Diplomarbeit

Type Analysis in a Java Virtual Machine

ausgeführt am

Institut für Computersprachen
Arbeitsbereich für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter der Anleitung von

ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

durch

Carolyn Oates
Kuefsteinstr. 30
3107 St.Pölten

Oktober 2008

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbststandig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Wien, am 20. Oktober 2008

Carolyn Oates

Kurzfassung

Obwohl virtuelle Methoden in objektorientierten Programmiersprachen beim Aufruf an viele unterschiedliche Definitionen gebunden werden können, wird oft nur eine Methodendefinition wirklich verwendet. Der Einsatz von Analysealgorithmen zur Bestimmung des statischen Typs in einer Java Virtual Machine (JVM), um konservativ die erreichbaren Klassen und Methoden abzuschätzen, kann einem Übersetzer mehr Möglichkeiten zur Optimierung geben. Das Laden von Klassen bei Bedarf in der JVM hat dynamische Analyse mit Profiling zur Laufzeit gefordert.

Die Typ-Analyse Algorithmen CHA, RTA, XTA, und VTA wurden in CACAO verglichen. CACAO ist eine Forschungs-JVM, die an der Technischen Universität Wien entwickelt wird. Drei Modi wurden getestet: statische Analyse (SA), dynamische Analyse (DA), und eine neue hybride Analyse (HPA). Alle Algorithmen bauen ihre Klassendatenflussdiagramme (CFGs) dynamisch während der Analyse auf. Klassen werden nach Bedarf geladen. Daher müssen alle Analysen das Einfügen von zusätzliche Klassen in ihre Klassenhierarchien behandeln. Die Notwendigkeit von Devirtualization für Inlining und die Implementierung von Inlining in CACAO wird diskutiert.

Je weniger überflüssige Klassen geladen sind, desto besser ist die Genauigkeit der Analysen und desto kürzer die Laufzeit der Analysealgorithmen. VTA ist am genausten mit der kleinsten Zahl unnötig geladener Klassen und erreichbarer Methoden. HPA lädt weniger Klassen und liefert genauere Ergebnisse als SA. Zwischen RTA, XTA, und VTA war es nicht entscheidbar, welcher Algorithmus am schnellsten ist, aber XTA verwendet am wenigsten Instruktionen. Höhere Kosten, um mehr Genauigkeit aus der Analyse zu holen, werden durch eine geringere Zahl erreichbarer Methoden ausgeglichen.

Abstract

Although in object-oriented programming (OOP) languages, virtual methods may resolve to multiple method definitions during runtime, often only one method definition is actually used. Use of static type analysis algorithms in the Java Virtual Machine (JVM) to conservatively estimate classes used and methods reachable can give more opportunities for optimizations like inlining. Lazy loading of classes by the JVM has encouraged dynamic analysis with profiling during an application run.

The type analysis algorithms, CHA, RTA, XTA, and VTA are compared in CACAO, a research Java Virtual Machine (JVM) developed at Technical University of Vienna, in three modes: static analysis (SA) mode, dynamic analysis (DA) mode, and a new hybrid pickup analysis (HPA) mode. All algorithms build their class flow graphs (CFGs) on-the-fly during the analysis. Classes are loaded on-demand as needed, so all modes and algorithms must handle classes inserted into their class hierarchy. The usefulness of devirtualization for use by inlining based on the implementation of inlining in CACAO is discussed.

Fewer extra classes loaded gives better precision and improved runtime. VTA is most precise with the fewest unneeded classes loaded and methods found reachable in all three analysis modes. HPA loaded fewer classes and yields more precise results than SA. Between RTA, XTA and VTA, there is no clear fastest algorithm, but XTA used the least instructions. Costs to make the analysis more precise were offset by having fewer reachable methods to analyze and vice versa for less precision and more reachable methods.

Acknowledgements

I want to thank Prof. Andreas Krall, for giving me the opportunity to do my thesis as part of CACAO project although he knew it would be part-time. I also want to thank many at Computer Language Institute for their help both with my studies (and German via labs) from the beginning this degree. And I want to thank family and friends for their encouragement and prayers especially while writing this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The challenge of static type analysis in object-oriented languages is without compiling or before compilation to accurately compute information found dynamically at runtime for use by optimizations. Runtime optimizations must have little overhead. Dynamic analysis must use the information available at a specific point during an application run to decide, if an optimization is useful enough to risk the costs of it being invalidated and backed out later in the same run.

Additionally the call graph (CG) information gathered while performing static or dynamic type analysis are useful for many optimizations. Some devirtualization client applications are: inlining ([TP00], [SHR$^+$00]); smaller library jar files ([TSL03]); garbage collection ([Hir04]); and program verifiers with test case generators ([RMR03],[ZR07]). Indirectly the CGs have been used for escape analysis and side effect analysis.

1.1 Static and Dynamic Type Binding

Programming languages bind a variable with its type attributes. Statically typed languages require the program to declare the types of all variables. Dynamically typed languages can determine a variable's type from the program's context at runtime, however this takes more runtime. If the compiler's optimizer can determine that a variable only has one type, the type can be statically bound to the variable before runtime even though the language is dynamically typed.

1.1.1 Polymorphic

In modern object-oriented programming languages (OOPL), like Java, the declared type is a class with attribute fields and methods directly related to the class. A class can be extended to have more methods in a subclass. And a sub-class can either inherit or override its super classes' methods depending on what the subclass requires. The class cone (fig. 1.1) includes the class and all its sub-classes [DGC95]. A variable may be assigned any class type within its class cone.

