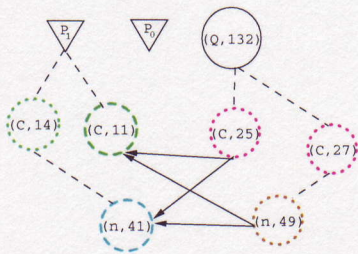


Figure 5.12 The iCDG for the procedure `addJobs` shown in Figure 5.11.

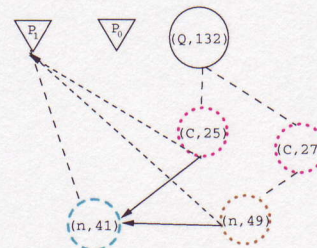
is written to a field in formal node P_1 . From the information contained in the iCDG of Figure 5.13(a), the optimizer can determine that the necessary check-pointing is the copying of the original contents of the field C and its field n prior to execution of the region. Then if a validation failure event occurs, the original values in these fields can be restored, any new objects created, discarded, and an unoptimized version of the code re-executed. The validation registration for this optimized region would consist of the



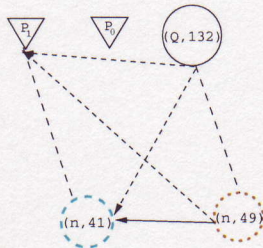
(a) initial iCDG with super nodes removed



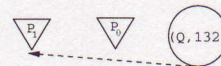
(b) dangling field nodes removed



(c) nodes (C,14) and (C,11) removed



(d) nodes (C,25) and (C,27) removed



(e) all field nodes removed

Figure 5.13 The intermediate graphs and final iCDG after removing field nodes.

class file names `ListClazz` and `ListClazzUser`. It would also specify that the type was coarse grain since any new version of these class files could invalidate the optimization. The rollback/recovery would specify both a *code* and a *call stack* entry. The code entry would consist of the same redirect to original version of the code discussed earlier. For the call stack entries, a stub could be inserted at the previous return points in the optimized code. This would then cover future entries into the region and returning entries.

For the case of executing code, the runtime must be able to signal the execution that a validation failure event has occurred. One way to signal the execution is by use of a flag. The optimized version of the code could check the flag prior to a commit of the state changes. If the flag is set, do not commit, but roll back to the original version and re-execute. Note that since the optimized code entry point has been replaced with an unconditional branch to the original code, the re-execution will automatically redirect to the original. Since only the optimized version of the code would check this flag, the setting of it would have no effect on the unoptimized version.

However, there may be no definitive way to determine if all optimized code segments have completed executing without examining the call stacks of every thread. Therefore, it may be difficult to determine when it is safe to clear the flag. For this reason, it may be desirable to use a flag located within the code space of the optimized code segment.

Although these models cover the state of the runtime at the time of the validation failure event, they do not fully discuss the intricacies of specific optimizations.

5.3 Additional Optimization Examples

Optimization concerns and validation of optimizations can also be specific to the type of optimization employed. In this section, we cover some of the types of optimizations introduced in Chapter 3 and what forms of validation would be necessary to enable aggressive optimizations. Additionally, we address optimization implementation concerns. Where appropriate, we show how the information contained with the CDG could be used to specify a fine grain validation, thus potentially expanding the lifetime of the optimization.

5.3.1 Stack allocation of object instances

In Chapter 3 we introduced the use of the iCDG/ACG for the discovery of stack allocatable object instances. In Chapter 4 we focused on the analysis necessary for discovering stack allocatable object instances at first allocation. In this section, we now address optimization implementation models and concerns for the implementation of stack allocation optimizations.

First, if the runtime is using an interpreter based start-up, then the interpreter must be *stack allocation aware*. This means that instructions such as `getfield` and `putfield` that used to go through the memory manager to the heap, now need to be redirected to use the stack. Ideally, we would like to replace these instructions with instructions that are specific to the runtime's interpreter and identify these as stack allocations and accesses. This can be accomplished within the current bytecode code space. Note that

in the Java Virtual Machine specification, only 205 of the available 256 unique bytecodes have been assigned [29]. This gives 51 available unique bytecodes within the instruction space. The use of the unassigned bytecode space allows the runtime to replace in the code array for the method, the original heap accessing instruction with the new stack local accessing version. This can be done swiftly upon entry to the method if the CDG is used. Since the CDG contains the bytecode address of each access by a heap-accessing object instance, and it also associates these with their given object instances, the runtime has the necessary information to implement this change. Furthermore, since the use of the “special” instructions is internal to the runtime and not visible outside of it, it does not affect other runtime implementations. Additionally, if future changes to the Java specifications use this address space, the runtime will need to be redesigned to accommodate the new instruction and at the same time the “special” instructions can be adapted. The original specifications showed a similar use of this unassigned bytecode space with the use of the “quick bytecodes” by the original implementation of the interpreter [38].

However, not all calling contexts have the same results. Referring back to Figure 3.5 in Chapter 3, the decision on which object instances to stack allocate was based on the type of the object instance used to locate `Hoe`. Therefore, the heap accessing bytecode instructions that need replacement with stack accessing counterparts are dependent on the runtime type of `Clazz o`. Since the decision of which bytecodes to replace is made based on calling context, then the code may only be applicable to this one use.

One way to handle this is to have a main code buffer for the original bytecode image of the code. The class's method table would point to the main code buffer version. When an analysis such as the iOCG/ACG described in Chapter 4 identifies stack allocatable object instances, a specialized version of the method could be created – for example, copying the main code buffer contents to a new memory location, replacing the instructions used to access the now stack allocated object instance, and returning a pointer to this new location. When the method's execution completes, since this was calling context specific, the code space is collected. This can be done as a trigger on the call stack, thus allowing the code space used to house the specialized versions to be collected. Collecting and regenerating the optimized segments can prove expensive. Therefore, in Chapter 7, we address some future work that may allow efficient context matching and reuse some of the optimized methods.

5.3.1.1 Flushing of stack allocated object instances

For optimization time stack allocation decisions described in Chapter 3, the decision is based on the model being used – always safe, sometimes safe, or speculative. However, in the event of a validation failure, the object instance may need to be moved from the stack to the heap. This can involve additional overhead both in the implementation and the design.

If object instances have been allocated on the stack instead of through the heap, then when flushing the object instance, all reference fields must also be updated. Therefore, the decision as to which object instances to stack allocate needs to be weighed against

the recovery cost of validation failure events including the expected failure frequency. Therefore, once an object instance has been identified as stack allocatable, the next important criterion for determining whether or not to stack allocate it is the type of fields it contains. We classify the object type criteria for stack allocation progressing from easiest to implement and recover from to hardest, as follows:

1. All fields of the object are primitive values.
2. One or more reference fields in the object but all reference fields for the given object instance point to stack allocated object instances.
3. One or more reference fields in the object and the reference fields may point to stack or heap allocated object instances.

For the first case, all fields are primitive: should the object instance need flushing, then only the stack allocated object instance is affected. Therefore, the flushing is a simple copy from the stack to the heap. For the second case, all reference fields point to other stack allocated object instances, then the flushing of one object instance may have a ripple affect. For example, if other stack allocated object instances point to it, then they also must be flushed to the heap for this property to be maintained. Note that the efficient implementation of such a property may require the use of double-ended pointers to keep track of all object instances with fields pointing to a given object instance. This overhead could mitigate any benefit from the optimization. The third case complicates the second even further. Note that when the memory manager collects unused memory locations, it often moves object instances within the memory space. Under this model,

even if an object instance that had been stack allocated should remain stack allocated, then at every garbage collection epoch, the reference fields accessing heap allocated object instances would need updating. Therefore, when implementing a speculative optimization model, it may be desirable to restrict stack allocated object instances to case 1.

Although stack allocation of object instances has the potential to show significant performance improvement, care should be taken when applying the speculative optimization model to this form of optimization so as to not mitigate the benefits.

5.3.2 Synchronization removal

In Section 3.4.2, we showed how the iCDG could be used to identify redundant and unnecessary synchronization operations. Furthermore, we showed how these operations could be eliminated. In this section we address the types of validation, rollback, and recovery necessary for both the sometimes safe and speculative optimization models.

In the *sometimes safe* model, nested synchronization may be removed along one of the paths but not the other path. This is because the other path is assumed unbounded, meaning the set of potential targets for the callee could expand at any instance. Furthermore, the expanded set has the potential of allowing to escape an object instance that was thread-local based on the iCDG when the code was optimized. Therefore, in order to enable aggressive optimizations, the *sometimes safe* model needs to check the escaping state of references passed across the callee boundary to determine if it is safe to continue in, or possibly even return to the optimized version of the code. This level of validation can again be accomplished using the CDGs for the callees.


```

520: class syncClazzA extends syncClazz {
521:     synchronized void Foo(syncClazz o, syncClazz p, syncClazz q){
522:         syncClazz r = new syncClazzA();
523:         <syncClazz>(r);
524:         synchronized(r){
525:             q = r;
526:             o.f.Bar(p, q);
527:         }
528:     }
529: }
530:

```

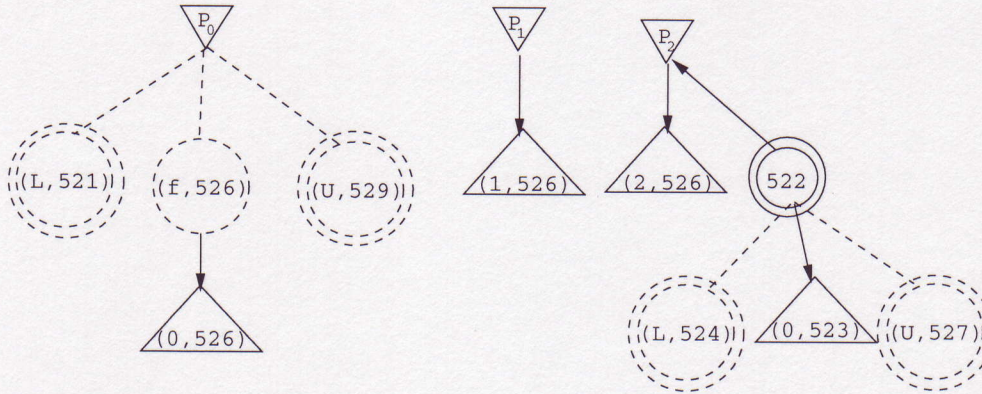


Figure 5.14 A new subclass of syncClazz in Figure 3.6.

Figure 5.14 shows a new synchronized class file that extends the `syncClazz` introduced in Figure 3.6. At the time the optimizer runs, `syncClazz` and `syncClazzA` have been loaded by the runtime. While optimizing `Hoe`, the optimizer makes the decision under the sometimes safe model, to inline the `syncClazzA` version of `Foo`. The optimizer then makes the decision to inline the context monomorphic version of `Bar` when the context is from classes `SyncClazz` or `SyncClazzA`. Once inlined, the optimizer is able to remove synchronization operations from the call to `Foo`, and `Bar`, as well as the synchronized block within the `syncClazzA` implementation of `Foo`. However, the internal removal

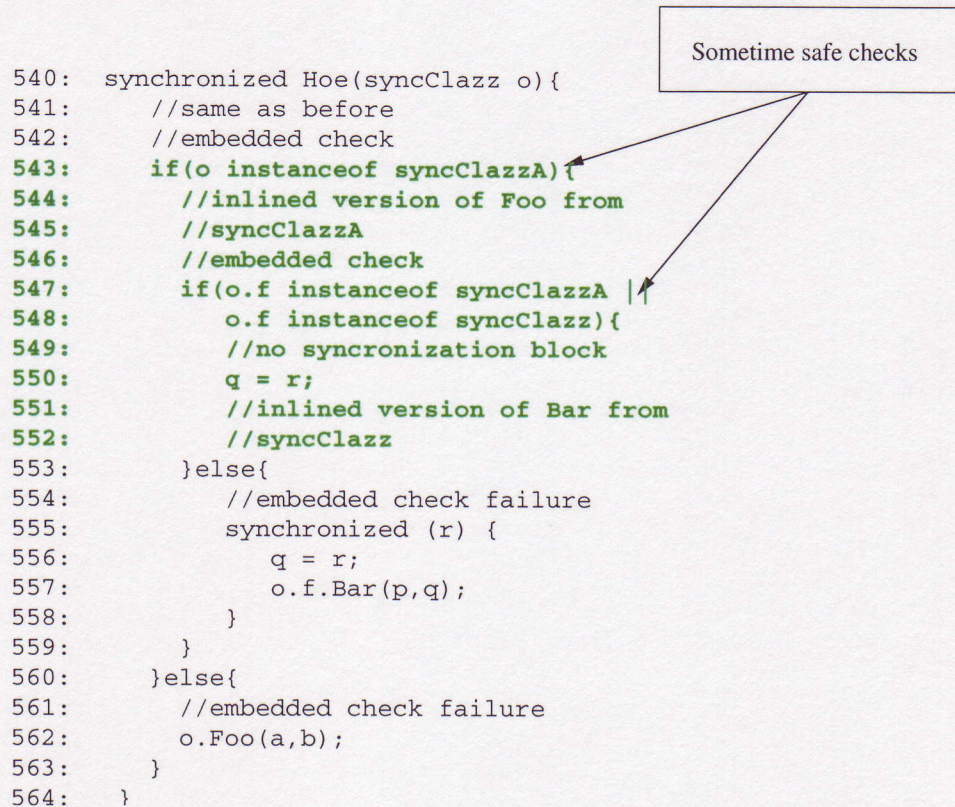


Figure 5.15 The conceptual view of the *sometimes safe* inlined version of Hoe.

of the synchronization block is only done on one path. On the alternative path, the synchronization is still present.

Figure 5.15 shows a conceptual view of the *sometimes safe* optimization model for this form of optimization. Note that there are two embedded validation checks, one at line 543 to determine if the inlined version of Foo is valid, and one on lines 547 and 548 to determine if the inlined version of Bar is correct. For the second check, on the validation failure branch, the synchronization block remains intact. The second branch


```

565: synchronized Hoe(syncClazz o){
566:     //same as before
567:     //embedded check
568:     if(o instanceof syncClazzA){
569:         //inlined version of Foo from
570:         //syncClazzA
571:         //embedded check
572:         if(o.f instanceof syncClazzA ||
573:            o.f instanceof syncClazz){
574:             //no synchronization block
575:             q = r;
576:             //inlined version of Bar from
577:             //syncClazz
578:         }else{
579:             //embedded check failure
580:             if(Formal 2 in the CDG of Bar does not escape){
581:                 q = r;
582:                 o.f.Bar(p,q);
583:             }else{
584:                 synchronized(r){
585:                     q = r;
586:                     o.f.Bar(p,q);
587:                 }
588:             }
589:         }
590:     }else{
591:         //embedded check failure
592:         o.Foo(a,b);
593:     }
594: }

```

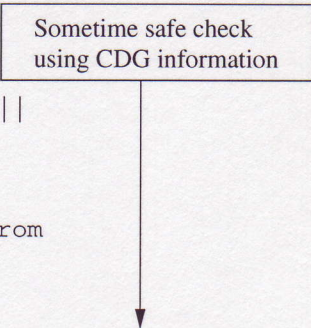


Figure 5.16 The conceptual view of the *sometimes safe* inlined version of Hoe utilizing the CDG.

could have been optimized further by eliminating the synchronization block there as well if the CDG for any new version of Bar is checked.

Figure 5.16 now shows a *sometimes safe* version of Hoe which becomes more aggressive by exploiting CDG information. In this version we perform not just a type check on `o.f`, but in the event this type check fails, we perform a property check on the CDG of the new version of Bar. On line 580, the optimized version checks the CDG to see if the new

version of `Bar` still retains the property allowing the synchronization removal. If it does not, then a version with the synchronization block is called.

5.3.3 Code motion after inlining

Sometimes even though an optimization is *sometime safe*, the information in the iCDG can be used to help guide code motion decisions. For example, in the *sometimes safe* optimized version of `Hoe` shown in Figure 5.16, the assignment `q = r` occurs in each of the branches. Furthermore, from the CDG for the `syncClazzA` version of `Foo` shown in Figure 5.14 it can be seen that the value of `r` remains *thread local* in the region preceding the call to `Bar`. Therefore, the assignment statement can be moved out of the blocks and above the first validation check at line 573. This new version of `Hoe` is shown in Figure 5.17. The optimizer can employ similar uses of the CDG information to perform code motion and potentially improve the level of optimization.

5.4 Conclusions

In this chapter we have presented many new concepts and ideas. We have classified not only the types of optimizations but also developed models for validation, rollback, and recovery. We have shown, in most cases, how the CDG, ACG, and subsequent iCDG can be used to implement several forms of validation. We have motivated the design of this validation, rollback, and recovery system with intuitive examples. In Chapter 7, we discuss future work for implementing this portion of the framework.


```

565: synchronized Hoe(syncClazz o){
566:     //same as before
567:     //embedded check
568:     if(o instanceof syncClazzA){
569:         //inlined version of Foo from
570:         //syncClazzA
571:         //embedded check
572:         //code motion of q = r
573:         q = r;
574:         if(o.f instanceof syncClazzA ||
575:            o.f instanceof syncClazz){
576:             //no sync block inlined version
577:             //of Bar from syncClazz
578:         } else if(Formal 2 in the CDG of Bar does not escape){
579:             //moved q = r
580:             o.f.Bar(p,q);
581:         }else{
582:             synchronized(r){
583:                 //moved q = r
584:                 o.f.Bar(p,q);
585:             }
586:         }
587:     }
588:     else{
589:         //embedded check failure
590:         o.Foo(a,b);
591:     }
592: }

```

Code motion after inlining

Figure 5.17 The new version of Hoe after code motion for Figure 5.14.

CHAPTER 6

RELATED WORK

The research presented in this dissertation builds upon the past research of several researchers. First, in the field of program analysis several of the techniques used here were developed from research in the field of object oriented program analysis. Although the majority of this work related to languages such as C++ or SmallTalk, it was still partially applicable to this problem domain. Additionally, over the past several years, many researchers have mapped Java back into a “closed-world” model in order to explore the opportunities for optimization. Some of these researchers have expanded into the realm of dynamic analysis focusing on a subset of the problem set. In addition, we have designed specific runtimes additions and therefore the design of Java runtime implementations is also related to our work. In this chapter, we review the previous research most closely related to the work here. We compare our work and design with the work of these researchers.

6.1 Static Analysis

Research into the use and behavior of references within an application has been an active topic for several years. The focus has ranged from trying to track the behavior

of references in programs, to trying to determine an efficient call graph for polymorphic languages, thus reducing the analysis space. However, reference tracking or alias analysis can be complicated in object oriented languages by the inclusion of reference fields within objects. Deutsch, [7] presents a technique for alias analysis that actually attempts to follow the fields within structures. It uses a numbering technique to identify the fields and works well when the set of class files is closed. However, in a dynamically loaded application, this assumption is not valid. Diwan et al. [39] describes a type based alias analysis for modulo-3 programs and its extensions into Java. They start by showing results for an analysis based only on type information. They then apply several improvements from field disambiguation to, finally, flow sensitivity. They also demonstrate that type information in strongly typed languages such as Java can be used to reduce the conservativeness of assumptions when optimizing an incomplete program. Furthermore, they show that parameter types can be used to eliminate potential aliases from the set of aliases. We have incorporated both the field disambiguation and the flow sensitivity in the CDG representation. Furthermore, we do not try to eliminate potential aliases. This is because our interprocedural analysis is dynamic, and in most cases the elimination of extraneous classes is an artifact of the system.