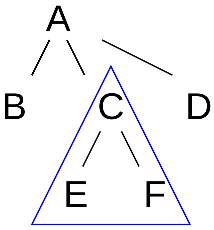


Figure 1.1: $\widehat{C}$, class cone of C

Interface classes define a common way to define methods to do a specific set of tasks. A class using an interface must define all the interface methods in addition to any methods the class defines, so the interface is complete for its task. In Java if a variable is declared to be an interface class, then the variable may resolve to any class type, which uses the interface class.

Variables in OOPLs are considered polymorphic, since they can be dynamically assigned to a variable typed to any class in their class cone. Class types using an interface class may not even be in the same class hierarchy, an interface method call may even resolve to methods in different class cones. Since methods are associated with a variable, method calls are actually virtual calls and also considered **polymorphic**. Resolving method calls at run-time due to OOPLs being dynamically typed also costs runtime.

Java allows variables, methods, and classes to be explicitly declared to one static class, which can be called monomorphic.

1.1.2 Dynamically monomorphic

If a virtual method call will always resolve to the same method definition, then the method could be devirtualized and statically bound once at runtime. This may allow other optimizations, which polymorphic method calls rule out, to be applied. Typical clients of the analysis in a JVM are: inlining, dynamic dispatch, side-effect analysis, def-use and escape analysis ([Hir04]; [RK02], [LAM07], [DRS07]; [MRR05]; [MRR05]; [ZR07]). It has also been used for garbage collection. Related clients of the analysis outside the JVM are application extractors and testing (verification). A number of static type analysis algorithms have been developed to conservatively determine such dynamically monomorphic method invocations.

Although a Java method may be invoked via virtual or interface calls, a program run may actually only use one method definition.

Object design fits two styles: similar environment, similar objects. They either are designed so the environment can change, such as printer hardware used, or so the similar objects can be used in different forms, such as shapes used in a geometry program. Usually only one environment is used at a time, so although the method calls are virtual or interface calls, they are often **dynamically monomorphic**. Similar objects of different forms are used, then the method calls are usually truly polymorphic. (ex: *x.area()*, where *x* is a *Square* and later a *Circle*). The greater challenge is when an application's design requires it to use classes with both design styles.

1.2 Java Virtual Machine (JVM)

A Java compiler creates a class file with machine independent bytecode for each class. The Java Virtual Machine (JVM) loads the classes the program needs, and creates executable code. Various kinds of JVMs exist:

- **online** JVM compilers create and run the executable code every time.
    - **bytecode JVM interpreter**, which translates and executes bytecode, line by line;

  - a **Just-In-Time, JIT, JVM compiler** which compiles methods as needed and immediately executes them;
  - **hot-spot JVM**, which is a mixture between an interpreter and a JIT. A profiler identifies hot-spot methods, which are executed often and compiles hotspot methods, while continuing interpretation until the compilation is finished.

- **offline** or **ahead-of-time** JVM compiles bytecode once to binary code, which may be run multiple times.

The optimizer for an a compiler may use optimizations that use more runtime than an online JVM compiler, where the user is waiting for immediate results. Sometimes optimizations for online JVMs are performed as an intermediate pre-processing step for online JVMs. Whether static or dynamic, type analysis results collected for optimizations must result in more runtime savings than extra analysis overhead. The Static Analysis algorithms tested were implemented in CACAO, the an open source research JVM developed at the TU Vienna [Kra98].

1.3 Static, Dynamic, Hybrid Type Analysis

Type analysis is used to determine dynamically monomorphic method invocations by finding all classes a method call may resolve to. Results of bytecode analysis may be used by tools, such as Eclipse and test case generators before invocation of a JVM or by the JVM for optimizations such as devirtualization for inlining.

Static Type Analysis (SA), (sec.5.1), or offline analysis gathers information before any executable code is produced. SA examines all possible reachable methods as defined by the analysis algorithm. The SA can gather information for multiple paths through the code or must be supplied with information from a previous run(s). Before run the exact path through the code is not known. SA is challenged to find only the paths through the code that are truly reachable.

Hendren et.al. point out in [QH04] that ”dynamic class hierarchy information can be used to build a conservative CG at runtime. However, it is desirable to have a more precise CG for most interprocedural analyses.” ***Dynamic Type Analysis (DA)***, (sec.5.1), or online analysis is performed during the run of the JVM. Only one run is analyzed. Only the information available at a specific point in the run is known, so analysis is not completed until the run is finished. DA is challenged to know when there is enough information to for a valid analysis.

Hybrid Analysis, HPA ([Ern03]) or *Blended Analysis* ([DRS07]) uses SA and DA in tandem, where one analysis is used as a pre-analysis and the other as post-processing of the first analysis. ***Hybrid Pickup Analysis (HPA)***, (sec.5.1) as implemented here combines SA and DA into one application run in the JVM as explained in detail in Chapter 5. HPA combines three runs into one run.

An analysis may only look at the types within a method, intra-procedural: or inter-procedural (IPTA), which also look at types between methods for the whole program. SAs and DAs discussed here are both *inter-procedural analysis, IPTA*, which look at type information. IPTA will be used in this work to refer to an analysis, where it is not important for the statement whether it is SA or DA or possible to use for both.