6.1.1 Class hierarchy and call graphs

Restricting the input set of class files in a polymorphic application is critical in static analysis in order to bound the analysis space. Reducing the number of class files assumed used by an application can also reduce the amount of conservatism in the analysis and

thus increase the precision of the results. DeFouw et al. [8] try to restrict the set of class files needed for the call graph construction by computing a set of classes that a runtime variable can resolve to. They then use this to cull out unreachable procedures from the call graph. The method is successful in reducing the size of the call graph when the full set of files used by an application can be assumed closed. Grove et al. [10] focus on the same problem space, reducing the number of targets in a call graph for a polymorphic application. They studied Cecil and C++ applications and found that the target of virtual function calls within a given application tended to have a very peaked distribution. Furthermore, they discovered that this distribution remained relatively consistent through several generations of the application. Although static, this finding may extrapolate to changes in a dynamic application. As such, the information could be exploited to reduce some of the overhead in our system. We address this in our future work section.

Bacon and Sweeney [12] compared three static methods of virtual function resolution: unique name (UN), class hierarchy analysis (CHA), and rapid type analysis (RTA). The UN is a link time algorithm and looks for unique names within the object files to eliminate those calls as being virtual. CHA uses the statically declared types of the objects in combination with the class hierarchy of the program to determine virtual method resolution. RTA improves upon CHA by using instantiated type information to reduce the potential call sites in the graph. All three depend and require access to the full code for the application. The introduction of new subtypes or code dynamically to any of the methods could cause the analysis results to be incorrect. For an application that has

full code knowledge, the RTA performed the best of the three methods with comparable costs to the CHA. As we addressed in Chapter 3 the dynamic nature of our problem space automatically culls a large portion of the class hierarchy for us. Although the full dynamic class hierarchy may be too much for a smaller region of the application, the use of CDG information when resolving targets in the ACG/iCDG construct eliminates additional potential targets. Therefore, in our model, we do not attempt to employ these techniques.

Calder and Grunwald [11] showed that profile information can accurately predict targets of indirect function calls, thus reducing the costs of resolution. By resolving the target of indirect calls via profiling, the calls for the common target can be inlined with a check used for the uncommon branch. They showed that this allows not only the optimizations normally associated with inlining but also allows the branch target hardware to predict the branch and eliminate the need to stall while the indirect call is resolved. This is similar to our *sometime safe* model of optimization. The interesting finding in their work is the effect of the check on the branch target buffer (BTB), thus suggesting the costs may be mitigated by standard hardware features. We have not examined this closely and consider an investigation of exploiting the BTB and potentially predicate registers in a given hardware design to facilitate some of the validation and optimization models as future work.

Sundaresan et al. [21] build a call graph starting with a class hierarchy graph of all the class files in the application. They approximate the runtime types of the receivers when building the call graph. They refine RTA by using a reachability graph. The graph

maps assignments to the types that can possibly reach a given node N . Therefore, a type must not only be instantiated in the program but must also have a valid path to the node in the call graph. They also collapse strongly connected components in their graph using a set of types to represent the type of the new node. They show that with the mapping back to class type and the collapsed nodes, their analysis can be performed in linear time. Again, their work hinges on a static “closed-world” view of the application. There may be some gain during the construction of the ACG/iCDG by employing these techniques to further reduce the set of potential targets.

Grove et al. [14] developed a formal specification for call graphs based on a lattice model. They then used this specification to evaluate different context sensitive and context insensitive call graph construction algorithms. They found, at the time, that interprocedural analysis had a noticeable impact on Cecil applications but not as substantial an impact on Java applications. However, this work was conducted soon after the introduction of Java as a programming language. Java applications have grown substantially in both size and complexity since this early work. They have also increased in the level and degree to which they employ the polymorphic properties of the language thus making their behavior closer to the Cecil programs studied in this work. They did find that flow sensitivity improved the results but did not scale well with the level of improvement versus the increase complexity. In our design, we incorporate the flow sensitivity in the intraprocedural analysis phase. We then store information on all potential definitions that can reach a given use in our CDG representation. Combining this with other techniques, the runtime can choose the desired amount of interprocedural flow

sensitivity dependent on the cost constraints of the system. This allows the runtime to incorporate a reasonable level of flow sensitivity into the particular analysis, scaling it as desired. We have not done a cost/benefit analysis on how much gain there is from different levels of flow sensitivity during the ACG/iCDG construction phase. This is part of the future work. Grove et al. also found that interprocedural escape analysis was sufficient in enabling allocation optimization. They also suggested there could be a substantial gain from incremental algorithms not only for scalability but for handling dynamic class loading features.

6.1.2 Interprocedural propagation

Object oriented programming styles tend to use polymorphism to implement control decisions. This tends to make the program decision logic more call-based than languages such as C. Therefore, the need for interprocedural analysis in order to optimize becomes more critical. Research into the analysis of object oriented applications has centered around the need for interprocedural analysis and the need to answer if not the general question of how references or data behaves, then a specific question.

Burke [40] deals with interprocedural analysis in Pascal and Fortran style programs. They use summary functions to represent the intraprocedural information and then propagate the information interprocedurally using the resulting summary function. The use of intraprocedural summary functions is similar in concept to our use of CDGs. However, the information we include in the CDG goes beyond that of a standard summary function, thus enabling a wider range of analysis and, ultimately, optimizations.

Chatterjee et al. [2] use a similar approach with summaries of the intraprocedural information. They adapt the approach to polymorphic languages forming a class hierarchy and conservative call graph for the interprocedural propagation phase. We take a similar approach with our use of CDGs for the procedures and the combining of the CDGs interprocedurally to find a solution. However, the dataflow element (*dfelm*) used in their analysis represents only points to pairs and potential pairs either intraprocedurally or propagated interprocedurally. They do not retain the information on the type or use, which we do retain in the CDG. The additional information contained within our CDG representation allows us to utilize it for a wider range of analysis as detailed in Chapters 3 and 5. Additionally, because they have a closed worldview, they combine the information from all potential call sites at a given node in the call graph. We, instead, only try to combine in this fashion in the speculative optimization model. In our other models, we avoid combining the CDG information from multiple potential targets when forming the iCDG. Goodwin [9] also addresses the problem of interprocedural analysis. He uses the analysis to compute live registers and thus improve register allocation in an actual optimizer. He handles the dynamic portion of the application, such as calls to dynamically linked libraries, conservatively. Since we delay the interprocedural phase until runtime, our analysis can eliminate some if not all of this conservatism, making our results potentially more precise.

However, sometimes a general solution is more than is necessary. Another form of static analysis that is similar to the work we have done is that of program slicing. A program slice is a form of analysis where the state of a program is examined at a given

point in order to answer a specific question. This is similar in concept to the work we presented in Chapter 4 where we looked at a specific context of the execution state to answer the specific questions concerning stack allocation of object instances. Duesterwald et al. [5] and Horwitz et al. [24] both present techniques to solve interprocedural equations for specific answers by walking backwards through the call graph and terminating once an answer is found. Reps and Rosay [6] describe a form of programming slicing called chopping. They use the chop or slice of a program at a given point with a given state to answer a specific question. They do assume that they have a closed worldview of the application.

6.2 Java Specific Research

Early research into Java applications approached the problem by applying techniques from earlier research into object oriented languages. They mapped the dynamic application state of Java back into a closed worldview of the application and focused on areas that set Java apart from the other languages.

6.2.1 Escape analysis

One key concept soon surfaced as a delineating factor in Java applications and that was the use of threads. Research soon focused on how to determine which object instances were shared among threads and which remained local to a given thread. Blanchet [15] published one of the first works in this area and used the term “escape analysis” to de-

scribe his technique. One of his primary contributions is the use of height information to represent the fields and relationships between objects in a program. He then constructed the full call graph for the application, and used a forward and reverse traversal algorithm using iteration to settle strongly connected subgraphs. This is similar to how we settled the strongly connected components in the ACG/iCDG construction. The difference being that he is using a points-to relation and connecting the intraprocedural information conservatively relying on the type height values to cull out extraneous procedures. We, on the other hand, contain more than just points-to information in the CDG. Therefore, we only connect the components if the context matches and stop at a maximum depth otherwise. His work focused on method local determination and stack allocation of object instances. One of the first works to show a performance improvement in this area, his system was implemented in turbo-J which converts Java to C code. The main drawback of his work is that it assumes a closed worldview of the application when making the height determination for the fields and the relationships between the objects. It is not clear from this work how the technique can be adapted when the set of class files may change. The dynamic changes could invalidate height assignments by enclosing new cyclic regions into the class hierarchy representation he uses.

Aldrich et al. [20] also present a method of performing escape analysis on a closed worldview of a Java application. They use the results of the escape analysis to eliminate redundant synchronization. They are also able to eliminate synchronization on thread local objects. They showed a significant performance gain could be achieved if these redundant operations were discovered and eliminated. Ruf [16] presents a similar tech-

nique using a closed-world static view of a Java application to perform escape analysis for synchronization removal. What made his work unique is that he does not restrict synchronization removal to only thread local object instances but also tracks who actually locks each unique object, whether or not it has escaped the control of a single thread. Because he uses a closed worldview, he is able to perform this level of determination, and therefore can remove synchronization from global objects that are provably only synchronized by one thread. The call graph he uses employs a form of method specialization to handle calling context by duplicating the method summary at each call site and unifying it for both the caller and callee. This is similar in concept to our *speculative* optimization portion of the mixed model described in Section 5.1.4 of Chapter 5. He uses context sensitivity in his alias analysis tracking, which allows him to do a better job of identifying objects that are only synchronized by one thread. We showed in Chapter 3 how our CDG/ACG formulation could be used to perform a similar form of escape analysis on a dynamic application. By using the CDG/ACG, the closed-world assumption of these previous researchers can be relaxed. We then showed in Chapter 5 how our system enabled aggressive optimizations by incorporating the appropriate validation and rollback mechanisms.

Rountev et al. [18] describe a method of doing points-to analysis for Java programs in such a way as to annotate the information in the points-to graph to include more detailed method and field information. By doing this, the authors show they can reduce the space required to hold the points-to graphs since the field information results in smaller graphs. With the smaller graph, they can also improve the accuracy of call

graph construction over RTA since the method information allows for better resolution of virtual function calls. They also show that the information can be used to identify method local and thread local objects. They assume access to the entire application code and do not address the problems associated with dynamic class discovery. Although we include more specific use information about the object instance, we also include field information in the CDG. The points-to relation we derive from the CDG, as described in Chapter 4, folds out this information to enable a very swift interprocedural propagation. However, because the field information is present, their form of points-to relation could be derived from the CDG if so desired.

Choi et al. [25] describe a method of performing escape analysis that uses an abstraction called a connection graph to represent the interrelationships between objects. Nodes in the graph contain a set of information pointers that point to fields of the object: a points-to edge that is established at a creation point, and deferred edges for assignment statements. A set of possible definitions for a variable are located by traversing all of its deferred edges until they terminate in points-to edges. They talk about the notion of bottom graphs to handle methods for which they do not yet have an analysis, such as native methods or methods not yet loaded. They say they exploit the strong type system in Java to resolve these. They do not talk about how they handle the dynamic class file location and creation in Java where a new subclass could appear at any moment. This would mean that almost all of the methods have the potential of being bottom methods. They do not state how they handle the escaping value of object instances that could be passed to bottom methods.

Bogda and Hülzle [17] present an interesting method of performing a quick version of coarse grain escape analysis. They ignore the control flow in the first pass, only looking for certain types of expressions and setting an escaping property based on the expression. They define the notion of s- escaping, that a reference is stored in a heap object and therefore the reference value escapes the stack. With this definition, they identify expressions of the form $x.f = y$ and $y = x.f$ as s- escaping and propagate this information to formal parameters of calls. The assignment must be to an already s- escaping reference or to a class object (static). Exceptions are treated as s-escaping and arrays are treated as one whole object. They perform the analysis on the OSUIF intermediate representation, which has already transformed the code into expression trees based on a static closure of the application. The static closure conservatively maps all polymorphic call sites to the full set of legal methods for the resolution. The technique provides a conservative and safe solution to the problem space. We have taken a different approach in that we identify these types of expressions as *reference affecting* during the intraprocedural analysis phase. We, however, do not try to propagate properties during this phase, but rather compactly represent the relations and properties in the CDG. This enables our technique to not only improve precision by delaying the interprocedural propagation until we have a ACG, but also to cover a wider problem space than their solution.

6.2.2 Static stability, “leaf” class/procedure determination

Some researchers focused on how to discover portions of the Java application that could not change in the presence of dynamic class loading. They then optimized these portions based on the provable stability of the code segments.

Ghemawat et al. [41] performs a form of analysis restricted to individual class files, called *field analysis*. Their analysis derives properties about fields in a given class file using Java declarations such as public, protected, and private. They then examine every load and store to a static single assignment (SSA) form of the methods within the class file to determine such properties as never null or always class C, not a subclass of C. They determine whether or not a field is always initialized, can be inlined, can be leaked, etc. Then then use these properties to determine if a reference escapes a method. In this regard, their work is similar to our *always safe* optimization model. The key differences are that we perform our analysis and determination on a particular application’s use of the class files and do not restrict the analysis to a single class file.

Zaks et al. [22] present a form of static analysis that exploits the Java package security guarantees to “seal” a set of calls. By extrapolating these security guarantees into provable states for classes and methods within the package, they can devirtualize call sites within the package. They construct a class hierarchy graph of each package, then traverse the graph for a given method to determine whether or not it is sealed. Only methods found not to escape the package are sealed. They showed a performance gain from this form of optimization. This work is complementary to our work and part of

what we use to define “leaf” procedures in the *always safe* optimization model described in Chapter 5.

Sreedhar et al. [23] define a system that analyzes part of the application prior to execution. Their analysis allows them to determine methods that can only resolve to one possible target at runtime even if a new class is loaded, extant, and methods that could have a different target at runtime if their type is different due to class loading, nonextant. They also present a framework for how to identify where to place checks for nonextant methods so that the correct version of the code is called. This is similar to the ACG *known* and *speculative* nodes and the *always safe* optimization model. However, we make no assumptions about the application ahead of time. Therefore, we can handle a fully dynamic application.

6.2.3 Addressing dynamic applications

Several researchers have also begun to study the problems associated with optimizing a dynamic application. Whaley and Rinard [42] were among the first to address the issues of dynamic application analysis with the creation of the points-to escape graph. This graph is used to determine escaping information via a compositional points-to graph. The program is represented with CFG that has only those statements that are relevant to the analysis. This is similar to the representation we describe in Chapter 2 for the intraprocedural phase of our analysis. They do not, however, discuss how they represent exception control flow or finally blocks within their CFG representation, but state in the paper this is assumed correct. For interprocedural phase, Whaley uses a call graph to

determine caller-callee pairs. The points-to graph contains nodes for each object creation point, and there are several fine grain variations on the node types described in the paper. It then represents interactions between the nodes in the graph via edges. The algorithm allows for the insertion of new information should an unknown callee become resolved. They do not discuss how they conclude that a callee site has obtained all potential callees, handle dynamic class creation, or how they adjust information in the presence of behavior changes in the application. The algorithm computes a safe solution using a join at call sites. The points-to escape graph is similar in concept to the CDG; however, the CDG contains both finer grain detail for the object instances and flow sensitive information. Furthermore, we do not restrict nodes in the CDG to just creation nodes. In this regard, the points-to escape graph is closer to a directed form of the OCG described in Chapter 4. Whaley later expanded this work to enable a form of partial method optimization [26]. They use profiling to identify rare code block as BBs that were not executed at all during the profiling phase. They replace rare BBs with stubs that transition to interpreted code. They ignore rare paths in their escape analysis and stack allocate objects that are method local. If the object could escape along the rare path and the execution traverses to the rare path, then the objects are copied from the stack to the main heap and pointers are updated. This approach is similar to what we discussed in Chapter 5 for the *sometime safe* model of stack allocation. We, however, address not only the concerns of a single object flushing but also related objects.

Vivien and Rinard [43] present a system that uses the points-to escape graph in a dynamic optimizing runtime. They compute partial points-to escape graphs that denote

objects that escape their allocating method. They do this dynamically with resolution of method information dynamically folded in. In their approach, each method has a points-to escapes graph for the objects it uses. Information from each method is folded back into the bigger picture. For methods not yet resolved, the objects are considered escaping and the method is marked as unresolved. The algorithm computes which unresolved method could produce the best returns and analyzes that one next. They perform the analysis on a CFG which has been preprocessed to contain only the reference information, assignments, field accesses (load and store), creations, and invocations. This is similar to the CFG we use in Chapter 2. Each method is initially processed under the assumption that the parameters are unaliased. If parameters are found to alias, the nodes are merged to conservatively represent the aliases. This is similar to our use of the CDG to construct the OCG in Chapter 4. The primary difference between the OCG we use and the points-to escape graph used here is that we have removed direction from our graph, thus enabling swift, dynamic decisions on first invocation.

Several researchers have applied the points-to escape graph representation to static analysis as well. Salcianu and Rinard [44] expand points-to escape graphs to now cover multiple threads executing together. The goal is to verify region based allocation schemes where objects used within the lifetime of regions of a group of threads do not exceed the region. This is common in many forms of parallel applications. The goal of this analysis is to allocate the object instances together, thus freeing the whole block once the region completes. Therefore, they look for references that outlive a region, thus escaping the region, versus escaping a given thread. In this regard, it is similar to our use of the

CDG/ACG information to provide hints to the memory manager for allocation. They can also use the thread graph to discover objects that are shared between the threads but only synchronized by one thread or separated by thread start barriers. This allows them to eliminate synchronization on objects that intrathread algorithms could not identify. Our CDG/ACG representation could be used to perform a similar form of static analysis. Souter and Pollock [45] use a modified version of the points-to escape graph to study the def-use association between aggregate members of classes in the calling context of a class/application. An aggregate relation is when a class of type A contains fields of class B. They used the modified version of the points-to escape graph to compute and represent the def-use information. They suggest that the improved def-use information computed by their annotations to the points-to escape graph could be used to improve program testing tools. This type of information is already contained within our CDG representation. Therefore, the CDG could be employed in a similar fashion for program testing.

Several researchers have also presented the use of partial information to perform optimizations in a running application. They also suggest that some form of validation and rollback may be necessary. Pechtchanski and Sarkar [46] perform an optimistic optimization on a partial call graph as then known by the application. This is similar to our optimization models based on the ACG formulation as described in Chapter 5. They use validation and have fix-up code for when the assumptions were incorrect. They do not give detail on their validation system and speak very abstractly about registering requests and performing corrections. This makes it difficult to understand how it actually

accomplishes what they claim it does. To perform a form of validation, they use a value graph for type information. The graph is used for type inference. Since a graph is constructed in a single linear pass, a graph of each new method executed is constructed and folded into the global graph. If a type assumption used for an optimization is violated, the code is invalidated and execution returns through unoptimized code. They do not perform analysis with this representation but only use it for quick validation. The information in the CDG can be used for the same purpose. However, the CDG contains much more information and enables a richer set of analysis and validation tools. Therefore, our design solves a much larger problem space than the specific case value graph. They also do not address the form of the validation or the mechanisms for redirection to the unoptimized portion of the code.

Ishizaki et al. [47] describe a form of devirtualization of methods using a partial CHA of the application. They “fix up” code when they discover a new class that overrides a method that had been devirtualized and inlined into another method. They locate the collision by use of a result cache that contains not only the class resolution information, but also address information on which locations in the code stream need patching. They patch the code stream by inserting a branch to redirect to the new virtual call. If code motion was used around the inlined site, the redirect of the branch may include fix-up code. This is similar in concept and design to our validation registration described in Chapter 5. The primary differences between our design and their result cache is our ability to specialize validations to specific state changes by use of the CDG. Furthermore, we can specialize the rollback and recovery to the particular optimization. We also present

a simpler design than the backward patching described here, with our use of redirection stubs at the optimized code segment entry point. Although this adds overhead to the target sites, we assume the overhead is mitigated by the next run of the optimizer during the next optimization phase of the runtime. Therefore, we view this added overhead as only temporary. We further address the need to check-point and roll back should a devirtualized method get overridden. However, we do not address the additional issues involved with interface definitions and leave that expansion to future work. They also keep track of how many dynamically loaded classes implement each interface. If only one class is found to implement it at compile time for the given method, the interface invocation can be weakened to a virtual invocation with the potential of further weakening.

6.2.4 Enabling aggressive optimization through annotations

The use of annotations within class files has also been suggested by other researchers. Azevedo et al. [48] was one of the first published works to suggest the use of annotations to improve code generation in a dynamic Java application. They present a set of annotations to be derived from Java source code and annotated into the bytecode files. They store the annotations using the attribute fields. We, on the other hand, do not assume source level access and derive our information from the bytecode representation. We do, however, also exploit the attribute properties of a bytecode file to persistently maintain our intra-procedural information. They store information such as register allocation, mapping it back to the bytecode stream. They are primarily concerned with quick code generation. We, on the other hand, use annotation to store information about

the interrelations of reference fields within a method. Our goal is to enable swift runtime optimization by amortizing the costs of dynamic runtime analysis. Krintz and Calder [49] also suggested that annotation may benefit runtime code generation. Their goals were to design a general purpose annotation that could be used by a wide range of optimizations as well as reduce the size expansion of such an annotation. They reduce the size by using a single Unicode character for the annotation type name in the constant pool and by gzipping the annotations. Since all other VM will silently ignore this annotation, only theirs needs to know that it is in a gzip format. They annotate the entire class file, not each method. They annotate information for global register allocation, flow graph construction, inlining, profile based method priority (optimizing), and constant propagation. Their annotations are application specific and cannot be used for library functions as of this publication, but they plan to extend that. They are also runtime specific, while our CDG is designed to be runtime independent.

6.2.5 Validation

Validation of threaded applications has been an active area of research for several years. Recently, researchers have begun to address these concerns in Java applications. Corbett [50] presents the design of a system that is capable of detecting unsafe sharing and synchronization problems in multithreaded Java applications. They are capable in their system to even identify globals as being “owned” by one thread. However, in order to provide this level of provable results, they rely on a closed worldview of the application and assume full application knowledge. They further assume, for simplification purposes,

that all procedures have been inlined. To handle polymorphic call sites, they use test cases for type determination and target the correct inlined code. We briefly mention in Chapters 3 and 5 how our research can also be expanded to perform this form of analysis. Naumovich et al. [51] also present a model for checking concurrent Java applications. His work deals primarily with checking properties in concurrent Java programs by using a representation called a Trace Flow Graph (TFG). The TFG represents programs as a form of a CFG which contains only events they wish to reason about. They then perform the analysis on the TFG and prove properties about the concurrent Java application. Again, this work assumes a closed-world, full-knowledge view of the application.

6.2.6 Java runtime designs

Several researchers have discussed the design of Java runtimes and although we do not restrict our work to a particular runtime design, we do discuss modifications that may be necessary to utilize it. Therefore, the design of the Java runtimes is also related to our work. Suganuma et al. [52] describe a Java runtime that uses a Mixed Mode Interpreter (MMI) intermixing the execution of interpreted code and compiled code. The MMI runtime shares the execution stack and exception handling mechanism between the two types of execution. The runtime uses profiling based not only on method counts but also on loop counts to discover which methods or sections of methods are candidates for optimization. They employ three levels of optimization in their dynamic runtime compiler to offset the cost-benefits of dynamic compilation. They do not address how their optimized code deals with dynamic class loading. They only present optimization

on a method boundary level with limited inlining, so it is possible that the dynamic class loading issues are never a concern. Our system of analysis and validation would complement their design by potentially efficiently enabling more aggressive optimizations in all three optimization level of their runtime compiler.

Burke et al. [53] presents an overview of the IBM Jalapeno Java VM written entirely in Java. Jalapeno uses a compile only approach to bytecode execution. All bytecode is compiled into machine code prior to execution. It uses back patching to support dynamic loading of classes after compilation. The profiling system uses a context sensitive call graph for maintaining information. The collected profile triggers the optimizing compiler when certain thresholds are met. Their analysis and optimizations rely on several verifiable bytecode guarantees such as the stack height and type, and every variable must be defined before use. We make the same assumptions in our analysis framework.

Kazi et al. [28] describe several types of systems for Java execution. They describe details of very early Java execution systems including very early generations of the JIT compilation architecture. They do give brief descriptions of Sun's Hotspot and IBM's Jalapeno as the examples of dynamically optimizing environments. They also describe some hardware implementations of the VM, although most of these projects are no longer active. They report on other Java analysis and optimization tools such as automatic parallelization tools similar in structure to the Fortran automatic parallelization tools. They cover some of the work in improving specific Java features such as synchronization and Remote Method Invocation (RMI) calls. They talk about Java's shortcoming for numeric computations and briefly discuss GC implementation. They then evaluate the

benchmarks available for Java research. Although not directly related to our work, this paper gives a good overview of the then available Java related tools.

6.3 Other Forms of Object Oriented Optimizations

Besides the forms of analysis and optimizations already addressed in this section, there are other forms of object oriented optimizations. Some of these, such as object inlining, do not have a clear means of implementation in the presence of dynamic class loading. Object inlining, inlines the fields of one object that are themselves objects into a parent object. It involves substituting the instances of the child object within the application with the appropriate field accesses to the corresponding new parent object. Dolby and Chien [13] describe a technique for performing object inlining. This paper formalizes the definition of object inlining and gives examples of its use. They define the notion of dynamic one-to-one as the determination that for a given execution, a child object field has one and only one mapping for a given parent object. They then define the transformation as first locating within an execution trace all one-to-one mappings, then creating the new parents and locating all child accesses that must now be transformed. Note that in the presence of dynamic class loading, new child classes can be loaded at any time. Furthermore, new users of the created inlined object could appear at any point, making it difficult to track and update the code space.

Although Java has single inheritance for class files, it allows multiple interfaces to be implemented. This is in essence a form of multiple inheritance that can complicate

Java optimizations. Alpern et al. [27] describe a method of overcoming the three main obstacles they see as inhibiting performance with interface calls. They are dynamic type checking, method dispatch, and inhibited compiler optimizations. They explain that – unlike the single class inheritance structure in Java class files that allows a VM to exploit the virtual method table offsets to redirect subclass methods by using the same superclass offset – interfaces could be implemented by unrelated classes and thus have no consistent offset. They present a coloring scheme for interface resolution that has conflict stubs for coloring conflicts. They also discuss how type information can be used to virtualize interfaces calls and even devirtualize them with the potential of inlining. Although our work can be extended to incorporate interface calls, we do not address the particulars in this thesis. Primarily, inclusion of interfaces in our design affects our runtime structures. The representation of the interface hierarchy, the interface inheritance for class files, and the ACG, all require adjustment to efficiently accommodate interfaces. However, the design of the CDG remains unchanged.

Other forms of analysis and optimization are specific to Java. Aggarwal and Randall [54] define a form of analysis called related field analysis. They say that related field analysis can be viewed as proving an invariant about related fields. They give an example of a loop that steps through the elements of an array. They analyze fields of the same class where field A is an array type and field B is an integer type. They then look to prove a relationship between the two fields. They then use this analysis to eliminate array bounds checks. The field relation information in the CDG can be used to facilitate this form of analysis.

Choi et al. [55] present a form of CFG construction for Java applications that groups exceptions that may be thrown in a BB into one factor edge connected to the appropriate handler. In a normal CFG, control is broken at the potentially exception causing instructions, and they are connected to an appropriated intramethod handler or the exit block for the CFG. They state that this modification could impact analysis techniques, especially backward propagation techniques. It does not complicate local forward analysis but actually enables more optimizations because the BBs are larger in an FCFG than in a traditional CFG. Because we used backward propagation for our intraprocedural analysis described in Chapter 2, we did not employ this optimization to our CFG construction. However, our intraprocedural analysis can be performed in a forward traversal. By transforming the intraprocedural analysis to a forward propagation algorithm, we may be able to take advantage of this optimization.

Fitzgerald et al. [56] use the detection of the construction of more than one thread of execution as a means of detecting a multithreaded application. If they discover only one thread, then they conclude that the application is single threaded and synchronization is unnecessary and can be eliminated. They also determine an early region in an application prior to the construction of the second thread when the application is in single threaded mode. They then can eliminate synchronization from these regions as well. Their work was performed on an earlier version of the Java runtime with earlier libraries. They did not address the newer library code in which helper threads are spawned very early in the application's lifetime. It is not clear how their work could be extrapolated to handle these cases.

Liang et al. [19] present an exploration and evaluation of constructing points-to graphs using adaptations to Steensgaard's and Andersen's algorithms. They explore two methods of handling the "this" parameter; one method treats "this" as a formal parameter and the other uses a simplification of field accesses which maps instances back to the class to simplify the handling of "this." They use a constructed abstract syntax tree representation of the applications to evaluate the effectiveness of the algorithms. They evaluate the algorithms on three different approaches for virtual call resolution, class hierarchy analysis, rapid type analysis, and a method that starts from main and discovers the virtual targets that are correct at each call site. They call this method FLY. They also rely on a user supplied model for handling container and table classes such as the Vector and Hash tables in the Java libraries. They do not address how they handle dynamic class resolution where the closure cannot be ascertained. They conclude from their study that the best results come from applying Anderson's algorithm with the field class mapping and either rapid type analysis or their on-the-fly analysis. Our construction of the class hierarchy as the classes are loaded, and our on-demand ACG construction, make our model similar to applying their FLY method. Therefore, their results suggest that our dynamic resolution may also be optimal.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this dissertation we have described the design of a framework for performing analysis in dynamically linked and loaded applications. This framework divided the problem space into intraprocedural and interprocedural analysis. For the intraprocedural analysis, we described our representation for the interactions among reference values which we called a Compact Dataflow Graph. For the interprocedural analysis, we presented our design of an Adaptive Call Graph which differs from traditional call graphs in two main ways. First, it is formed for the procedure being considered for optimization and not necessarily the whole application. Second, it distinguishes between call sites that are *known* and those that remain *speculative* within the graph. The ACG is then part of our described framework for efficient and effective interprocedural analysis, which iteratively forms the ACG in conjunction with the iCDG. We then presented an implementation of our framework to solve the first invocation stack allocation analysis problem. We demonstrated how information could be easily extracted from the CDG and formed an intermediate structure that represented the points-to information within the graphs. We then simulated a Java runtime that we constructed and evaluated the effectiveness of our design by using traces collected from a production level runtime.

We expanded our framework beyond its application to the dynamic analysis domain into validation and recovery. We started by defining three optimization models for dynamic application optimizations: always safe, sometimes safe, and speculative. Then for each of these three models, we described how our framework helped implement each model. Furthermore, we described the validation necessary to enable aggressive optimizations, and the types of rollback and recovery needed in the event of validation failure.

Although in this dissertation we described our design of a new framework for interprocedural analysis, which presents a formable step in the construction of a dynamic application analysis system, there is still a great deal of research left to be accomplished within this domain.

7.1 Context-Specific Analysis Result Retention

In the implementation of our framework that we described in Chapter 4, we made no attempt to maintain and reuse intermediate or even previous interprocedural analysis results. In turn, this means we also do not have the ability to retain and reuse the optimization results.

The design of a reuse system consists of several layers, including storage, and locating the correct optimized method versions for each context. The storage of multiple results for a given method can be accomplished via a modification to the structure pointed to by the method pointer. We show an abstract version of this modification in Figure 7.1. In this modified method representation, an invocation of a method now takes an additional

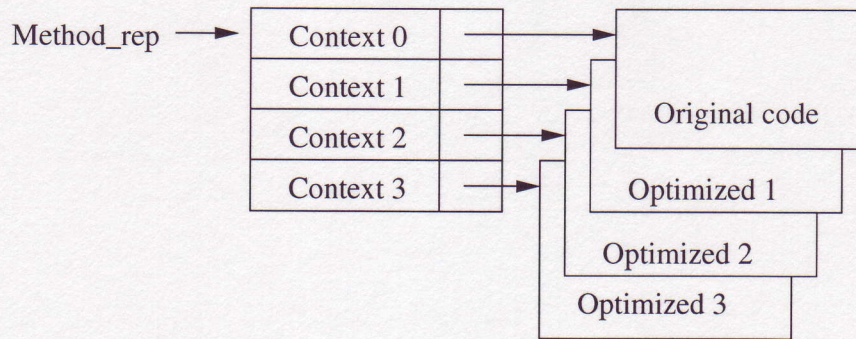


Figure 7.1 Theoretical modified method pointer to handle multiple context versions.

step besides the standard virtual table lookup. When an invocation for a given method is made dynamically, it is made with a certain context. This context is then matched with the available contexts in the method representation. If a match is found, then that version of the code is used with the default being the original code or `Context 0` in Figure 7.1.

The matching of a context is not as simple as Figure 7.1 implies. We made the decision to discard intermediate analysis results not because of the new method representation design but rather because of the difficulty in recognizing what was meant by a context. The context for which a given analysis result is valid can be rather complex. For example, not only is the exact type of the object instances being passed interprocedurally important, but also how the fields within these object instance connect. The matching of exact object instance types requires accessing the memory representation of the object instance and recovering its exact class type. Then this set or types require matching with the stored copy or potential copies of the intermediate results. One technique we

proposed for efficiently implementing such a system may be a class numbering scheme similar to the one used by Blanchett [15]. However, we cannot implement a height system as simple as his since the class hierarchy is never assumed closed and the height values may require realignment with each newly loaded class file. Instead, a system that sequentially assigns a class id number to each newly loaded class file and then stores that number within the runtime's representation for that class may be a viable solution. Then when a new object instance of a given class is allocated, its memory representation contains the assigned number for that class file. With this design, when the types of the context require matching, the numeric values can be retrieved from the memory representations and matched with the stored context. This system would allow for even don't care positions within the context identifiers shown in Figure 7.1.

However, there is more to a context than just the exact types of the object instance. The analysis results often depend on the connections between the object instances as well as their escaping state upon entry. This level of matching is not a problem for our framework. Note that in Chapter 4, we showed how the bit vector was used to pass connection and state information across the interprocedural boundaries. This same bit vector approach could be used for the matching of the state information. The bit vector associated with each pertinent type would also be part of the context identifiers shown in Figure 7.1.

As future work, we hope to implement within the simulator several techniques for matching context. We hope to be able to use the context matching to define an efficient technique and be able to retain and reuse previous results.

7.2 Quantifying the Validation Framework

Our validation framework described in Chapter 5 has not as of yet been implemented. We foresee this framework as being potentially instrumental in the implementation of a modified version of the Hot Spot detector developed by Merten et al. [57, 58]. With our framework, it may be possible to retain optimized hot spots even in the presence of newly loaded class files. It also may allow us to recognize when a currently active hot spot is no longer valid due to the newly loaded class files. We hope to use the validation framework in this way and measure its effectiveness in identifying potentially disqualified hot spots. We hope to measure this in two ways: first, how effective our system is in identifying which class file loads truly caused a hot spot disqualification and, second, how much longer hot spots lived when the fine grain validation mechanisms are employed versus straight class loading disqualification.

7.3 Accomidating Additional Java Features

Although we designed the analysis framework for use within the Java language, we have not fully specified the design for additional language features. This incorporation should be fully explored before full incorporation of these techniques within a production Java runtime. Among the Java language features yet to be fully specified are the use and inheritance of the `interface` definitions, and the use of the iCDG for interprocedural exception tracking.

7.3.1 Interfaces

Interfaces, in Java, are an abstract form of a class file. They contain fields and method names but no implementation for the methods. Class files that implement the interfaces are then required to provide an implementation for the methods specified in the interface definition. Class files are only limited in the number of interfaces they can implement by the size of the interface count field in the bytecode file specifications [29]. Therefore, interfaces provide a way of implementing a form of multiple inheritance. On a programming level, the programmer can call interface methods directly, thereby allowing for the inclusion of new class files at any time in the application.

In order to accommodate interfaces, the class hierarchy representation within the runtime needs to be able to handle multiple inheritance. Furthermore, there may now be an extended group of object instance types that can resolve to a node on the ACG. This can increase the number of speculative nodes in the ACG. Additionally, the validation requests now need to specify an interface name. This can complicate the validation process. All of these areas need further exploration.

7.3.2 Interprocedural exception tracking

In Java, the occurrence of an exception can redirect execution through several layers of the call stack to a handler. This form of redirection has been used by some programmers to implement a form of the C programming language style `setjmp/longjmp`. The most notable use of exceptions in this manner is the SpecJVM98 benchmark, `_228-jack` [59].

However, Java exceptions incur overhead that is not necessary when used in this fashion – for example, the building of a stack trace that is discarded when the exception is caught.

The CDG representation does contain connections for targets of potential interprocedural exceptions. However, we have not fully explored the use of the CDG to solve this problem space. Therefore, we still need to determine if the inclusion of exit points for all potential exception points when forming the CFG for the formation of the CDG described in Appendix A, is necessary for the analysis of interprocedural exceptions. It is not clear at this point whether or not this step is necessary, and further investigation is still needed.

7.4 Additional Validation and Verification Uses

We believe that the validation system we described in Chapter 5 can be applied to additional validation and verification domains. For example, the information contained within the CDGs gives an accurate representation of the intraprocedural object instance usage and state. This information could also be used to solve harder validation problems such as data race detection.

7.4.1 Race detection

The iCDG/ACG structures can be viewed as a first step in the design of a race detection system. There are two ways of detecting races in an application: dynamically or statically. In static race detection, a graph of the running threads with thread control

boundaries is constructed. From this construction, properties are deduced about the given threads. This requires full knowledge of the class files used by the application. However, the applications we are studying are dynamic. Therefore, it may not be possible to build a static view of the application. Furthermore, since the class file input set to the application could change at any time, the detection mechanism needs to be adaptive as well.

The dynamic solution depends on the goal of the race detection system. For example, in order to provide precision, the system may require far more overhead than is tolerable in a running application. However, an alternative goal is to use race detection as a means of providing stability to the system. For example, if the runtime can detect with a reasonable amount of overhead that the application has a potential data race, then the optimizer may choose not to optimize sacrificing performance for stability. Therefore, a safe solution may be sufficient. To accomplish this, an intermediate structure similar to the OCG we introduced in Chapter 4 may be needed. This use of the iCDG/ACG is currently under investigation and therefore part of future work.

7.5 Extrapolation into Other Languages

Although our work was based on the Java language, we do not envision its use as limited to just this language. Therefore, we view Java as a smaller subset of languages such as C++. In extrapolating this research into other languages, we foresee several

challenges but view them as adjusting the boundary conditions used in our problem description.

7.5.1 Eliminated boundary conditions

One restriction within Java that could be considered a boundary condition is the single inheritance structure on the class hierarchy. The class hierarchy in C++ is more complex with multiple parents. However, since C++ is statically linked and compiled the solution is known. Therefore, what we discover dynamically can be provided by the annotations in the C++ executables.

Another boundary condition within Java that is absent in C++ is the use of a memory manager. This boundary in Java makes it virtually impossible for even native code to step into the space of another object instance and change it. This imposition is basically because the representation of the object instances in memory as well as their locations and relative locations to each other, is under the control of the memory manager. This means that a code segment using a pointer to memory cannot make any assumptions about what will be located at a given offset from it. In C++, often pointers are used to step through memory locations, making the linking and alias tracking problem more difficult. It may not be possible to sufficiently represent the links between object instance since the relation may be dependent on unknown offsets. However, by employing some of the techniques developed in alias analysis of C and C++ programs, a conservative, safe representation may be achievable.