The *ideal* type analysis algorithm would quickly analyze only the exact same code reachable dynamically by the program run. Common characteristics found among IPTA algorithms can be used and compared to find, which benefits certain algorithm traits bring.

SA is appropriate when all paths through the program should be analyzed and runtime is not critical. DA requires a quick analysis, but only needs to analyze the actual path through the program.

1.4 Whole program Problems

Whole program analysis for Java, whether static or dynamic, is a challenge, since the Java program is formed at runtime from available class files as needed by the bytecode and by using dynamic language features of Java. IPTA analyzes information from the bytecode.

But methods and classes used through the *Java Native Interface (JNI)* are not available through bytecode. Java also allows classes to be dynamically loaded through input at runtime. *Reflection* allows information about a class to be asked, which indirectly causes this class to be loaded. The whole program is not known before runtime nor by following the program flow through the code. Program entry points must be provided or captured.

1.5 Classes Loaded

How many and when classes are loaded affects Java program analysis, since it affects if a single or multiple method definitions are available. Static analysis must load as many classes as are needed to determine all possible reachable methods during runtime. Dynamic analysis must work with the classes the application run currently has loaded. However DA must handle the case where a class with it method definitions are inserted into the class hierarchy.

1.6 Notation

Italics is used for definitions, terms and code. **Bold** is used for emphasis to help the reader find an important term or point easier.

A wide hat like $\widehat{C}$ indicates the class cone of the class or classes. When $C.m$ is used, C is the declared type of the variable invoking the method, m. The method name, m, implies the method's unique signature, which includes its class, name, parameters and return type. Methods in the *main* method's class are referenced without reference to their class.

Variables are referred to with v as a subscript of what the variable is associated with. For example: $C_1.m_{1.v.from}$ refers to a variable,v, in method $C_1.m_1$. The $v.from$ indicates $from$ is a part of the variable.

1.7 Overview

The remainder of this work is organized as follows: Chapter 2 gives an overview of important related theory and well-known Static and Dynamic Analysis algorithms. The algorithms compared are explained in section 2.6.1. To complete the background information needed, the chapter ends with an overview of the Java bytecodes used by IPTAs. Chapter 3 summarizes work related to applications of IPTA algorithms in Java.

Chapter 4 uses a motivating test case with virtual method calls to demonstrate when specific tests and algorithms (simple tests, CHA, RTA, XTA, VTA, Andersen Points-To) can determine when a method is (dynamically) monomorphic. The effect of using the class load scheme on the algorithms is also demonstrated via the test case. Chapter 5 describes the design decisions, the implementation and testing in CACAO of the three modes, SA, HPA, and DA, and the algorithms RTA, XTA and VTA. Chapter 6 reports the results. Chapter 7 summarizes conclusions from the results compared to other related work. Appendix A contains a pseudo code overview comparison of the algorithms implemented (CHA, RTA, XTA, VTA). Rather than duplicating information in the same place Appendix B contains some tables for results reported as only as graphs in Chapter 6.

Chapter 2

Type analysis algorithms

This chapter gives an overview of requirements which all *Inter-procedural Type Analysis (IPTA)* must fulfill to be usable. IPTA characteristics are explained along with algorithms having the characteristic described. The chapter ends with an explanation of the Java bytecodes important for IPTA of Java bytecode.

Structures analyzed by IPTA algorithms vary from a simple class hierarchy and the program's call graph (CG), to call flow graphs (CFG) with context information on individual expressions. Although simpler IPTA algorithms were tested, the more detailed IPTAs use a simpler IPTA's call graph (CG) as a starting point or to prune their CG. A more precise CG makes static analysis (SA) faster and makes decisions based on dynamic analysis (DA) more exact.

The best acceptable solution for a data flow analysis algorithms lies between the largest safe solution, *meet over all points (MOP)*, and the smallest sound solution, *maximum fixed point(MFP)*. The MOP and MFP are equal for the ideal solution. IPTA algorithms like RTA, XTA and VTA are compared to see which one is the MFP of those tested. CHA will be used as the MOP for testing. Having an overview of additional IPTAs helps order where the algorithms tested fit in overall.

The best acceptable solution for a data flow analysis algorithms lies between the largest safe solution, which are equal for the ideal solution. IPTA algorithms like RTA, XTA and VTA are compared to see which one is the MFP of those tested. CHA will be used as the MOP for testing. Having an overview of additional IPTAs helps order where the algorithms tested fit in overall.

Many SA algorithms were designed before Java and later adapted and tested with Java. Type and reference analysis algorithms used for IPTAs are rooted in basic data flow techniques ([Kil73]) and reachability ([Sri92]).

SA and DA have many of the same characteristics and properties, as Ernst in [Ern03] points out. He encourages researching using SA algorithms for DA and vice versa, as well as Hybrid analysis, using combinations of both. The applications of SA and DA differ. SA is used with Java for program extraction, verification and testing. DA is used for online compiler optimizations.

This chapter ends with an overview of the Java bytecodes used by IPTAs and referenced throughout the rest of this thesis.

2.1 Algorithm requirements

The following section formally defines an IPTA algorithm's goals of delivering an ideal solution with only reachable methods for the lowest cost. The ideal solution for this thesis is explained. Factors that affect IPTA characteristics are explained. It ends with descriptions of better known SA algorithms that have been studied previously.