7.6 Exploiting Hardware Specific Features

We conducted some early work on the evaluation of Java's early runtime implementations of certain hardware features [60]. At the time, we found that early Java interpreters performed poorly on current hardware features such as Branch Target Buffers (BTBs). Recently, some researchers have found that under the newer optimization models for Java runtimes, this is not the case. They have suggested that certain forms of optimization such as our *sometimes safe* model even benefit, not suffering the full cost of the added check. This is because the branch is almost always one path with the rare case being the other, thus making it easily predictable by most branch prediction hardware. One area of exploration for this research is the use of hardware features such as the BTB or predicate registers to enable forms of validation. We plan to evaluate how the forms of optimization proposed in our Dynamic Application Analysis Framework map to these hardware features, as well as how each model's validation, rollback, and recovery can be enhanced by certain forms of underlying hardware structures.

APPENDIX A

BYTECODE LEVEL ALGORITHM AND ANNOTATIONS

In Chapter 2 we presented a source-level abstraction to describe the formation of the CDG. However, one primary assumption of our system is that we will only have access to a bytecode representation of the class file and not a source level version. Under this assumption, the actual analysis used is designed to operate on a bytecode level representation of the class files. Furthermore, the final CDG is then written back out using the annotation facilities provided by the Java specifications [29].

In this chapter, we describe the actual analysis for forming the CDG using only the bytecode representation of the file. We present an example of this using the same example method used in Chapter 2 for describing the source-level abstract version. We then compare the resulting graphs, the abstract one in Chapter 2 and the CDG formed from the the bytecode representation, and show that the two are equivalent. Finally, we describe the format of the annotations added to the bytecode files.

A.1 Intraprocedural Algorithm for Forming CDG

The intraprocedural algorithm is performed in the following steps:

1. Form a traditional *Control Flow Graph* (CFG).
2. Adjust the CFG by breaking the *Basic Blocks* (BBs) and adding additional arcs for Java specific language features.
3. Perform a linear pass over the graph to determine which temporary locations are used by each instruction.
4. Remove any instruction not operating on a reference value.
5. Perform a flow-sensitive, iterative, backwards, dataflow propagation algorithm forming an intermediate graph.
6. Reduce the results to a CDG.

The CFG is formed utilizing traditional methods for CFG construction as given in [61]. It uses control flow to determine the *Basic Block* (BB) boundaries and then connects the appropriate BBs. The main differences between a traditional CFG and the one used here is we also represent internal exception flow and subroutine arcs which are part of the Java language.

A.1.1 Breaking BBs for exceptions and subroutines

In the event of an exception, the exception handling mechanism within Java redirects control to the nearest enclosing handler that is capable of handling the particular

Table A.1 Exception scoping example.

	startPC	endPC	handlerPC	type
0	0	20	23	myExceptionType
1	0	23	33	java.lang.Throwable

(a) Before parent exception

	startPC	endPC	handlerPC	type
0	0	20	23	java.lang.Throwable
1	0	23	33	myExceptionType

(b) After parent exception

type of exception. For example, in Table A.1(a), the range covered by the handler for `myExceptionType` is from the bytecode instruction occurring at location 0 to the bytecode instruction ending before location 20. The handler for this exception starts with the bytecode instruction occurring at line 23. A table similar to the one shown in Table A.1 is included in the bytecode file for each method containing a handler. When an exception occurs, execution continues from the handler code forward, and does not return to the point at which the exception occurred. Handlers can be contained within the method (i.e., intraprocedural handlers), or in a caller method (i.e., interprocedural handlers), with the degenerate case of no handler. In the intraprocedural analysis phase we are concerned with the intraprocedural exceptions and connect the appropriate arcs for both explicit and implicit exceptions.

The explicitly thrown interprocedural exceptions are also represented in our refined CFG. For handler ranges that enclose an invocation, we make a conservative assumption that any callee covered by its range can return to it and form the appropriate connections. Although implicit exceptions not handled intraprocedurally can be viewed as method exit points, we do not represent these exits in our refined CFG.

In the event of an implicit exception, the object creation for the exception object is handled by the virtual machine and not represented within the bytecode for the method; therefore, we rely on the interprocedural analysis and profiling of the running application to detect these exceptions.

The exception arcs added to the CFG are added in two phases. First, using the handler ranges specified in the exception table for the bytecode file for the method (Table A.1), the BBs are broken such that the range handled by a particular handler is completely enclosed within a subset of BBs while maintaining the original control flow. These BBs are then connected via a special exception arc to the specified handler. Note that in locating the correct handler for a given exception type, the inheritance structure of the exceptions is also considered.

The second phase breaks the BBs at the potential exception point, maintaining the original control flow and adding a new arc to the appropriate handler. There are two types of intraprocedural exceptions we represent within the CFG, implicit exceptions and explicit exceptions. Implicit exceptions are exceptions that can be thrown by a subset of the Java bytecode instructions. Table A.2 shows each set of bytecode instructions that can cause an implicit exception or error to be thrown, and which errors or exceptions they

Table A.2 Potential implicit runtime exceptions and errors thrown by specific bytecode instructions.

#	Bytecode	Runtime Exceptions	Runtime Errors
1	invokestatic	User Defined	Initialization errors UnsatisfiedLinkError
2	invokespecial	User Defined NullPointerException	UnsatisfiedLinkError
3	invokevirtual	User Defined NullPointerException	AbstractMethodError UnsatisfiedLinkError
4	invokeinterface	User Defined NullPointerException	IncompatibleClassChangeError IllegalAccessError AbstractMethodError UnsatisfiedLinkError
5	areturn	IllegalMonitorStateException	(none)
6	getstatic, putstatic	(none)	Initialization errors
7	checkcast	ClassCastException	(none)
8	getfield,	NullPointerException	(none)
9	instanceof	(none)	(none)
10	aaload, baload, caload, daload, faload, iaload, laload, saload, bastore, castore, dastore, fastore, iastore, lastore, sastore	NullPointerException ArrayIndexOutOfBoundsException	(none)
11	arraylength	NegativeArraySizeException	(none)
12	aastore	NullPointerException ArrayIndexOutOfBoundsException ArrayStoreException	(none)
13	new	(none)	Initialization errors
14	newarray, multianewarray, anewarray	NegativeArraySizeException	(none)
15	monitorenter	NullPointerException	(none)
16	monitorexit	NullPointerException IllegalMonitorStateException	(none)
17	athrow	User Defined NullPointerException IllegalMonitorStateException	(none)

can throw. We classify these instructions into 17 sets based on the types of exceptions and errors each set throws. Due to the polymorphic properties of the Java language, it is possible for a parent class of these exception types to also catch and handle the thrown exception. Figure A.1 shows a graphical view of the exception and error class hierarchy for the implicit exceptions and errors shown in Table A.2. This hierarchy is part of the standard Java library files and the hierarchy is defined in [29]. Under each exception or error shown in Figure A.1 we have specified which set in from Table A.2 is capable of reaching a handler of the given class type. Handlers of a parent type are valid handlers for the exceptions and errors in Table A.2. The type of exception handled by the handler connected by the exception arcs in the first pass, is used to define the set of bytecode instructions that can reach it via throwing the appropriate type of implicit exceptions. For example, if a BB is connected via an exception arc to a handler of type `ClassCastException`, then the BB is broken at any `checkcast` instruction contained within it and connected from that point directly to the handler.

The reason for this breaking and direct connection is that although the operand stack has been emptied of all values except the exception object, the local variable locations are still valid. It is possible for the handler code to affect the state of one or more of the local variable values. This information needs to be collected and transferred up the reverse path from the handler. The reason for connecting the exact instruction that can throw the exception directly to the handler instead of the end of the BB, is that the local variables could change from the point where the exception is feasible and the actual end of the BB. Therefore, in order to guarantee correctness, the exception point is connected directly

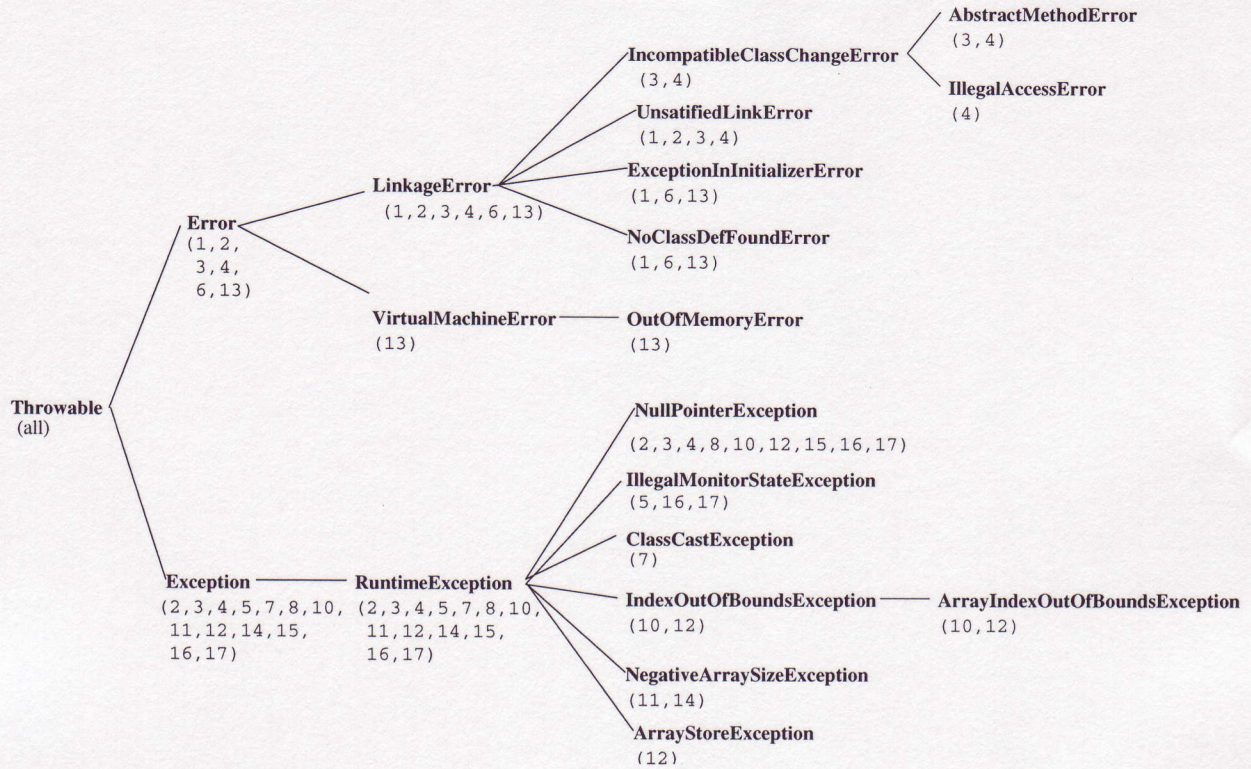


Figure A.1 A graphical hierarchy of the implicit bytecode exceptions.

to the handler by breaking the BB at any bytecode instruction that could reach the handler's exception type. The interprocedural exceptions are handled here by breaking any BB containing a handler arc at any invocation bytecode instruction it contains and adding an additional arc connecting it directly to the handler. These instructions are the first four instructions listed in Table A.2. This connection is not precise but is safe. It is possible to connect an invocation to a handler it can never reach. However, although we include extraneous arcs, this can only increase the potential reachable uses for a given definition and not reduce them since the original path is still included in the set. We are able during the intraprocedural analysis propagation phase to cull out some of the

extraneous information added by these arcs. This will become clearer when we discuss the intraprocedural analysis propagation.

The other type of exceptions are explicit exceptions thrown by using the key word `throw`. They can either be library exceptions like the ones shown in the implicit set in Table A.2, or user defined exceptions that can subclass from any point in the hierarchy of library exceptions as long as `throwable` is the root of the hierarchy [29]. Therefore, explicit exceptions always break a BB and are conservatively connected to all enclosing handlers for the BB stopping the scoping of the connections only if the handler type is an exact match for the exception thrown. This conservative approach can add additional control flow arcs to the graph but will not remove any arcs. Therefore, it can only add superfluous information which is safe though not precise.

Subroutines in Java bytecode files are regions of code that the regular execution stream explicitly redirects execution to. At the end of the subroutine, execution returns to the next instruction in the regular stream. Subroutine regions can either be explicitly added by a programmer or implicitly added by the Java source to Java bytecode compiler. They are used primarily for freeing resources before continuing execution.

Subroutines can be called from multiple points within the Java method. Therefore for the analysis, we use simple duplication of the subroutine blocks to avoid pollution from connections to multiple streams that would occur if all streams were connected to one instance of the subroutine blocks.

This completes the first two steps in the analysis process. The next step identifies the operand stack and/or local variable locations used by a given bytecode instruction.

A.1.2 Reducing analysis bytecode and assigning arguments

To assign the temporary locations used by each instruction we assume that the bytecode is verifiable. This assumption can be lifted by simply performing the steps outlined in [29] for bytecode verification. Verified bytecode guarantees that the operand stack height will be the same upon entry to each BB no matter which path is used to reach the BB. Furthermore, it guarantees that if a primitive value is written to a temporary location, only a primitive value of the same type can be read from that location, and likewise for reference values. Therefore, we can use a linear pass starting at the root BB and traversing each BB only once to determine which temporary locations are used by each instruction.

To reduce the number of instructions analyzed by the algorithm, we remove any instruction that strictly uses primitive values. This is safe since these instructions cannot operate on reference values or affect the contents of any temporary location containing a reference value. Table A.3 gives the bytecode instructions that remain in the CFG after this step. They are subcategorized into eighteen subcategories, based on the type of instruction. The instructions account for 57 of the 202 assigned bytecodes, with both `aload_<n>` and `astore_<n>` representing four distinct instructions each. The percentage of these 57 instructions in the unique methods executed within the threaded programs we used is given in Table A.4. The percentage comes into play if portions of the analysis are performed at load time. For example, if the application dynamically

Table A.3 Reference affect events and their bytecode instructions.

category	bytecode instructions
<i>pop</i>	pop, pop2
<i>constant</i>	ldc, ldc_w, ldc2_w, aconst_null
<i>swap</i>	swap
<i>stores</i>	astore, astore_< <i>n</i> >
<i>duplication</i>	dup, dup_x1, dup_x2, dup2, dup2_x1, dup2_x2
<i>loads</i>	aload, aload_< <i>n</i> >
<i>exception</i>	athrow
<i>lock</i>	monitorenter
<i>unlock</i>	monitorexit
<i>allocation</i>	new, newarray, anewarray, multianewarray
<i>array store</i>	aastore, bastore, castore, dastore, fastore, iastore, lastore, sstore
<i>array load</i>	aaload, baload, caload, daload, faload, iaload, laload, saload, arraylength
<i>field write</i>	putfield
<i>field read</i>	checkcast, getfield, instanceof
<i>static read</i>	getstatic
<i>static write</i>	putstatic
<i>return</i>	areturn
<i>invoke</i>	invokestatic, invokespecial, invokevirtual, invokeinterface

loads an unannotated class file, the runtime may decide to create CDG dynamically for each method used in the unannotated file.

The bytecodes listed in the eighteen subcategories shown in Table A.3, are transformed into a structure called a *Reference Affecting Event* (RAE), denoting it as an event that can affect a reference value within the method. The RAE contains information on which of the eighteen types it belongs to, the temporary locations it uses, and some additional refining properties such as the bytecode line number it occurred at, a field specifier and/or parameter locations.

Table A.4 Percentage of dynamic instructions containing the bytecode instructions in Table A.3.

Benchmark	Program	Library
JGFCrypt	51.2%	64.0%
JGFLUFact	51.2%	64.0%
JGFSOR	53.8%	63.9%
JGFSeries	60.4%	63.9%
JGFSparseMatmult	69.2%	63.9%
JGFMoldyn	68.1%	63.9%
JGFRayTracer	67.3%	63.9%
Heat	62.9%	61.1%
Fib	70.7%	63.7%
MSort	63.4%	60.9%
NQueens	67.0%	61.0%
BarrierJacobi	54.4%	63.1%
LU	68.6%	63.6%
MatrixMultiply	65.2%	63.7%
Total	63.6%	63.1%

The handling of implicit exceptions by the analysis is further refined here. Implicit exceptions create a new object instance of the exception type, empty the operand stack, and then “throw” the exception for the new type. When we perform RAE transformation on the bytecode, we also add an *allocation* RAE for any implicit exception arc that may reach a handler in the CFG. This RAE is added to the top of the handler block. For explicit exceptions, the information accumulated during the analysis of the handler is merged into the exception allocated location. For implicit exceptions, this object instance became live at this point and the information for the object instance is saved. This will become clearer in the next section.

This then completes the third and fourth steps of the analysis. To this point, all passes have been linear. It is the propagation step forming the intermediate graph that is nonlinear.

Once transformed into an RAE representation, the intraprocedural analysis then forms an intermediate graph by performing a reverse dataflow propagation algorithm.

A.1.3 Forming the intermediate graph

The static analysis operates on the RAE transformed CFG representation to form an intermediate graph of the unique references used within the method. The graph formed is defined in Table A.5. Note that this graph only assigns directed field accessing edges and undirected use/define edges called property edges. The directed dataflow edges are added in the next pass by transforming this intermediate graph. The direction for the dataflow edges is defined by the property type. Note that property nodes have a different meaning in this analysis than what was defined before. Property nodes actually hold all define and use information for a given node. Therefore, only the connections between fields needs to be represented as directed. The *use* and *define* edges can be added once the temporaries have been removed from the graph. In this way, all *use* and *define* information for each unique reference will have been merged into one location for the reference instead of multiple temporary locations. This becomes clearer when the next pass is described. The graph, iG , contains two types of nodes, n which are data nodes, and P , which are property nodes. The property nodes contain all the use and define information for a given data node. The data nodes can be viewed as temporary

Table A.5 Definition of the intraprocedural analysis graph.

Symbol	Definition
iG	$::= (n^*, e^*, P^*)$
e	$::=$ An edge connecting 2 nodes. Three types of edges $\{\rightarrow, \leftarrow\}$: directed strong edge $\{--\rightarrow, \leftarrow--\}$: directed weak edge $\{—\}$: undirected weak edge
n_i	$::=$ data nodes, $\{s_i \nu_i r_i\}$
P	$::=$ property nodes, $\{(L, l) (U, l) (R, l) (W, l) (Gw, l) (Gr, l) (T, l) (A, l) (F_j) (P_j, l) (c, l)\}$ where l is the line number.

n_i type	Definition
s_i	$::=$ an operand stack location: $i = 0, \dots, (max_stack - 1)$
ν_i	$::=$ a local variable location: $i = 0, \dots, (max_local - 1)$
r_i	$::=$ reference value known used in the method but not residing in s or ν

P type	Definition
(L, l)	$::=$ a lock operation occurring at line l
(U, l)	$::=$ an unlock operation occurring at line l
(R, l)	$::=$ a read operation occurring at line l
(W, l)	$::=$ a write operation occurring at line l
(Gw, l)	$::=$ a write access of a global variable occurring at line l
(Gr, l)	$::=$ a read access of a global variable occurring at line l
(T, l)	$::=$ a thrown exception occurring at line l
(A, l)	$::=$ an allocation occurring at line l
(F_j)	$::=$ the j th formal parameter to the method
(P_j, l)	$::=$ the j th parameter to the method called at line l
(c, l)	$::=$ a constant value loaded at line l

place holders for the use and define information concerning a given reference value. For example, at the end of this propagation phase, a node, ν_0 should have a property node (F_0) attached to it. This is because, referring to Table A.5, F_0 refers to the 0th formal parameter to the method and the virtual machine specification state that this is passed in local variable location zero [29]. Note that the property node, F_0 , would be the *define* for the temporary location ν_0 and the transformation in the next pass will replace the node ν_0 with a new node corresponding to (F_0). Any uses of ν_0 will be connected via a directed outgoing edge to the new node (F_0). This is explained further when we describe this transformation in the next section.