2.1.1 The ideal solution

The ideal solution ([Bin07]) can formally be described as:

- *sound*, delivering the expected results;
- *complete*, including all the information expected and needed;
- and *precise* or *exact*, containing no extra classes or methods.

In short the ideal solution is exact, having neither too little nor too much.

Realistically type analysis (TAs) are only an approximation of runtime information, since unknowns like program inputs can vary.

- *Pessimistic (conservative, safe)* TAs may include and analyze extra information, but are sound and complete. Algorithms prune out unneeded information to come closer to the ideal solution.
- *Ideal* TAs have the same exact solution as the program run.
- *Optimistic (incomplete, unsafe)* TAs approximate and miss information. Algorithms add to the information to get closer to the ideal solution.

  An algorithm may start optimistic, but add too much information to achieve a safe (pessimistic) end solution.

IPTAs are evaluated against the *ideal* solution definition. Decisions made on DA results during runtime may be made on an incomplete solution, since information is not complete until the end of the program run.

SA	sound	pessimistic (safe)	complete	all code paths	imprecise (too much)
DA	unsound	optimistic (unsafe)	incomplete until program ends	1 code path	precise (exact)

Table 2.1: Comparison of SA vs. DA

This thesis considered solutions usable to identify dynamically monomorphic methods as candidates to be inlined. Inlining requires a sound solution, where it is known if a method is really polymorphic at runtime. Complete is desired, incompleteness must be recognized. If a method is analyzed as dynamically monomorphic and inlined, but found to be actually polymorphic during runtime, the optimization must be corrected. Precision is desired, but if too few methods are inlined, then an optimization was missed, but no corrections are required. So a pessimistic solution is required for the MFP when the inlined code is executed.

Ideally exactly all dynamically monomorphic methods that exist should be found and resolved by the IPTA algorithm.

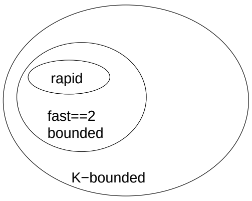


Figure 2.1: Algorithm classifications by iterations cf. [Ryd07] Lecture 1 pp.15-16

2.1.2 Reachable methods

Resolving virtual method calls are the SA algorithm's challenge. Based on some program entity, like a method, instantiation, or expression, the IPTA algorithm determines if a statement, and subsequently a method call, is ***reachable***. This affects how precise a SA algorithm is. A ***worklist solution*** ([CHK02], [Ryd07]) is often used. Methods are taken from a *method worklist (mWL)* and analyzed.

This mWL needs a starting point for its CFG. In Java the most inexact, but safe SA would initialize its mWL CFG pessimistically to analyze all methods in all available class files [1] and prune out unreachable methods. Alternatively and more exact in the end, it may start optimistically with the *main* method and ***on-the-fly*** work to a sound CFG by adding reachable methods to be analyzed to the method mWL. Besides *main*, Java has entry point methods determined by the JVM implementation.

The IPTA gathers information used for optimizations like devirtualization. Post-processing may be used to show the whole program CFG. A DA algorithm analyses only methods actually invoked during runtime. DA builds a CFG only when an optimization or post processing analysis requires it.

2.1.3 Costs

IPTA algorithms have 2 major costs: **time** and **memory**. Runtime can be measured for specific benchmarks for particular implementation. The time needed by algorithms can also be estimated by various characteristics of the algorithm, such as the number of iterations through the code.

Time costs for IPTA algorithms can be estimated by how many iterations over the code are needed ([FKU75], [KU76]) to converge to the smallest acceptable solution [2] Ryder in [Ryd07] summarizes the convergence costs for data flow algorithms for reference analysis as:

- *K-bounded*: The analysis converges in K iterations to a solution.
- *Fast*: One pass around a cycle is enough to summarize its contribution to the data-flow solution but there may be data-flow between nodes.
- *Rapid*: One pass around the cycle is enough, since there is no flow of data between nodes.

Using the number of iterations a SA algorithm for ordering allows the SA algorithms can be placed into a partially order set (POSET). This allows SA algorithms to be compared mathematically using a lattice as presented in ([GDDC97] and [GC01]). Acceptable algorithms can be chosen from a lattice knowing the

[1] *All* available classes are usually classes in a classfile's constant pool.

[2] [Ryd07] and [KS92] explain further.

usable range of solutions lies between the largest safe solution and the smallest acceptable solution.

Memory costs determine the scalability of an algorithm. The number of sets kept is a measure of the memory costs. The number of sets needed is determined by the number of details kept by the algorithm. The characteristics of the application analyzed determine how often specific details are found and stored. Using more memory incurs a secondary overhead, by causing the garbage collector to be invoked more often.

IPTA algorithms should use as little memory as possible, since they keep information for use by the optimizer. Where possible, to save space **Flags** and **Binary Decision Diagrams (BDD)** vectors are preferable to a full pointer based **Call Graph (CG)** set based algorithm.

2.1.4 Java Whole Program Challenges

Whether static or dynamic, not all classes used by a Java application can be determined through the bytecode alone due to certain Java language features ([Hir04], [TP00],[Ryd03]):

- ***Dynamic class loading*** during runtime based on program request;
- ***Java Native Interface, (JNI)*** class loading and method invocation outside the bytecode;
- ***Reflection*** is using methods, like those in the package *java.lang.reflect*, to ask for information about a class, method, or field.