The data nodes can be connected to other data nodes via a directed field accessing edge, e . They can also be connected to a property node via an undirected property edge, p . The data nodes can be of three types, either an operand stack location, s , a local variable location, ν , or none of the above, r . The formal parameters passed to a procedure are passed in local variable locations as defined in [29]. The procedure may use additional local variable locations up to the maximum number specified in the method's definition in the bytecode file; therefore, the intermediate graph is initialized with a node location for each of the ν nodes. The operand stack nodes are actually temporary holders. They correspond to the locations on the operand stack being used during the execution of the procedure. The operand stack is guaranteed not to exceed the value specified in the bytecode representation of the method for the maximum stack height. Therefore, the initialization of the intermediate graph also contains a node for each of the specified stack locations. These define the initial *working set* of nodes for the graph. Note that

by assuming verifiable bytecode, we are guaranteed that on entry to a procedure, the operand stack is always empty. Since the algorithm performs a reverse propagation from the exit points of the procedure up to the entry point, at the end of this pass, none of the stack nodes should have edges attached to them.

A reference value defined within the procedure will have resided at one point in either a node of type ν or type s . It will not be present in the final set of these two types of nodes. Therefore, we define the node set r , which holds the reference values defined within a given method and the corresponding edges. These new nodes, r , are added to the graph as needed during the analysis. They are not present in the initial graph. When a reference value residing during the analysis in a temporary location defined as the node set ν and s , is known to no longer reside there, a new node r is added to the graph and all edges that were attached to the temporary node are now moved to the new node, r . We therefore define “live” nodes in the graph as nodes that have edges attached to them. Nodes without edges are simply part of the working set of nodes in the intermediate graph. The final graph is defined as the subgraph containing only “live” nodes, obtained at the entry point of the procedure. Therefore, upon completion of the reverse traversal, only the ν nodes corresponding to formals of the method and the r nodes should remain in the set of live nodes in the final graph. If any other ν or s nodes remain in the graph, they are extraneous data added by the conservative edges in the CFG introduced when exception arcs were added, and can be removed. This is correct because we have assumed the bytecode is verifiable and therefore guaranteed that the only “live” locations in the node set ν, s , are the formals to the method.

The reverse propagation algorithm can perform one of three potential actions on the number of nodes within an intermediate graph, increase the number, decrease the number, or allow the number of nodes to remain constant. To perform these three actions, the algorithm uses the functions $add(n_i)$, $remove(n_i)$, and $kill(n_i)/transfer(n_i, n_j)$. The $add(n_i)$ function adds a new node, n_i , and its corresponding edges to the graph. The $remove(n_i)$ function performs the inverse, removing a node, n_i , and its corresponding edges from the graph. Conversely, the $kill(n_i)$ function differs from the $remove(n_i)$ function in that it only breaks the edges corresponding to a node within the graph; it does not remove the node n_i from the graph. Note that the breaking of edges has the effect of removing the node from the set of “live” nodes. It does, however, remain in the working set. Therefore the $kill(n_i)$ function does not change the number of nodes in the intermediate graph. The $transfer(n_i, n_j)$ function transfers the information from one node, n_i , to another node, n_j , thus neither increasing nor decreasing the number of nodes in the graph. We can then define the $transfer(n_i, n_j)$ function as being composed of two logical functions, $copy(n_i, n_j)$ and $kill(n_i)$. All three of these, $copy(n_i, n_j)$, $kill(n_i)$, and $transfer(n_i, n_j)$, are defined in Figure A.2. The $transfer(n_i, n_j)$ first copies the edge information from node n_i to node n_j , then kills the edges attached to n_i . With the four basic functions, $add(n_i)$, $remove(n_i)$, $kill(n_i)$, and $transfer(n_i, n_j)$, we can describe the effects on the intermediate graph, of processing each RAE encountered during the reverse traversal.

$$\begin{array}{l}
\text{copy}(n_i, n_j)\{ \\
\quad \forall k\{ \\
\quad \quad \exists\{(n_i) \dashrightarrow (n_k)\} \Rightarrow (n_j) \dashrightarrow (n_k); \\
\quad \quad \exists\{(n_i) \dashleftarrow (n_k)\} \Rightarrow (n_j) \dashleftarrow (n_k); \\
\quad \quad \exists\{(n_i) \cdots (P)\} \Rightarrow (n_j) \cdots (P); \\
\quad \} \\
\} \\
\\
\text{kill}(n_i)\{ \\
\quad \forall k\{ \\
\quad \quad \exists\{(n_i) \dashrightarrow (n_k)\} \Rightarrow (n_i) - \setminus \dashrightarrow (n_k); \\
\quad \quad \exists\{(n_i) \dashleftarrow (n_k)\} \Rightarrow (n_i) \dashleftarrow \setminus - (n_k); \\
\quad \quad \exists\{(n_i) \cdots (P)\} \Rightarrow (n_i) \cdots \setminus \cdots (P); \\
\quad \} \\
\} \\
\\
\text{transfer}(n_i, n_j)\{ \\
\quad \text{copy}(n_i, n_j); \\
\quad \text{kill}(n_i); \\
\}
\end{array}$$

Figure A.2 Definition of the *copy*, *kill*, and *transfer* operations.

Tables A.6 and A.7 contain the execution behavior and the effects of each RAE on the intermediate graph. For example, the first rule in Table A.6 is for the RAE $\text{pop}(s_i)$. When a `pop` bytecode instruction is executed, it discards the top entry on the operand stack. Therefore, when the reverse traversal analysis encounters a $\text{pop}(s_i)$ RAE, it knows that if the operand stack location represented in the working set by node s_i contains any edges, these are from the conservative edges added to the CFG and are actually extraneous information. This is true because this location was actually cleared in the forward execution of the instruction and any object reference that was loaded into it

Table A.6 Rules for adding nodes and edges based on RAE entries, part a.

RAE	Execution behavior/reverse analysis traversal effect
$l : pop(s_i)$	execution order: clear contents of s_i analysis effect: kill(s_i)
$l : constant(s_i)$	execution order: load $c_{id} \rightarrow s_i$ analysis effect: add(r_j), transfer(s_i, r_j), add property edge: (r_j) $-\neg$ (c_{id}, l)
$l : swap(s_i, s_j)$	execution order: trade positions, $s_i \rightarrow temp$, $s_j \rightarrow s_i$, $temp \rightarrow s_j$ analysis effect: add(temp), transfer($s_i, temp$) transfer(s_j, s_i), transfer(temp, s_j), remove(temp)
$l : store(\nu_i, s_j)$	execution order: move $s_j \rightarrow \nu_i$ analysis effect: kill(s_j), transfer(ν_i, s_j)
$l : duplication(s_i, s_j)$	execution order: copy(s_i, s_j) analysis effect: transfer(s_j, s_i)
$l : load(s_i, \nu_j)$	execution order: copy(ν_j, s_i) analysis effect: transfer(s_i, ν_j)
$l : exception(s_i)$	execution order: empty operand stack, put s_i on stack, transfer control to handler analysis effect: add property edge and node: (s_i) $-\neg$ (T, l)
$l : lock(s_i)$	execution order: obtain lock for s_i remove(s_i) analysis effect: add property edge and node: (s_i) $-\neg$ (L, l)
$l : unlock(s_i)$	execution order: release lock for s_i remove(s_i) analysis effect: add property edge and node: (s_i) $-\neg$ (U, l)
$l : allocation(s_i)$	execution order: add s_i of type A_{id} analysis effect: add(r_j), transfer(s_i, r_j), add property edge and node: (r_j) $-\neg$ (A_{id}, l)
$l : array\ store(s_i, s_j)$	execution order: move s_j to memory location specified by s_i $s_i \rightarrow s_j[index]$ analysis effect: add to information s_i and s_j (s_j) \dashrightarrow (s_i); add property edge and node: (s_i) $-\neg$ (W, l); (s_j) $-\neg$ (W, l)

- id is the index into the *Constant Pool* of the class file which uniquely identifies the object within the class file.

Table A.7 Rules for adding nodes and edges based on RAE entries, part b.

RAE	Execution behavior/reverse analysis traversal effect
$l : \text{array load}(s_i, s_j)$	<p>execution order: place the contents of $s_i[\text{index}]$ in stack location s_j $s_i[\text{index}] \rightarrow s_j$</p> <p>analysis effect: save the information concerning s_j, update properties of s_i $\text{add}(r_k); \text{transfer}(s_j, r_k); (s_i) \dashrightarrow (r_k);$ add property edge and node: $(s_i) - -(R, l); (r_k) - -(R, l)$</p>
$l : \text{field write}(s_i, s_j)$	<p>execution order: move $s_i.\text{id} \leftarrow s_j$</p> <p>analysis effect: $(s_i) \dashrightarrow (s_j);$ add property edge and node: $(s_i) - -(W, l); (s_j) - -(W_{\text{id}}, l)$</p>
$l : \text{field read}(s_i, s_j)$	<p>execution order: copy $s_i.\text{id} \rightarrow s_j$</p> <p>analysis effect: $\text{add}(r_k); \text{transfer}(s_j, r_k);$ add property edge and node: $(s_i) \dashrightarrow (s_j); (s_i) - -(R, l); (r_k) - -(R_{\text{id}}, l)$</p>
$l : \text{static read}(s_i)$	<p>execution order: copy $G_{\text{id}} \rightarrow s_i$</p> <p>analysis effect: $\text{add}(r_k); \text{transfer}(s_i, r_k);$ add property edge and node: $(r_k) - -(Gr_{\text{id}}, l)$</p>
$l : \text{static write}(s_i)$	<p>execution order: move $G_{\text{id}} \leftarrow s_i$</p> <p>analysis effect: add property edge and node: $(s_i) - -(Gw_{\text{id}}, l)$</p>
$l : \text{return}(s_i)$	<p>execution order: move callee(s_i) \rightarrow caller(s_j)</p> <p>analysis effect: add property edge and node: $(s_i) - -(F_{-1}, l)$</p>
$l : \text{invoke}$ $(s_i : P_{-1}, s_i : P_j, ..s_n : P_k)$	<p>execution order: $\forall x = i, ..n; y = j, ..k :$ $\text{caller}(s_x) \rightarrow \text{callee}(\nu_y); \text{callee}(s_j) \rightarrow \text{caller}(s_i)$</p> <p>analysis effect: $\text{add}(r_m); \text{transfer}(s_i, r_m);$ add property edge and node: $(r_m) - -(P_{-1}, l);$ $\forall x = i, ..n :$ add property edge and node: $(s_x) - -(P_k, l)$ where k is the kth parameter</p>
method descriptor $(F_i, ..F_n)F_{-1}$	<p>analysis effect: $\forall j = i, ..n :$ add property edge and node: $(\nu_j) - -(F_k, l)$ where k is the kth formal</p>
method attributes (synchronized)	<p>analysis effect: add property edge and node: $(\nu_0) - -(L, -1)$</p>

- id is the index into the *Constant Pool* of the class file which uniquely identifies the object within the class file.

should have been transferred to another node before reaching this RAE. The analysis, therefore, kills any edges leaving this node, thus removing it from the set of “live” nodes.

The size of the intermediate graph is defined by the number of data nodes contained within it. It can increase whenever new nodes are added to it. Only data nodes of type r can be added to a graph after initialization of the graph. There are six RAEs in Tables A.6 and A.7 that add these types of data nodes. The bounding of this set is discussed later.

There are two additional entries in Table A.7 that were not specified in the set of RAEs in Table A.3. They are the *method descriptor*, which is used to identify the formals of the method, and the *method attributes*, which is used to assign additional properties to the nodes. Both of these are processed on the final graph and only add property nodes to the graph. The processing of the *method descriptor* helps to eliminate additional extraneous data from the final graph. This is because only the data nodes of type ν corresponding to formal parameters of the method were defined at entry to the method. Any reference that occupied a legitimate data node of type ν that was defined within the method and not defined as a formal, should have been transferred to a node of type r by the analysis. Therefore, since we have assumed verifiable bytecode, we can safely kill the edges attached to any node ν not identified as a formal to the method. If the *attributes* information defines the method as *virtual* and *synchronized*, this enables the analysis to add the corresponding locking and unlocking operations to the ν_0 node. This additional property information is used by some forms of interprocedural analysis


```

access flags: 0
name index: 19, Bar
descriptor index: 18, (LClazz;LClazz;)V
max stack: 2
max locals: 4
code length: 18
0:  new cpIndex: 2, Clazz
3:  dup
4:  invokespecial Clazz <init> ()V
7:  astore_3
8:  aload_3
9:  aload_1
10: putfield cpIndex: 6 class: Clazz field: f
13: aload_3
14: putstatic cpIndex: 7 class: Clazz field: g
17: return

```

Figure A.3 Bytecode representation of Bar from Figure 2.3.

including identification of extraneous synchronization and identification of potential data races.

To explain the analysis, we start with the simple case of a single BB method. The algorithm for iterative settling of multiple BB CFGs will be presented later.

We start by using the same sample method, Bar from Figure 2.3. The bytecode representation of Bar is shown in Figure A.3. Note that there is no control flow or intraprocedural exception handler in this method; therefore, it contains only one BB. Following the steps outlined, the temporary locations used by each bytecode instruction are determined as in [29]. Then the RAEs for the method are formed. These are shown in Figure A.4.


```

method descriptor( $F_0, F_1, F_2$ )
0: allocation( $s_0$ )
3: duplication( $s_0, s_1$ )
4: invoke( $s_1 : P_0$ )
7: store( $\nu_3, s_0$ )
8: load( $s_0, \nu_3$ )
9: load( $s_1, \nu_1$ )
10: field write( $s_0, s_1$ )
13: load( $s_0, \nu_3$ )
14: static write( $s_0$ )

```

Figure A.4 RAE representation of Bar from Figure A.3.

The graph is constructed by applying the rules in Tables A.6 and A.7 to the RAEs in Figure A.4 in a backwards dataflow analysis. By performing the analysis in a backwards propagation, we are able to fold out the temporary locations as we compute the final graph. Figure A.5 steps through this process. Property nodes are shown with dotted circles while data nodes are shown in solid circles. The edges follow the edge connection convention given in Table A.5. For clarity, only nodes that are part of the “live” set of data nodes are shown in the intermediate graphs formed during the RAE processing.

Starting at the bottom of the RAE representation for Bar shown in Figure A.4, the first RAE encountered is the 14:*static write*(s_0). This adds the property node labeled ($Gw_7, 14$) to the graph and connects it via a property edge to the data node (s_0). The Gw_7 denotes that this is a *global* property node which writes to a global location specified by the information contained in the *constant pool* at location 7. The 14 is the line number the global write occurred at. The intermediate graph formed by processing this RAE is shown in Figure A.5(a).

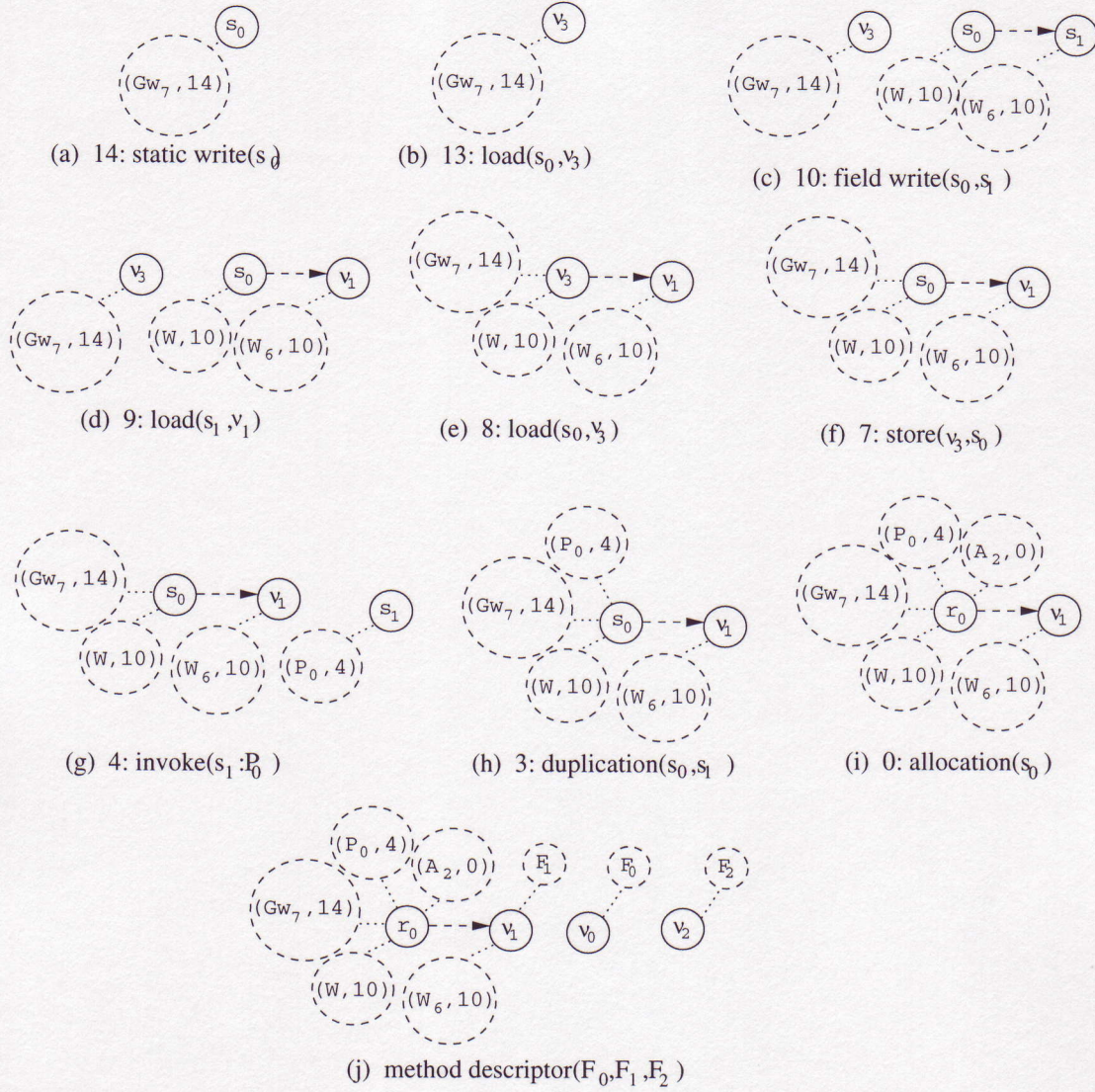


Figure A.5 Graph construction for Bar in Figure A.4.

Progressing in the reverse direction up the list of RAEs shown in Figure A.4, the next RAE, 13:load(s_0, v_3), simply transfers the information between two nodes in the working set. It transfers the edges connected to node s_0 to node v_3 . It does not add any new nodes to the graph. The intermediate graph formed from processing this RAE with the graph from Figure A.5(a) as input is shown in Figure A.5(b).

Progressing in the reverse direction, the next RAE, $10:field\ write(s_0, s_1)$, forms a field accessing edge between nodes s_0 and s_1 . It also attaches property nodes to each of these – one to the parent object, s_0 , which simply specifies it was involved in a write operation at line 10, and one to the object stored in the field which gives the *constant pool* index for the field *id*, 6. Since we define the size of the graph as the number of data nodes contained within the working set, the processing of this RAE did not affect the size. This is because, as we previously stated, the nodes s and ν are the initial working set of nodes and the number of them is defined in the bytecode file for the method. It does not grow or shrink in size during the analysis, and it constitutes the initial set of nodes, remaining in the working set at all times. They can transition to and from the *live* set of nodes but are never removed from the working set. They are removed from the graph only in the final step, reducing the graph to only *live* nodes, if they are not part of the *live* set. The processing of this RAE added property nodes to the graph but no data nodes as seen in Figure A.5(c).