Classes loaded and method invoked outside the bytecode must be found out ([Hir04]) or provided to the SA ([TP00]); or the reference analysis for Java will be neither sound, nor complete.

DA requires the JVM to be able to handle updating of the analysis as classes are lazily loaded, which may invalidate optimization decisions made as the program proceeds ([HvDDH05]).

2.1.5 Common Characteristics

Common characteristics, which affect precision and indirectly speed, among reference and type analysis algorithms are noted in [Ryd03] and [Hin01]. Combinations of these characteristics have been explored by various algorithms and are being used to design new algorithms. Some frameworks like Vortex ([GC01]) and Sable soot ([VRCG$^+$99]) have been implemented to easily use and test some characteristics alone or in combination.

Flow-sensitive algorithms follow the order of code flow (statement order). "A solution is computed for each program point ([Hin01])." A program point may be higher-level than just a statement. *Flow-insensitive* algorithms are faster since they can analyze code linearly, without regard to branch or specific returns from method calls, but not as precise.

Context-sensitive algorithms associate solutions with a specific invocation (calling) context. Enclosing calling contexts may also be considered. *Context-insensitive* algorithms group all calling contexts together, so the method is either called (i.e. *reachable*) or not called (i.e. *not reachable*). Again keeping less information is faster, but more imprecise.

Object representation An object may be represented by one abstract object for the whole program or some program construct such as a method, variable or instantiation site. Many IPTAs are set-based, keeping type (type-based) or reference (flow-based) information sets based on some attribute like a method, field, variable, or instantiation site. Some type analysis implementations use just flags, BDD with vector bit sets ([Mil07], [MRR05], [SGSB05]), or compute the information when needed ([SGSB05]) saving on the total memory needed.

2.2 Type-based Unification

IPTA algorithms keep the type information based on some program point. Insensitive algorithms unify all the types for the program to a specific program entity, such as: class hierarchy, variables, and creation site. Flow and context information is only available at the level of the unification point. If the unification entity contains no detail of the flow inside a method, the algorithm is precise enough to build a CG, but too imprecise to build a CFG. Especially for SA a CG is useful to limit an initial analysis to only the reachable methods.

2.3 Call Graph Building

A call graph is built start with the main method (or other program entry points) and adding methods determined reachable by the IPTA algorithm. This limits the analysis to only the reachable, as methods, which the algorithm determines to be reachable, rather than all methods in a class file. CG algorithms are not considered precise enough for many optimizations ([Ryd07]).

The program representation ([Ryd03]) may either start with a simple CG and update this CG via *pruning* or additions; or build the CG *lazily on-the-fly*. A simple CG may consist of all methods in the constant pool of the class file for classes referenced in the class constant pool or built by a fast, but imprecise algorithm. Research in [GC01] recommends on-the-fly because fewer methods must be analyzed. Fast algorithms can use one algorithm for the first iteration and a more precise algorithm for their follow-up pass.[3]

2.4 Field-sensitive

In Java type flow within and between methods is not only through the method's CG and CFG, but also fields. [Ryd03] explains field-sensitive as distinguishing between different fields of an abstract object.

2.5 Design Characteristics

The *needs of the client application* ([Hin01], [HDDH07]) and *how easy the information is available* drive which characteristics are favored when designing an analysis algorithm.

[3] The VTA implementation in [SHR$^+$00] used RTA to create an initial CG. However this thesis created the CG for VTA lazily on-the-fly.

[HDDH07] adds the additional analysis requirement of ease-of-use with few to no user inputs. Realistically users in a production environment will avoid a tool or optimization which requires inputs.

2.6 Interprocedural Type Analysis Algorithms

The IPTA algorithms determine which methods are reachable based on the class types the type analysis algorithm considers used. The following explains the most common IPTAs grouped by characteristics.

2.6.1 Context-Insensitive algorithms

Context-insensitive algorithms have been popular to build CGs due to their speed, but are often not a final solution, due to no flow graph, resulting in less precision. A CG is built starting with all program entry points, like *main*, and analyzing all methods reachable from the entry point.

Class Hierarchy Analysis (CHA) All classes in the loaded class hierarchy are considered used. If a method, $C.m$, is invoked from a reachable bytecode, then all method definitions of m in the $\widehat{C}$ are considered reachable ([DGC95]). CHA is context- and flow-insensitive; it uses 1 abstract reference variable per class throughout the program and it uses 1 abstract object to represent all possible instantiations of a class ([Ryd07]).

CHA used with Java is defined by the class loading scheme, for example: SA uses all classes in the class file constant pool as in the CHA example in [BMA03] and DA uses a smaller class hierarchy of all classes actually used during runtime ([QH05]). CHA performs better with a smaller class hierarchy since fewer classes usually means fewer available method definitions resulting in more monomorphic method definitions.

Static Unification Algorithms

These algorithms analyze only reachable methods. They collect or unify all class types to a specific scope within the program flow (program, method, variable, expression) (Table 2.2).

Algorithm	CHA	RTA	XTA	VTA	CFA-0	Pts-To
Program Entity	class hierarchy	instantiated classes	methods, fields	variables, fields	expressions	creation site

Table 2.2: Algorithm Node Entities

Rapid Type Analysis (RTA) – RTA only considers classes instantiated via *new* as *used* and method definitions in these classes as *reachable* ([BS96]). The algorithm requires only one pass, by marking reachable methods in unused classes, so they can be changed to *used* should the class be instantiated later. In RTA a program *uses* classes and *has a* reachable method set.

Static RTA can also be affected by the class loading scheme. If the class loading is done as a separate pass then classes instantiated in unreachable methods may be loaded. [BMA03] shows such an example.

eXtended Type Analysis (XTA) – XTA narrows the context of class information to a method. A class type set is kept for each method and field. The method class set contains classes:

- instantiated in the method via *new*;
- passed in via parameters;
- passed back via a return from method calls;
- written to static fields via a method variable or field.

The field class set contains classes:

- instantiated in via *new*
- of variables or fields written into the field.

Only called methods, whose class is in the calling method's class set are considered *reachable*. XTA is (type propagation) flow-sensitive between methods and fields, but context-insensitive, since all method invocations are grouped together ([TP00]). So a method's bytecode can still be analyzed linearly.

Variable Type Analysis, VTA, keeps a class set per field and variable. Only expressions that use reference variables (methods, *this*, parameters, local, *return*) and fields are analyzed. Circular assignments are unified in the original VTA to just one variable to speed analysis. A starting CFG can be given by another CG analysis (CHA, RTA, VTA)([SHR$^+$00]). It is flow-sensitive between methods using a type propagation graph, but context-insensitive, since all method invocations are grouped together. VTA is the first of the CFG algorithms complex enough that SA implementations start with a CFG from a simpler algorithm, rather than creating its CFG on-the-fly.

The algorithm proceeds in the following steps [BMA03]:

1. Initialize variable nodes;
2. Initialize edges between variables in the type propagation graph.
3. Initialize types of the variable nodes;
4. Unify and collapse variable paths with circular assignments.
5. Propagate types

0-CFA (Call flow analysis) – In 0-CFA classes used by the current expression are considered. So type sets are kept for each expression. Type propagation is between expressions.

Points-to style analysis

Points-to style analysis keeps class type information about per object instantiations (creation site), either as a group by class or single creation site. Sets of references to the allocation sites are kept.

Steenguaard style Points-to *(unification, equality, symmetric)* –
([Ste96]) algorithm uses unification of class type information for assignments. It is good for circular assignments, otherwise flow information is lost and the type set is too large.
$PtsTo(A) == PtsTo(B)$ after an assignment of $A = B$.
If before: $PtsTo(A) = \{A\}$ and $PtsTo(B) = \{B\}$,
then after: $PtsTo(A) = \{A, B\} = PtsTo(A) \cup PtsTo(B) = PtsTo(B)$.

Andersen style Points-to algorithms *(inclusion, subset, directional)* – ([And94]), where class type information flows into the assignee by assignments. $PtsTo(A) \subset PtsTo(B)$. It is flow-sensitive, but context-insensitive. After assignment $PtsTo(A)$ contains $PtsTo(B)$. So $PtsTo(A) = \{A, B\} = PtsTo(A) + PtsTo(B)$ and $PtsTo(B) = \{B\}$ remains unchanged.

A detailed example, comparing CHA, RTA, XTA, VTA and Andersen Points-To, will be explained in Chapter 4.

2.6.2 Sensitive algorithms

Flow-sensitive and context-sensitive have been considered unable to size well in the past because of multiple iterations of the code to follow all possible code flows and the memory needed to store the contexts. Use of Binary Decision Diagrams, BDD, has improved the efficiency for context-sensitivity ([LH06]). Some better known "sensitive" algorithms are:

k-CFA and k-l-CFA –([Shi91], [GC01]) keep calling contexts to a depth of k for l previous statements. RCFA is a scalable version of CFA reported as twice as precise as RTA in [Pro02].

Cartesian Product Analysis, CPA – ([AH95]) analyzes all class type combinations of a method, i.e. Cartesian product of the class types. Return type is the union of the analyzed method's class types ([KH02]). CPA is faster than k-CFA, because calling contexts with the same parameter type contexts are grouped together ([KC07]).

Object sensitive – OBA is a context- and field-sensitive analysis which uses the receiver object for the "calling" context ([MRR02], [MRR05]).

Comparing two k-CFA algorithms and Object Sensitive, Lhotak in [LH06] found that context sensitivity improved call graph precision by a small amount, improved the precision of virtual call resolution by a more significant amount and enabled a major precision improvement in cast safety analysis.

There is no one best analysis algorithm for finding dynamically monomorphic virtual method calls in Java for all optimizations. The right analysis must be used or designed for specific applications or cost goals ([Hin01]), [HDDH07]. More algorithms are now being researched in combination with the well-known IPTA "non-sensitive" algorithms ([KC07], [DRS07], [Mil07], [HDDH07]) for a balance between precision and costs for use by multiple client optimizations or program understanding applications like test and verification.

2.7 Overview of Java bytecodes

IPTAs gather the most information by analyzing the fewest bytecodes possible. This section gives an overview of which bytecodes are needed to gather type and points-to information. Although not bytecodes, constructor methods are briefly explained. A full description of JVM bytecodes is in [LY99].

2.7.1 The *new* bytecode

The Java *new* bytecode is used when an object is to be instantiated. Type analysis uses the *new* bytecode to know a class is used by the program and for creation sites. This includes arrays, since arrays use *new* to instantiate all their elements ([LY99] section 7.9 Thread example) and the analysis does not have to allocate space for the array itself.