The next RAE is the $9:load(s_1, \nu_1)$. This performs a transfer, transferring the edges attached to s_1 to node ν_1 . Because we only show “live” nodes in the intermediate graphs in Figure A.5, node s_1 is not shown in Figure A.5(d). Only node ν_1 is shown in the intermediate graph obtained from processing this RAE. Note that s_1 does remain in the working set for the intermediate graph. Comparing the graphs in (c) and (d), the incoming field accessing edge that was pointing to node s_1 is now moved to node ν_1 and the property edge between nodes s_1 and $(W_6, 10)$ is now moved to between nodes ν_1 and $(W_6, 10)$.

Progressing up the RAEs for **Bar**, the next one encountered, $8 : load(s_0, v_3)$, is also a load RAE and performs a similar edge transfer as that performed by the previous load. This can be seen by comparing graphs (d) and (e) in Figure A.5. The property edge between nodes s_0 and $(W, 10)$ has been moved to now connect nodes v_3 and $(W, 10)$. The node s_0 no longer has any edges attached to it and as such is not shown in the graph in Figure A.5(e).

This RAE is followed in the reverse progression by the RAE $7 : store(v_3, s_0)$. The store RAE again performs a transfer, this time transferring the edges from v_3 to s_0 . The effects of this transfer can be seen by comparing the intermediate graphs in Figure A.5(e) and (f). The edge between nodes v_3 and $(Gw_7, 14)$ and the edge between v_3 and $(W, 10)$ now exist between these two property nodes and the data node s_0 . The node v_3 no longer has any edges attached to it and is therefore no longer in the “live” set and not shown in Figure A.5(f).

The next RAE in reverse progression is $4 : invoke(s_1 : P_0)$. This RAE attaches a property node to data node s_1 making it “live” again. This property node identifies the reference in data node s_1 as being used as the 0th parameter to a callee called at line number 4. The intermediate graph resulting from processing this RAE is shown in Figure A.5(g).

The *invoke* RAE is followed in reverse progression by $3 : duplication(s_0, s_1)$ which identifies data nodes s_0 and s_1 as being the same referenc value. Therefore, the processing of this RAE transfers the edges connected to s_1 to s_0 . The effects of this transfer on the intermediate graph can be seen by comparing graph (g) and (h) in Figure A.5. Note that

data node s_1 no longer has any edges attached to it, therefore removing it from the set of “live” nodes.

The next RAE in reverse progression is $0 : allocation(s_0)$. An *allocation* RAE identifies a data location as becoming “live” at this point in the method. Since we are traversing backwards, we do not know anything about the data node before this point in execution order and have accumulated behavior information about the reference that was stored into it at this point. We therefore need a new data node to hold this accumulated information. We create a node r_0 and add it to the graph. We then transfer the edges connected to node s_0 to the new node r_0 , removing s_0 from the set of “live” nodes. Note that since we have added a data node to the working set, we have increased the size of the intermediate graph. The effects of this on the intermediate graph can be seen by comparing the graphs (h) and (i) in Figure A.5.

Finally, we reach the *method descriptor* RAE which identifies which data nodes in the set of ν_i nodes in our working set were live coming into the method. For this method, the *method descriptor*(F_0, F_1, F_2) identifies the data nodes ν_0 , ν_1 , and ν_2 as being “live” on entry, respectively. We denote this in the intermediate graph by attaching the property nodes F_0 , F_1 , and F_2 to their corresponding data nodes. This can be seen in Figure A.5(j). Note that the graph in Figure A.5(j) only has nodes of type ν corresponding to formals to the method and nodes of type r among its “live” set. Since only the “live” set of nodes is shown, and we defined the final graph as the subgraph containing only “live” nodes reached at the entry point of the method upon completion of the reverse traversal, this is also the final graph for this method.

Although not immediately obvious, the final graph of Figure A.5(j) contains the same information contained in the CDG in Figure 2.4(c). Both graphs show that formal parameters 0 and 2 were not used within the method. This is shown in the CDG of Figure 2.4(c) by these nodes being unattached and in the final graph in Figure A.5(j) by only having formal parameters property nodes attached to these data nodes. For formal parameter 1, both graphs contain the same information though in different forms. The node labeled 28 in Figure 2.4(c) is analogous to the data node labeled r_0 with the property $(A_2, 0)$ attached in Figure A.5(j). The field node in Figure 2.4(c) labeled $(f, 31)$ and attached via a field access edge to node 28 and a dataflow edge to node P_1 , is analogous to the field access edge attaching data nodes r_0 and ν_1 with the properties $(W, 10)$ and $(W_6, 10)$ attached to them, respectively. The use of node 28 as a parameter is denoted by the dataflow arc to the node labeled $(0, 29)$ in Figure 2.4(c). This represented in the final graph of Figure A.5(j) by the property node labeled $(P_0, 4)$ attached to the data node r_0 . The difference in the line number between these two entries is because Figure 2.4(c) is a source level graph while Figure A.5(j) is a bytecode level graph. The invocation in the source code at line number 29 is the same invocation that occurs at line number 4 in the bytecode representation. Finally, the node representing the dataflow of the global write in Figure 2.4(c) and labeled $(g, 32)$ is denoted in Figure A.5(j) as the property node $(G_7, 14)$ attached to data node r_0 . Again, the variance in the line numbers is due to the source versus bytecode representation; however, the access is the same.


```

0:  formGraph(refinedCFG){
1:      worklist := initializeWorklist(refinedCFG);
2:      while(worklist ≠ ∅){
3:          BBj := removeNext(worklist);
4:          testGraph :=  $\sqcup_{i=0}^n G_{ti} \forall i \in \text{child}(BB_j)$ ;
5:          if( $\neg \text{visited}(BB_j) \parallel \text{testGraph} \not\subseteq G_{bj}$ ){
6:              Gbj := Gbj  $\sqcup$  testGraph;
7:              Gtj := process(BBj);
8:               $\forall i \in \text{parent}(BB_j)\{$ 
9:                  if( $G_{tj} \not\subseteq G_{bi} \ \&\& \ BB_i \neg \in \text{worklist}$ ){
10:                     worklist.append(BBi);
11:                 }
12:             }
13:         }
14:     }
15: }

```

Figure A.6 The algorithm for processing CFG to form graph

Next we discuss the construction of this intermediate graph on a CFG containing more than one BB. We will describe the transformation of the final graph into the actual CDG in Figure A.5(j) after fully specifying this step.

A.1.4 Iterative, backwards, dataflow algorithm

The iterative part of the backwards dataflow algorithm uses a worklist based algorithm. It starts with the refined CFG reduced to an RAE representation and places the BBs in the CFG in reverse topological order onto a worklist. The algorithm is given in Figure A.6. The algorithm iterates over the BBs on the worklist until no more property or data nodes can be added to the graph and a final graph is formed. This final graph is

the top graph of the entry BB for the method. We begin by defining the following two terms:

Definition 9 For a given BB_i , G_{ti} is the graph associated with the topmost point or entry point for forward execution of BB_i . It is initialized to an empty graph.

Definition 10 For a given BB_i , G_{bi} is the graph associated with the bottommost point or the exit point for forward execution of BB_i . It is initialized to the join of the top graph of all of its children.

The algorithm starts by initializing the *worklist* on line 2 of Figure A.6 by placing the BBs in the *refinedCFG* on the *worklist* in reverse topological order. Next it processes each BB on the *worklist* until the list is empty. To process a given BB_j , it starts by forming a join of the top graphs of all of its children BBs. This is shown on line 4 of the figure. Then on line 5 of Figure A.6, it performs a test to see if the BB has not been visited or if the *testGraph* created from the join of the children's top graphs is not a subset of the bottom graph of the BB. If either of these tests is true, it then initializes the bottom graph of the BB to the join of its current bottom graph with the *testGraph* and processes the BB to arrive at a new top graph. This is shown on lines 6 and 7 of Figure A.6. It then checks for each parent of the BB to make sure that its new top graph is still a subgraph of the parent. If it is not and the parent is not already on the *worklist*, it appends the parent to the end of the *worklist* to be reprocessed. This is shown on lines 8-12 of Figure A.6. The algorithm only terminates when there are no further BBs to process on the *worklist*.

The key to proving termination of the algorithm lies in the proof that there is a maximum saturation point for the join operation performed at line 4 of Figure A.6. This is because if the bottom graph of a BB currently being processed is not a subset, then on line 6 of Figure A.6, the bottom graph of a given BB is always set to at least the join of the result, thereby making it at least equal and, by definition, a subset. Therefore, if the join forms a graph that saturates at a maximum point and cannot grow beyond that point, then the bottom graph must be at least equal to that after it is set to the result on line 6 and will not be reprocessed. The key properties in this proof are that a join operation will only increase the number of entries in a graph and that the potential number of entries is finite. Although the data nodes ν and s are finite in size and distinct, the graph size can increase with the addition of data nodes of type r . Therefore, the join operation must be capable of recognizing that two data nodes of type r actually represent the same object and can be merged into one.

Definition 11 *Two nodes, n_i and n_j , from two graphs G_x and G_y are defined to be joinable if and only if at least one of the following is true.*

- $n_i = s_a$ and $n_j = s_a$; $a = 0, \dots, (\max_stack - 1)$.
- $n_i = \nu_b$ and $n_j = \nu_b$; $b = 0, \dots, (\max_local - 1)$.
- $n_i = r_c$ and $n_j = r_d$ and at least one of the following holds:
 - r_c has property (P_{-1}, l') and r_d has property (P_{-1}, l') . They were both returned from the same callee method at the same line number.

- r_c has property (A_m, l') and r_d has property (A_m, l') . They were both allocated with the same type at the same line number.
- r_c has property (Gr_m, l') and r_d has property (Gr_m, l') . They were both read from the same global location at the same line number.
- r_c has property (c, l') and r_d has property (c, l') . They were both initialized to the same constant reference value.
- r_c has property (R_m, l') and $(n_a) \dashrightarrow (r_c)$ and r_d has property (R_m, l') and $(n_a) \dashrightarrow (r_d)$. They were both read as the same field from the same base object at the same line number.

As given in Table A.5, the size of the ν and s nodes is finite. These are defined and given in the method representation in the classfile. Referring back to the rules for adding nodes to the graph as given in Figure 2.2, there are only a finite set of RAEs that can add nodes of type r to the graph. Each of these adds a property field to the node which is used to identify a join operation as given in definition 11. Therefore, since the number of RAEs in a method is finite, and the number of RAEs that can increase the size of the nodes r is finite, and by definition 11 no two nodes r will have the same property after the join, the maximum number of r nodes contained in the resulting joined graphs is also finite. Therefore, the graph size is finite and has a saturation point.

The subset test at line 5 of Figure A.6 then tests for each node in *testGraph* that there is a node in G_{bj} that contains at least the same set of properties.

Table A.8 Property nodes divided into “defines” and “uses.”

define	use
<i>constant</i>	<i>pop</i>
<i>allocation</i>	<i>swap</i>
<i>static read</i>	<i>store</i>
<i>array load</i>	<i>duplication</i>
<i>field read</i>	<i>load</i>
<i>invoke($s_i : P_{-1}$)</i>	<i>exception</i>
	<i>lock</i>
	<i>unlock</i>
	<i>array store</i>
	<i>field write</i>
	<i>static write</i>
	<i>return</i>
	<i>method descriptor</i>
	<i>method attributes</i>

The maximum size of the graph is a function of the number of “defines” that can occur within a method. This is because “defines” are capable of adding data nodes to the graph. Therefore, we define the size of the final graph as follows:

$$size(fG) = number(RAE_{define}) \quad (A.1)$$

This states that the size of the final graph is a function of the number of “define” RAEs contained within an RAE representation of a bytecode method. We further define a “define” RAE as one capable of adding a data node of type r to the graph. Table A.8 shows the RAEs from Tables A.6 and A.7 divided into “defines” and “uses.” The RAEs capable of adding data nodes of type r to the graph are the RAEs *constant*, *allocation*, *array load*, *field read*, *static read*, and *invoke*. The *invoke* RAE only adds a data node of type r if it

returns a reference value. There is a theoretical limit to the maximum potential size any given graph could reach. This limit is defined by the limits given in [29]. The “define” RAEs can be viewed as a function of the method descriptor and the size of the method. The number of parameters passed to a method is limited to 255 as specified in [29]. The maximum size of a bytecode method is limited to 2^{16} by the size of the field specifying the exception handler coverage ranges in the bytecode file specifications [29]. Data nodes of type r that are added to the intermediate graph have at least one “define” property node attached to them. There are six RAEs capable of adding a data node of type r to the graph as shown in Table A.8. The corresponding bytecode instructions for these RAEs is shown in Table A.3. Of this set, the smallest size instruction is the `ldc` instruction, which is two bytes in length [29]. Therefore, a completely theoretical maximum size for the intermediate graph, which is given by a theoretical bytecode method of the maximum size with the maximum number of parameters and containing only instructions of type `ldc`, is given by:

$$size = (255 + 2^{16}/2) \approx 2^{15} \quad (\text{A.2})$$

This is purely a theoretical maximum limit to the potential size of a graph and in practice the final graph is much smaller.

A.1.5 Connecting defines and usages

The current graph represents unique objects as data nodes and the usage and definitions for the objects as properties attached to the data nodes. In this phase, we simply

definitions

(A, l)	$::=$ an allocation occurring at line l
(F_j)	$::=$ the j th formal parameter to the method
(c, l)	$::=$ a constant value loaded at line l
(P_{-1}, l)	$::=$ the return value from a method called at line l
(Gr, l)	$::=$ a read access of a global variable occurring at line l

uses

(L, l)	$::=$ a lock operation occurring at line l
(U, l)	$::=$ an unlock operation occurring at line l
(R, l)	$::=$ a read operation occurring at line l
(W, l)	$::=$ a write operation occurring at line l
(Gw, l)	$::=$ a write access of a global variable occurring at line l
(T, l)	$::=$ a thrown exception occurring at line l
(P_j, l)	$::=$ the j th formal parameter to the method called at line l

Figure A.7 Property nodes broken into definition and usage nodes.

replace the objects with their “define” property node and attach the “use” property nodes to it. Figure A.7 divides the property nodes from Table A.5 into “definitions” and “uses.” The formation of the final CDG is performed in the following steps:

1. Mark all “define” property nodes.
2. Attach all the remaining “use” property nodes to the “define” nodes via dataflow edges from the “define” property node to the “use” property node.
3. Move all field accessing edges to corresponding “define” property nodes.
4. Remove all now extraneous data nodes (type r and ν) from the graph.

Figure A.8 shows the steps for performing this final phase on the intermediate graph from Figure A.5.

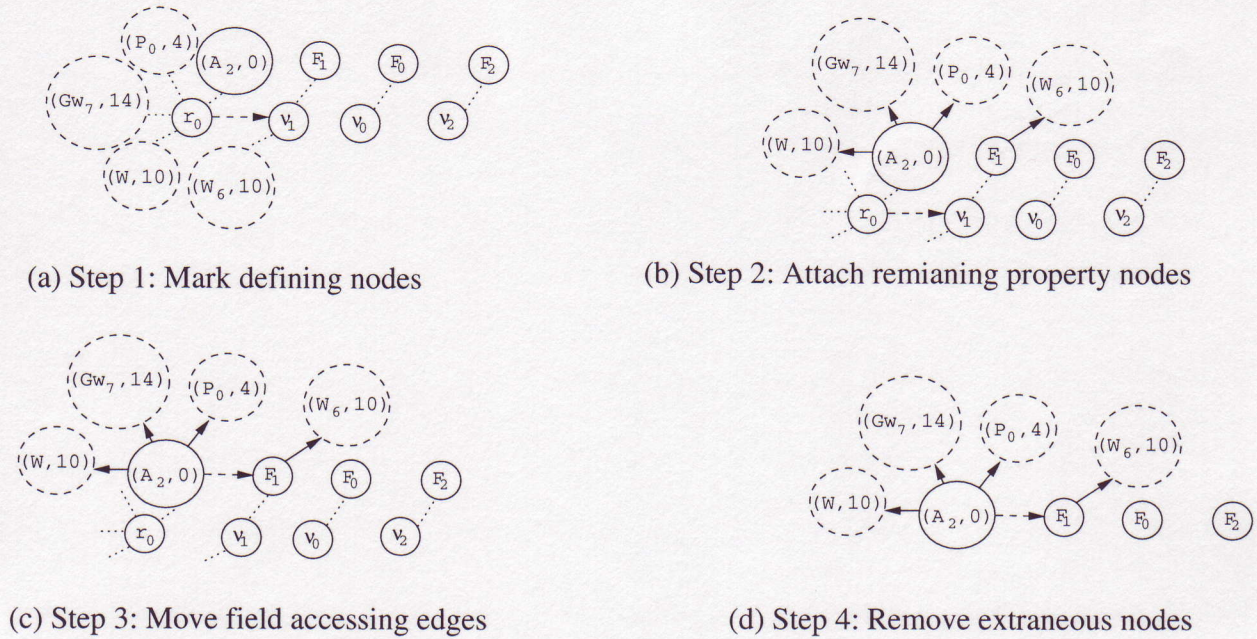


Figure A.8 Final CDG formation from the graph for Bar in Figure A.5.

A.2 Annotations

The primary purpose of this section is to describe the format of the annotation used to represent the CDG within classfiles. We focus primarily on the entries and what each field means to enable users to read and understand the annotations.

To form the actual persistent annotation, the remaining nodes in the graph are sorted into a table based on their bytecode order. The links between the entries are represented in the table as offsets from the entry's location to the entry it is linking to. This table is annotated into the classfiles.

The annotation used to represent the CDG in a classfile is designed to use a minimum amount of space while still maintaining the information within the CDG. The CDG

represents relationships between nodes as directed edges on a graph. To convert this graphical view into an annotation, the nodes in the CDG are placed in a table. The edges in the CDG are then represented as directed pointer to their corresponding entries in the table. We order the nodes in the table starting at their bytecode address. Formals to a method take up the first few entries in the table where their index into the table directly matches their corresponding formal index. Edges to and from parameter nodes then point to and from these entries. Forward edges are then transformed into offsets from the entry's location to the table entry it points to. The table entries are then reduced in size and annotated into the classfiles.