2.7.2 Method Invocation

There are four invoke bytecodes (*invokestatic*, *invokespecial*, *invokevirtual*, *invokeinterface*) which cause a method to be called.

Non-virtual invocation

The following method invocations, grouped by bytecodes, are always monomorphic since the methods may not be overridden by subclasses:

- all *invokestatic* invoked methods, which are all methods declared *static*;
- all *invokespecial* invoked methods, which include the instantiation constructor, <*init*>, and *private* methods;
- *invokevirtual* invoked methods which are also declared *final*.

Virtual invocation

The ***invokevirtual*** bytecode is used when multiple target methods are allowed (i.e. polymorphic), based on the class of method call and its subtypes. IPTAs look at what types are used as defined by the analysis algorithm to determine if such a method call is really dynamically monomorphic.

The ***invokeinterface*** bytecode is used when a method from an interface class is invoked. Interface calls may also be polymorphic. Resolving an interface method requires looking at the classes that implement the interface. The IPTA may have to look at multiple class cones to determine, if the method invocation is dynamically monomorphic. So devirtualizing interface methods, where the interface has multiple implementing classes will have higher costs for an IPTA.

2.7.3 Constructors and *finalize*

There two kinds of constructors, class and instance. The **instance constructor** uses the name of the class in the Java code and the name <**init**> in the bytecode. When a class is instantiated via *new*; via dynamic instantiation; or because the class is a super class to an instantiated class. The instance constructor is invoked by *invokespecial*.

The **class constructor** initializes static fields for a class and is always named, <***clinit***> with no parameters. The class constructor is invoked directly from the JVM when the class is instantiated or when a static method or static field is used.

The ***finalize*** method for a class is also invoked by JVM, if the class instantiation is garbage collected.

2.7.4 Casting

The ***checkcast*** and ***instanceof*** bytecodes also reference classes and can be used by an IPTA to determine if a class can be used by the program.

2.7.5 Assignment

Type flow occurs through assignment statements, which have been translated to the appropriate Java bytecode. In Java all bytecodes, which use reference classes start with an *a*. So the *aload*s and *astore*s bytecodes indicate to an analysis flow between local variables in a method. The other x*load*s and x*store*s bytecodes use default classes like *istore* for integer store.

Local fields use ***putfield*** and ***getfield*** to write *to* and read *from* local fields respectively. Static fields use ***putstatic*** and ***getstatic*** to write *to* and read *from* static fields respectively.

*Load*s and *get*s push on to and *store*s and *put*s pop off the JVM stack.

2.7.6 Other Bytecodes

Other bytecodes are only used by IPTAs for stack analysis, so the correct item is pushed and popped for later use by one of the above bytecodes. The Java bytecode may be translated to 3 address code, like *add* a, b, c, such as is done for VTA by Sable Soot in [SHR$^+$00]. Java bytecode is a stack machine or 0 address code, like *load* C, *load* B, *add*, *store* C ([Gea74]).

Chapter 3

Related Works

Type information gathered using an IPTA with a CFG includes SA, DA, and Hybrid analysis. SA and DA algorithms do not differ, but the results are affected by the class loading differences and information availability between SA and DA. SA typically looks at all possible runs of a program, so for a specific application run it will have more spurious methods and give a less accurate analysis. DA looks only at one specific application run.

IPTA using CFG algorithms is useful for program understanding and a number of optimizations. [GC01] names: class analysis (side effect of IPTA), what static fields and instances are modified by a method call; exception detection; escape analysis; tree shaking of reachable methods (side effect of IPTA). Class analysis and tree shaking are actually side-effects of IPTA, which can give a Java application smaller footprint.

First SA algorithms were adapted or developed for Java, then these algorithms were further adapted for DA for a variety of client applications:

- CHA static [DGC95] then dynamic[IKY$^+$99], [QH05];
- RTA static [BS96][TP00] then dynamic [QH05] (inlining);
- XTA static[TP00] then dynamic [QH04] (inlining);
- VTA static[SHR$^+$00] then dynamic [QH05] (inlining);
- Points-To Andersen static [And94] then dynamic [HDDH07] plus many other client applications including: Garbage collection (GC) [Hir04]; Escape analysis [RK02], [LAM07], [DRS07]; Def-use, side-effect analysis [MRR05]; Def-use [MRR05];and program slicing [ZR07].

Figure 3.1 shows the basic Java process, where in this process IPTA can be applied, and examples of where IPTA has been applied based on the applications needs. Some IPTA results from test tools are of interest for optimizations, since test tools also want to know exactly what classes and fields are used and which methods are reachable by an application.

3.1 Pre-processing Analysis

Pre-processing SAs run before the JVM starts to speed-up the JVM. Their input and output are the application's classfiles for an iterating application.