Figures A.9 and A.10 show the format for each of the entries in the annotated table representation. There are several points to note about the entries. First, note that these entries are variable length. The use of variable length entries allows the annotation to use the minimum size for each type of entry instead of the maximum size needed to handle any given entry in the set of entries. Second, edges are now offsets from the current position. The offsets now point to where within the method bytecode stream the object instance is first defined. This could be as a formal passed to the method, an object created within the method, a field read within the method, a returned value from another method, etc. The formals passed to the method are left as implicit entries at the top of the table. The table starts indexing its bytecode entries at the maximum local variable index value. This value is given as part of the method representation in the bytecode file. The bytecode offset is an offset from the previous entry that corresponds

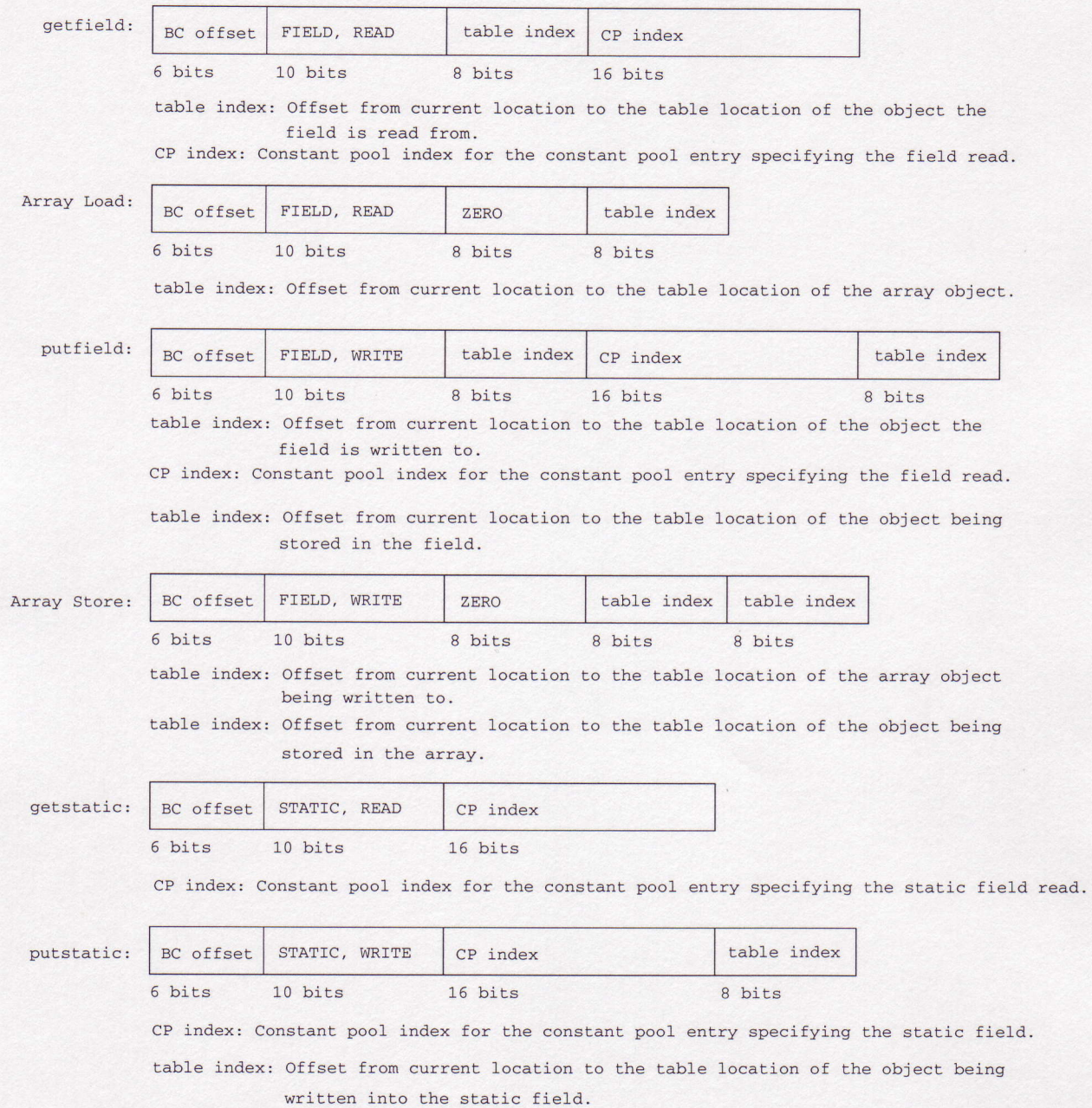


Figure A.9 The format for the table entries used to annotate the CDG information into a class file.

returned by:

BC offset	RETURNED, BY	table index
6 bits	10 bits	8 bits

table index: Offset from current location to the table location of the object being returned.

returned from:

BC offset	FIELD, WRITE
6 bits	10 bits

parameter:

BC offset	PARAMETER	table index	Parm num
6 bits	10 bits	8 bits	8 bits

table index: Offset from current location to the table location of the object being passed.

Parm num: The parameter number the object corresponds to.

create:

BC offset	CREATE	CP index
6 bits	10 bits	16 bits

CP index: Constant pool index for the constant pool entry specifying the object type being created.

Monitor:

BC offset	LOCK	table index
6 bits	10 bits	8 bits

table index: Offset from current location to the table location of the object being locked/unlocked.

thrown:

BC offset	THROWN	table index
6 bits	10 bits	8 bits

table index: Offset from current location to the table location of the object being thrown.

Figure A.10 The format for the table entries used to annotate the CDG information into a class file.

to the distance in bytes within the bytecode array for the method between the previous table entry and the current entry.

The maximum address length for a bytecode address is 16 bits as given in [29]. However, we are starting at the lowest indexed node and progressing forward. Therefore, instead of representing an absolute address within the table, we use an offset from the previous bytecode address. Since the table entries are in ascending order, the value is always positive. We choose a 6-bit size for this field based on several factors including the desire to maintain byte boundaries, the average bytecode distance between entries in the files analyzed and, finally, the size and desired representation for entry types which shared the field space. The 6-bit field size means the maximum difference between any two entries is limited to 63 bytes. However, it is possible for the distance between two entries to be greater than 63 bytes. Under these conditions, a Bytecode Line Number stub is used. The format of this stub is shown in Figure A.11. The stub contains a value of 63 for the offset followed by a zeroed-out usage field to denote it as a bytecode stub. This gives the stub a total size of 16 bits or 2 bytes. When reading the access table and a bytecode stub is encountered, the runtime simply adds the 63 to the offset of the next entry. In this fashion, the stubs can be layered to provide any necessary amount of offset between two entries in the table.

One of the fields consistent across all of the entries in Figures A.9 and A.10 is a usage field. This field contains a single bit representation of the type of access. It is 10 bits in length, thus covering the ten potential types of accesses. Figure A.12 specifies the usage type of each entry based on the bit location set. Noting that the bytecode address stub

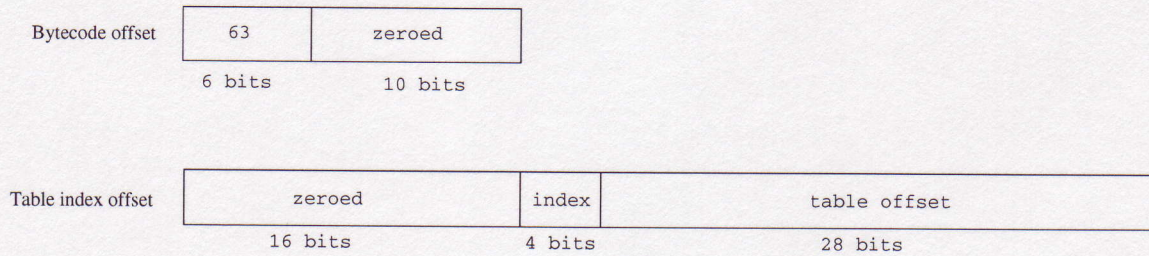


Figure A.11 The format for an index stub within the annotated table.

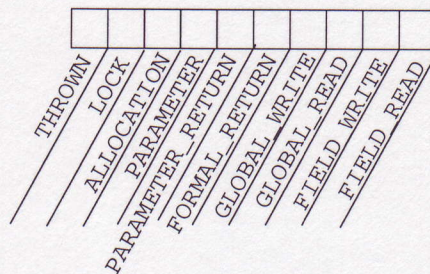


Figure A.12 Entry type specification for usage bit fields.

left this field cleared, this then completely specifies the locations in this field. This usage value, in combination with the other field entries, determines the type and size of the other fields in the entry. For example, if the Field Read bit is set, then the next field is 8 bits in length and contains the indexed offset into the table that is the offset from this usage location to the location within the table where the object instance was defined. If this field is set to zero, then the entry is for an array access and the next field is the table entry offset. For cases when the field is nonzero, it is followed by a 16-bit constant pool index. The entry in the constant pool for the given class file is an entry of type `field info` which contains the information of the type of field. Therefore, this pair of values


```

attribute_info{
    u2 attribute_name_index;    // this is IMPACT_AccessTable
    u4 attribute_length;        // length in bytes of the table
    u1 info[attribute_length];  // table
}

```

Figure A.13 The format of the attribute field used to hold the table.

uniquely identifies the field type and memory location. Although the constant pool index covers all possible values, the 8-bit table index does not. Since the table index field is finite in length, the offset is ± 127 which limits the distance between a use and a define in the table. This is again handled by the use of a stub the format of which is shown in Figure A.11. Since the full bit space of the 10-bit usage field is specified, we denote that this is a table index stub by setting both the bytecode offset and the usage fields to zero. In this fashion, it is distinguished from the bytecode index stub and does not require expanding the usage bit space to accommodate the type. Looking at Figures A.9 and A.10, there are four potential byte locations within a given entry that are offsets into the the table. The next field, index, specifies which of these four locations the stub is associated with. This then leaves a 28-bit field to represent the offset.

The table is annotated into the bytecode file using the attribute info field which is part of the method info field within the bytecode file. The format of the attribute info field as well as comments identifying what information is present in each field, is shown in Figure A.13.

APPENDIX B

SIMULATOR ARCHITECTURE

The simulator used to simulate a Java runtime consists of four major parts. These parts are labeled in Figure B.1. In this appendix we describe each of these four main sections in detail.

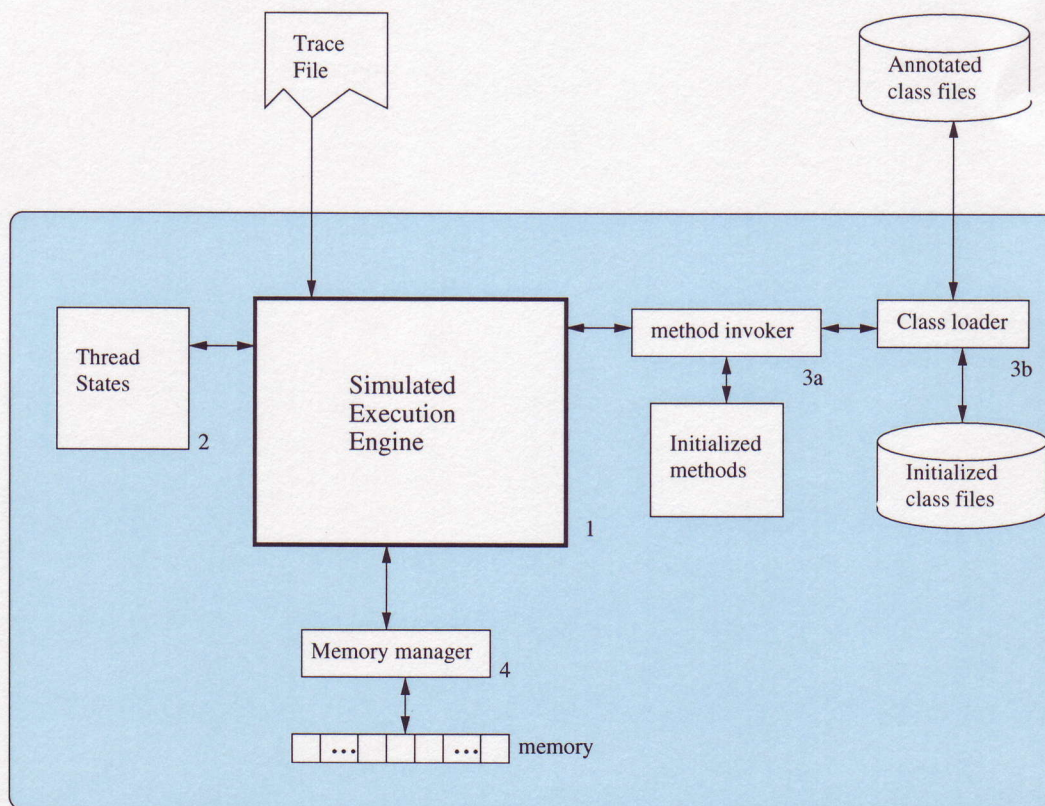


Figure B.1 An abstract overview of the simulated runtime environment.


```

virtual jobject spec.benchmarks._205_raytrace.Face.GetVert(jint)(I)Lspec/benchmarks/_205_raytrace/Point;
T9      105900867      0
T10     105900868      0
T9      105900869      1
T10     105900870      1
T9      105900871      4
T10     105900872      4
T9      105900873      5
T10     105900874      5
T9      105900875      6
T10     105900876      6

```

Figure B.2 Section of the trace file for the `_227_mtrt` benchmark from the SpecJVM98 benchmark suite.

B.1 Simulated Execution Engine

The input to the Simulated execution engine is a line read from a trace file collected from the HP Hotspot 1.0 VM. The line read consists of two potential types, an invocation line, which contains a full path resolved signature for the method invoked, or an execution line, which contains a thread id (TID), the absolute number of the instruction and the bytecode line number at which the instruction occurred. Figure B.2 gives a small section from one of the SPECjvm98 benchmarks [59]. The first line in this figure is an invocation line. This line is actually a combination of the standard trace mechanism output and a modification we made to record the actual method signature. If this is read from the trace file, it is passed to the method invoker to recover the method.

The next line shown in Figure B.2 is a trace output of an execution of one of the lines of bytecode. The format of this line consists of the thread ID, the absolute execution order for the instruction, and the bytecode line number in the method that the instruction occurred at. We have also included a disassembled version of the bytecode for the method `GetVert` in Figure B.3. The disassembled version also contains the CDG table for this


```

access flags: 1, public
name index: 16, GetVert
descriptor index: 9, (I)Lspec/benchmarks/_205_raytrace/Point;
attribute count: 2
attribute[0]
name index: 12, Code
max stack: 2
max locals: 2
code length: 7
0:    aload_0
1:    getfield cpIndex: 5
           class: spec/benchmarks/_205_raytrace/Face
           field: Verts, [Lspec/benchmarks/_205_raytrace/Point;
4:    iload_1
5:    aaload
6:    areturn
attribute[1]
name index: 26, IMPACT_AccessTable
number of entries: 3
Table[0]: (implied, local variable 0)
Table[1]: (implied, local variable 1)
Table[2]: (1, field_read, table offset: -2, CP index: 5)
Table[3]: (4, field_read, Array, table offset: -1, table offset: 0)
Table[4]: (1, returned_by, table offset: -1)

```

Figure B.3 Bytecode disassembly for the method GetVert.

method. This is shown at the bottom of the figure. The first two entries in the CDG table correspond to the implied entries for the local variable locations described in Appendix A. The remaining three entries are the entries that were annotated in the bytecode file. When the line is loaded, the simulated execution engine first checks to see if the thread ID matches the thread ID for the thread state it currently has loaded. If it does not, it places a request to the thread state manager to recover the state for that thread and switches its loaded state to correspond to that thread. The current state of the thread that was previously loaded is passed to the thread state manager for storage. The

absolute instruction order for the trace line is ignored by the simulator since it has no bearing on the behavior of the simulator. The bytecode line number is then used to access the internal representation for the CDG for the method representation for this thread's context to see if there is an entry corresponding to this bytecode line number. If there is, then the CDG is updated.

The updating of the CDG when a trace line corresponding to an entry is encountered, varies depending on the type of the entry and the type of analysis being simulated. For the simulation of the use of the Object Connection Graph (OCG) described in Chapter 4, the CDG is updated only for the exact or oracle method. It is used to track the exact escaping state for the memory entries. To accomplish this, each entry in the CDG has a pointer to the memory location it uses. These are established either when an allocation is reached or when the OCG analysis is conducted by the method invoker. The memory locations pointed to by the entries are allocated and controlled by the memory manager. Any requests for new memory locations go through this manager. The format and tracking information for the memory locations follows later in this appendix.

When an invocation entry is encountered, the call stack for the thread is updated to reflect the call. The actual signature for method is constructed in the same fashion that the virtual machine uses. The simulator recovers the constant pool index corresponding to the method type from the bytecode for the method. It then constructs the signature based on the constant pool entries. Since we are dealing with a trace file, we know exactly which version of this method is subsequently executed, so object type resolution to locate the method is not necessary. Instead, when the actual instructions corresponding to the

callee method are encountered in the trace file, the signature stored on the thread's call stack is compared with the actual method called to be certain they match.

The handling of return instructions is a little more involved than that of invocations. When a return instruction is encountered in the instruction stream, a flag is set in the thread's state to indicate that it executed a return. The return point is then verified with the call stack entry for the method prior to proceeding with the execution. The verification of the return point is multileveled. First, the callee is verified to be certain it is returning from the method the call stack claims was called. Then a look-ahead is performed for the caller thread, to verify that the next instruction being executed follows logically from the invocation instruction to the callee.

The primary reason for the invocation and return checks is that it is possible for calls to native code methods to be interspersed with calls to bytecode methods. The trace file does not contain execution information on native methods. If a native method is encountered in examining the call stack of a thread, then all objects passing across the call interface are marked as escaping. This level of conservatism insures that the results will always be correct if not precise.

Once the CDG had been updated for the instruction, the simulated execution engine proceeds to process the next line.

B.2 Thread States

The thread states used in the simulator can be viewed as a database of the known currently active threads based on the trace file being simulated. A thread becomes active when its thread ID is first encountered during the processing of trace lines by the simulated execution engine. A thread is only considered no longer active when the entire trace of the given benchmark has been processed. The thread states are stored and accessed via their unique thread IDs. This is the entry in the first column of the execution trace line shown in Figure B.2.

When a new thread state is created, it is initialized with a set of state information unique to that thread. Figure B.4 shows the fields contained in each active thread state. For each thread, the state stores a reference to the bytecode representation of the method it is currently executing. Since thread states are only created when a trace file execution line is first encountered for the thread, it will always be initialized to a reference to the executing method. The only time this entry will not be set is if the thread state has an empty call stack and the thread encounters a return bytecode statement from its executing method. Even if this event occurs, the thread state will remain in the thread state database until the entire trace for the benchmark has been processed. When the thread is first encountered, a copy is made of the CDG table for the method it is executing. A reference is then placed in the thread state to this CDG table. The bytecode line number within the current method is then recorded in this field of the thread state shown in Figure B.4.

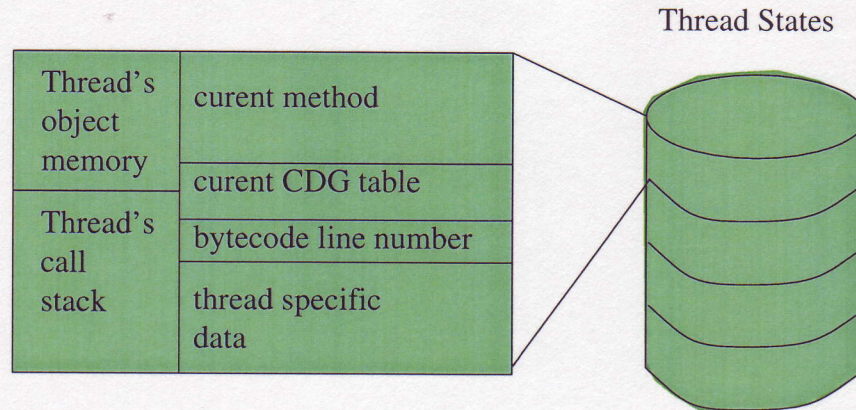


Figure B.4 The format of the thread state entries.

As the thread executes within the simulated execution engine, it can encounter allocation bytecode instructions. When such an event occurs, the simulated execution engine places a request to the memory manager for a new memory location. A reference to this new location is stored in the corresponding entry for the current CDG table for the thread as well as a reference added to the thread's view of main memory. These references are folded into the CDGs for this thread along any invocation interfaces.

In addition to allocation bytecode instructions, the simulated execution engine can encounter invocation instruction. When this occurs, a call stack entry is created and pushed onto the bottom of the stack used to hold the thread's call stack. A reference to this call stack is contained in the thread's state as shown in Figure B.4. The fields in the call stack entry are shown in Figure B.5. The first of these, the caller signature, is the trace file invocation line for the method currently being executed. A reference to the method's current CDG table is also added to the call stack entry along with the bytecode

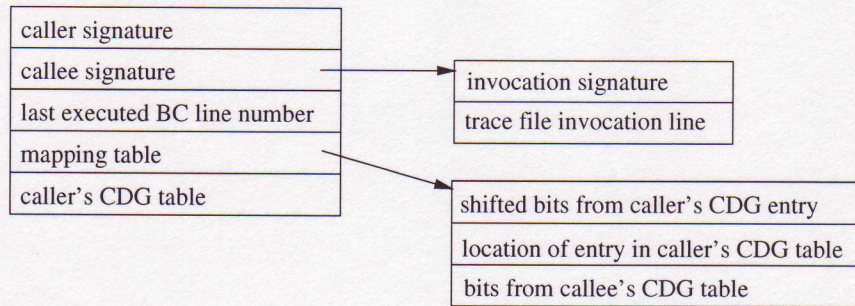


Figure B.5 The format of the call stack entry.

line number within the method that the invocation instruction was encountered at. Next, the invocation signature is created by the simulated execution engine as detailed above. This is stored in the top field of the callee signature. Finally, a mapping table is created to facilitate the mapping of the caller parameters to the callee formals.

The mapping table contains an entry for each parameter slot identified by the callee's method signature. For each one of the parameters passed, information is stored in this table. The fields for each entry are also shown in Figure B.5. Although not an exact representation, the field shifted bits from the caller's CDG entry correspond to the highlighted section of the example interface shown in Figure 4.10. Note that this field also holds any state information being transferred across the interface. The next field is an index into the caller's CDG for the parameter being passed. This allows for swift folding of any changes from a result of the invocation back into the caller's table upon return. Note that since the parameter numbers correspond directly to their location within the mapping table and these locations match exactly with their corresponding locations within the callee's CDG, no index is needed for the forward mapping. If the

entry in the caller's CDG shows that it expects a return value back, then a reference is created to hold the return value. The return value entry contains the same fields as a parameter entry but is not contained in the mapping table since it does not directly map to a known callee table entry.

The final field shown in Figure B.4 is the thread specific data. This field actually holds several pieces of information. For example, when the simulated execution engine simulates an execution line, it changes the bytecode line number field in the thread state. However, it may be necessary to also know the previous bytecode line number executed by the thread. Therefore, it records this information in a field marked in the thread specific data. Additionally, when the execution line in the trace file corresponds to a return statement, the simulated execution engine will want to know that the previous instruction executed was a return. This is important for when the next line executed by the given thread is encountered. Since the simulated execution engine will need to pop the call stack and fold back in CDG table results, it needs to know that a call returned.

The thread specific data field is also used to track any thread specific analysis information. For example, if we are tracking exactly how many bytecode lines each thread executes, then this information is stored here. It can also track items such as how many bytecode instructions were executed since the last time this thread executed.

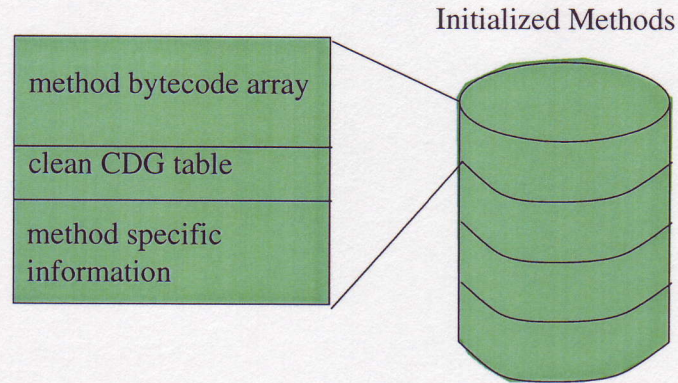


Figure B.6 The format of the initialed methods used by the method invoker.

B.3 Method Invoker

The method invoker is called by the simulated execution engine anytime an invocation line is read from the trace file. An invocation line corresponds to the first line in the sample trace file shown in Figure B.2. The method invoker keeps a table of all methods it has initialized. Figure B.6 shows an abstract view of this table. Entries in the table are accessed via the invocation line read from the trace file. This mapping is a one-to-one mapping since we have modified the trace output to include the full method signature. To illustrate this, we have broken the invocation line from Figure B.2 into its constituent parts.

Table B.1 contains the labeled version of the invocation line from the trace file. The first field in an invocation line is the type of the method. There are only two types, virtual and static. A virtual method is located via a method table pointer used with the object instance it is called with, while a static method is located via a class file.

Table B.1 Sample line showing fields in the invocation line from a trace file.

virtual	jobject	spec.benchmarks._205_raytrace.Face.GetVert	
Type of method	Return Type	Exact path location of the class file	Method name

(jint)	(I)Lspec/benchmarks/_205_raytrace/Point;
Parms	Actual method descriptor

This is not critical for the simulator since we do not distinguish between the two types in our method invoker. The next field is the trace mechanism's notation for the return type from the method. Note that for this method, the return type is `jobject`. The trace mechanism uses this return type for any reference type returned from a method regardless of whether its an object or an array. This is one of the reasons we found it necessary to include the actual method descriptor in the trace output. This actual method descriptor is shown as the last item in the labeled invocation line of Table B.1. The return type is specified at the end and is given as an object of type `Point` with the full path information. Since the invocation line is from a trace file, it has already been resolved to the exact version of the method invoked. Therefore, even though this method is virtual, we know it is the version located in the `Face` class file which can be found in the directory `spec/benchmarks/_205_raytrace`. The name of the actual method invoked follows the class file name. For this example, the method is `GetVert`. This name, plus the actual descriptor, are what is used to form the method signature.

This signature is used to make certain that an invocation or return on a thread's call stack is indeed correct. The method name is followed in the invocation line by the formal parameters to the method. This parameter listing uses the same ambiguous `jobject` for any reference types, and therefore we ignore it when resolving methods. Finally, the invocation contains the actual method descriptor for the method invoked during the execution of this benchmark. This is obtained as the program is executing and output by the trace mechanism. Note that the invocation line will be exactly the same every time we invoke or execute part of this exact method. Therefore, we use the full invocation line to access the initialized method in the table.

An initialized method consists of several fields. The first of these points to the byte array for the bytecode for the method. This byte array is used by the simulated execution engine. The next field points to a clean copy of the CDG table. This copy is what was read in from the annotated class file. It does not contain any changes. This clean copy is needed every time the method is invoked. It is copied into the thread's state and then any information is folded into the copied version. Finally, the initialized method can contain pointers to other items that may be needed by the simulated execution engine. For the simulation we ran, this included a clean copy of the OCG for the method and a flag indicating whether or not the method contained an allocation.

There are times when the method invoker will not have an initialized copy of the method requested in its table. When this occurs, the method invoker places a request to the class loader for the method. The class loader will locate the requested class file

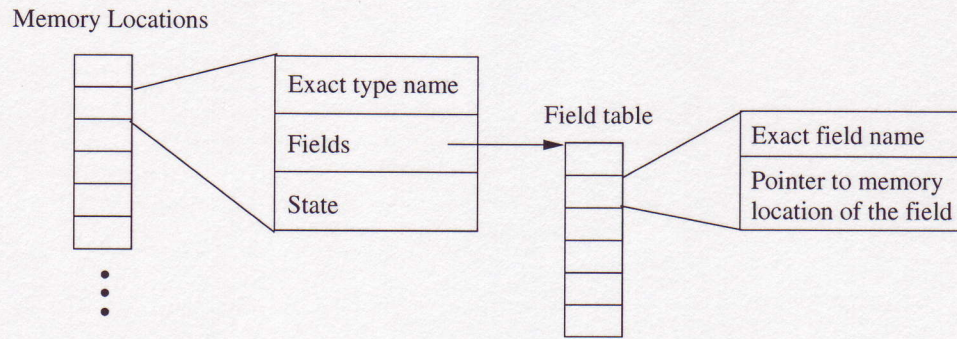


Figure B.7 Simulator memory format.

and then the requested method and return it to the method invoker. In this way, only methods actually invoked go through the added overhead of initialization.

B.4 Memory Manager

The simulator keeps track of the unique memory locations used during the trace. This tracking of memory locations is done primarily by the memory manager. Figure B.7 shows the format of the simulated memory. The memory locations are represented as entries in a growable array. Each entry has the exact type name for the entry. When an allocation request is received from the simulated execution engine, the memory manager creates a new entry in the array and a return reference to it. When a field write occurs, the simulated execution engine notifies the memory manager with the reference to the memory location the field belongs to, an identifier for the field, and a reference to the memory location being stored in the field. At that point, the memory manager uses the unique field name passed by the simulated execution engine to access the field in the field

array. It then changes the field pointer to point to the newly specified memory location. If an entry does not exist in the field array, one is created.

The memory format shown in Figure B.7 also contains an entry for state. For the simulation we ran, we used three state values: one indicated whether the memory location was method escaping via OCG analysis, another indicated whether the memory location actually escaped during execution, and the final state was used during garbage collection.

The garbage collector used by the memory manager is a mark and sweep collector. At the time a garbage collection epic occurs, the collector places a request to each thread to mark as “live” any memory locations it knows about. The thread manager then goes through the current CDG and all call stack CDGs and marks all references within them live. When all threads have marked their references live, the memory manager then traverses the memory locations to mark any location reachable via a field link from a live location also live. This process is iterative and completes when no further updates can be made. The garbage collection uses the following steps.

1. Starting at the first location in the memory array,
 - If the location is live, check memory locations reachable via its field array.
 - If field reachable location is not live, mark it live and set *changed* state in garbage collector.
2. When end of memory array is reached,
 - If changed state is set clear it and start again at top of array.
 - If changed it is not set, collect any memory location not marked as live.

For the simulation we ran, the collected memory locations were then analyzed and the results recorded.

REFERENCES

- [1] J. Gosling and H. McGilton, "The Java language overview: A white paper," 1995, <http://java.sun.com/whitePapers/>.
- [2] R. Chatterjee, B. G. Ryder, and W. A. Landi, "Relevant context inference," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999, pp. 133–146.
- [3] M. Burke and L. Torczon, "Interprocedural optimization: Eliminating unnecessary recompilation," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 3, pp. 367–399, 1993.
- [4] J. Yur, B. G. Ryder, and W. A. Landi, "An incremental flow- and context-sensitive pointer aliasing analysis," in *Proceedings of the 21st International Conference on Software Engineering*, 1999, pp. 442–451.
- [5] E. Duesterwald, R. Gupta, and M. L. Soffa, "A practical framework for demand-driven interprocedural data flow analysis," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 6, pp. 992–1030, 1997.
- [6] T. Reps and G. Rosay, "Precise interprocedural chopping," in *Proceedings of the Third ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 1995, pp. 41–52.
- [7] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994, pp. 230–241.
- [8] G. DeFouw, D. Grove, and C. Chambers, "Fast interprocedural class analysis," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998, pp. 222–236.
- [9] D. W. Goodwin, "Interprocedural dataflow analysis in an executable optimizer," in *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997, pp. 122–133.
- [10] D. Grove, J. Dean, C. Garrett, and C. Chambers, "Profile-guided receiver class prediction," in *Proceedings of the Tenth Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1995, pp. 108–123.
- [11] B. Calder and D. Grunwald, "Reducing indirect function call overhead in C++ programs," in *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994, pp. 397–408.

- [12] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ virtual function calls," in *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1996, pp. 324–341.
- [13] J. Dolby and A. Chien, "An automatic object inlining optimization and its evaluation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 345–357.
- [14] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1997, pp. 108–124.
- [15] B. Blanchet, "Escape analysis for object-oriented languages: Application to Java," in *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999, pp. 20–34.
- [16] E. Ruf, "Effective synchronization removal for Java," in *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, 2000, pp. 208–218.
- [17] J. Bogda and U. Hulzle, "Removing unnecessary synchronization in Java," in *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999, pp. 35–46.
- [18] A. Rountev, A. Milanova, and B. G. Ryder, "Points-to analysis for Java using annotated constraints," in *Proceedings of the OOPSLA '01 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001, pp. 43–55.
- [19] D. Liang, M. Pennings, and M. J. Harrold, "Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java," in *ACM SIGPLAN - SIGSOFT Workshop on on Program Analysis for Software Tools and Engineering*, 2001, pp. 73–79.
- [20] E. G. S. J. Aldrich, C. Chambers, and S. Eggers, "Static analysis for elimination unnecessary synchronization from Java programs," in *Proceedings of the Sixth International Static Analysis Symposium*, 1999, pp. 19–38.
- [21] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallie-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for Java," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000, pp. 264–280.
- [22] A. Zaks, V. Feldman, and N. Aizikowitz, "Sealed calls in Java packages," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000, pp. 83–92.

- [23] V. C. Sreedhar, M. Burke, and J.-D. Choi, "A framework for interprocedural optimization in the presence of dynamic class loading," in *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, 2000, pp. 196–207.
- [24] S. Horwitz, T. Reps, and M. Sagiv, "Demand interprocedural dataflow analysis," in *Proceedings of the third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995, pp. 104–115.
- [25] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for Java," in *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999, pp. 1–19.
- [26] J. Whaley, "Partial method compilation using dynamic profile information," in *Proceedings of the conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001, pp. 166–179.
- [27] B. Alpern, A. Cocchi, S. Fink, and D. Grove, "Efficient implementation of java interfaces: Invokeinterface considered harmless," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001, pp. 108–124.
- [28] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja, "Techniques for obtaining high performance in java programs," *ACM Computing Surveys*, vol. 32, no. 3, pp. 213–240, 2000.
- [29] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed., Reading: Addison-Wesley, 1999.
- [30] M. T. Conte, A. R. Trick, J. C. Gyllenhaal, and W. W. Hwu, "Study of code reuse and sharing characteristics of Java applications," in *Proceedings of the Workshop on Workload Characterization*, 1998, pp. 27–35.
- [31] M. T. Conte, "A characterization of code reuse within Java applets and applications," M.S. thesis, University of Illinois at Urbana-Champaign, 1999.
- [32] J. Manson and W. Pugh, "Core semantics of multithreaded Java," in *ISCOPE Conference on ACM Java Grande*, 2001, pp. 29–38.
- [33] W. Pugh, "Fixing the java memory model," in *Proceedings of the ACM 1999 Conference on Java Grande*, 1999, pp. 89–98.
- [34] J.-W. Maessen and X. Shen, "Improving the Java memory model using CRF," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000, pp. 1–12.
- [35] "The Java Grande benchmark suite, v1.0," 2001, <http://www.epcc.ed.ac.uk/javagrande>.

- [36] D. Lea, *Concurrent Programming in Java: Design Principle and Patterns*, 2nd ed. Reading: Addison-Wesley, 1999.
- [37] "Java SDK and RTE 1.2 for HP-UX.," 2000, <http://www.hp.com/products1/unix/java/>.
- [38] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Reading: Addison-Wesley, 1996.
- [39] A. Diwan, K. S. McKinley, and J. E. B. Moss, "Type-based alias analysis," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998, pp. 106–117.
- [40] M. Burke, "An interval-based approach to exhaustive and incremental interprocedural data-flow analysis," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 341–395, 1990.
- [41] S. Ghemawat, K. H. Randall, and D. J. Scales, "Field analysis: getting useful and low-cost interprocedural information," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 334–344.
- [42] J. Whaley and M. Rinard, "Compositional pointer and escape analysis for Java programs," in *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999, pp. 187–206.
- [43] F. Vivien and M. Rinard, "Incrementalized pointer and escape analysis," in *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, 2001, pp. 35–46.
- [44] A. Salcianu and M. Rinard, "Pointer and escape analysis for multithreaded programs," in *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 2001, pp. 12–23.
- [45] A. L. Souter and L. L. Pollock, "Contextual def-use associations for object aggregation," in *ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001, pp. 13–19.
- [46] I. Pechtchanski and V. Sarkar, "Dynamic optimistic interprocedural analysis: A framework and an application," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001, pp. 195–210.
- [47] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, "A study of de-virtualization techniques for a Java just-in-time compiler," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000, pp. 294–310.

- [48] A. Azevedo, A. Nicolau, and J. Hummel, "Java annotation-aware just-in-time (AJIT) compilation system," in *Proceedings of the ACM 1999 Conference on Java Grande*, 1999, pp. 142–151.
- [49] C. Krintz and B. Calder, "Using annotations to reduce dynamic optimization time," in *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, 2001, pp. 156–167.
- [50] J. C. Corbett, "Using shape analysis to reduce finite-state models of concurrent Java programs," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, no. 1, pp. 51–93, 2000.
- [51] G. Naumovich, G. S. Avrunin, and L. A. Clarke, "Data flow analysis for checking properties of concurrent Java programs," in *Proceedings of the 21st International Conference on Software Engineering*, 1999, pp. 399–410.
- [52] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani, "A dynamic optimization framework for a Java just-in-time compiler," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001, pp. 180–195.
- [53] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeno dynamic optimizing compiler for Java," in *Proceedings of the ACM Conference on Java Grande*, 1999, pp. 129–141.
- [54] A. Aggarwal and K. H. Randall, "Related field analysis," in *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, 2001, pp. 214–220.
- [55] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and precise modeling of exceptions for the analysis of Java programs," in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 1999, pp. 21–31.
- [56] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi, "Marmot: an optimizing compiler for Java," *Software-Practice and Experience*, vol. 30, no. 3, pp. 199–232, 2000.
- [57] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, "An architectural framework for runtime optimization," *IEEE Transactions on Computers*, vol. 50, pp. 567–589, June 2001.
- [58] M. C. Merten, "Run-time optimization architecture," Ph.D. dissertation, University of Illinois, Urbana, IL, 2002.
- [59] "Spec JVM98 benchmarks.," 1998, <http://www.spec.org/osg/jvm98/>.

- [60] C.-H. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W. W. Hwu, "A study of the cache and branch performance issues with running java on current hardware platforms," in *Proceedings of COMPCON*, 1997, pp. 211–216.
- [61] S. Muchnick, *Advanced Compiler Design & Implementation*. San Francisco:Morgan Kaufmann, 1997.

VITA

Marie Therese Conte was born on September 22, 1956 in Upper Darby, Pennsylvania. In 1989 she enrolled at the University of Delaware as a continuing education student. In 1991, she was matriculated at the University of Delaware as a full time electrical engineering student. As an undergraduate student, she headed the student chapter of IEEE for two years, served as an ambassador for the college of engineering, served as cataloger for Tau Beta Pi honor society, and served as bridge correspondent for Eta Kappa Nu honor society. In addition, she completed a dual concentration, in both material and devices and computer engineering, and obtaining a minor in computer science. Her GPA was high enough to qualify to become an engineering scholar taking part in undergraduate research under the direction of Professor David Mills. She chose to conclude her undergraduate research by writing and successfully defending an undergraduate thesis entitled "Indoor Wireless Networks: An Exploration of Some of the Engineering Issues Involved." This then qualified her to receive a degree with distinction. In addition, she won numerous awards including the IEEE Student Activities Award, the Department of Electrical Engineering Alumni Award, the Department of Electrical Engineering Faculty Award, the Zonta International Women in Engineering Student Excellence Award, and Golden Key Honor Society Outstanding Junior Student Award. She graduated cum laude in 1995 obtaining an bachelor's of electrical engineering. She then pursued graduate studies at the University of Illinois at Urbana-Champaign. While attending graduate

school, she worked for three semesters as a teaching assistant, receiving a nomination for the Olesen award. At the same time, she conducted research under the guidance of Professor Wen-mei Hwu, receiving her masters of science degree in 1999. She also received the Intel Fellowship and the Mavis Fellowship. In addition, she spent several summers working for Intel and Hewlett Packard, both in Oregon and California. Upon completion of her Ph.D., she will begin full time employment with the Intel corporation in Hillsboro, OR.