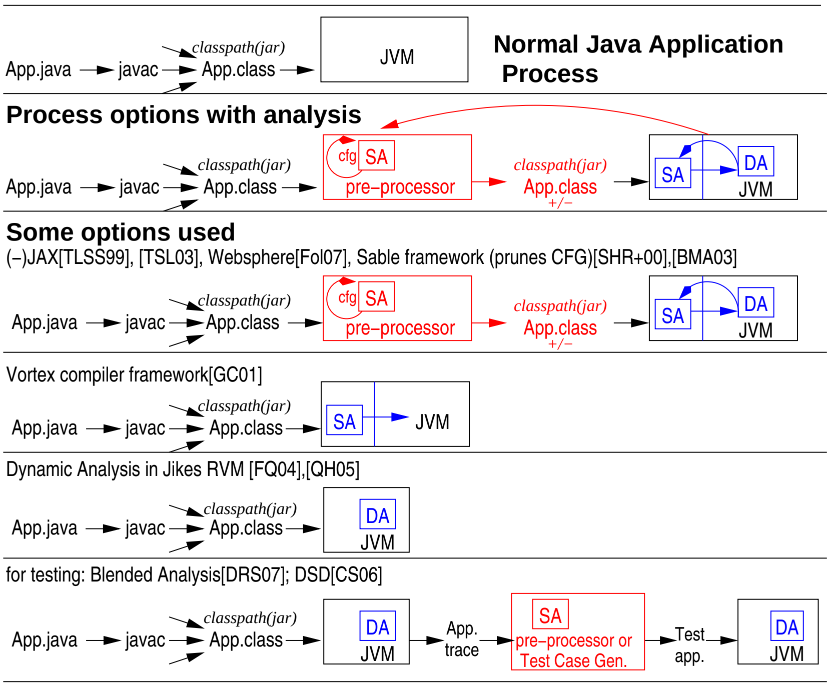

App.java
javac
classpath(jar)
App.class
JVM
Normal Java Application Process
Process options with analysis
cfg
SA
pre−processor
classpath(jar)
App.class
+/−
SA
DA
JVM
Some options used
(−)JAX[TLSS99], [TSL03], Websphere[Fol07], Sable framework (prunes CFG)[SHR+00],[BMA03]
Vortex compiler framework[GC01]
Dynamic Analysis in Jikes RVM [FQ04],[QH05]
for testing: Blended Analysis[DRS07]; DSD[CS06]
App. trace
pre−processor or Test Case Gen.
Test app.

Figure 3.1: Overview where IPTA occurs in Java process

IBM Jikes Application eXtractor (JAX) ([TLSS99], [TSL03]), and Sable Soot framework [SHR$^+$00] use pre-processing IPTA to reduce the applications size and to perform static optimizations to the bytecode. Limitations of preprocessor optimizations are briefly discussed. [KVM00] uses pre-processing analysis to perform class verification before the runtime.

3.1.1 Smaller application footprint

Besides optimizing the bytecode itself, unused classes are removed out of the *classpath* or *jar* file and unreferenced methods and fields removed out of the classfiles. A smaller application footprint:

- reduces the class hierarchy ([TSL03]);
- lower class loading time ([Fol07], [LAM07]);
- allows a Java application to fit into an embedded system ([Fol07]);
- reduces time to pass a Java application over networks ([TLSS99]).

The result must be sound and complete, so the classes, methods and fields left will be a superset of what any program runtime needs. The more precise the algorithm used by the preprocessor, the smaller a *classpath jar* can be.

Unused interface methods must be implemented, so JAX replaces the method code with a *return*.

Libraries for Java applications contain more class files and methods, than used by any one program. Static fields are often used for constants, all of which are not used by one program. JAX reduced the class libraries needed with SPECjvm98 to 37.5% of their original size using XTA [TSL03]. The HyperJ benchmark program, which heavily uses interfaces, was reduced to around a fourth of its original size (27.8%). The major size reduction (53.3%) came from IPTA reachability analysis. Bytecode optimization (dead code removal, inlining, devirtualization, class hierarchy compression, and name compression) accounted for remaining the 18.9% of the application size reduction.

3.1.2 Bytecode Optimizing

Not all bytecode optimizations can be performed statically. A preprocessor can only communicate with the JVM through its classfile, so some optimizations, like inline opportunities, are communicated indirectly by changing attributes or the bytecodes used.

To communicate inlining opportunities to the JVM, JAX changed methods attributes to *final* when a method was never overwritten in its class hierarchy. Also where legal, *invokevirtual* was changed to *invokespecial*, if a method was determined to be dynamically monomorphic. Where not legal the analysis results are lost or must be reanalyzed at runtime.

3.1.3 User-defined Classfile attributes

Additionally [LY99] section 4.7.1 allows adding user defined classfile attributes. For mobile devices, the CDLC Hotspot pre-verifier adds a stack map attribute when performing class verification before delivering an application. This attribute tells the CDLC Hotspot the results [KVM00]. However classfiles grew by 5%.

3.1.4 Algorithm comparisons

CHA is used as the standard largest solution for context-insensitive algorithms and 0-CFA as the smallest solution for $O(n^3)$ algorithms [GC01]. [TLSS99] tested JAX with RTA and CHA with real Java applications such as large reservation system developed by a customer. RTA removed 18.6% more classes and 16.8% more methods than CHA.

[TP00] looked at optimistic algorithms with type collection points (class, field, method, fields and methods (XTA)) **between RTA and 0-CFA** to see which had the fewest extraneous calling edges to unreached methods. **XTA**, which considers both fields and methods, was 0.8% more precise than RTA at eliminating spurious polymorphic method calls. XTA ran up to 8.3 times slower than RTA.

[SHR$^+$00] sought to design a 1 iteration algorithm that scales linearly and improves on RTA. In Sable Soot framework ([VRCG$^+$99]), [SHR$^+$00] starts with a conservative CFG from an initial IPTA and improves the CFG with **VTA** by pruning unused methods using VTA. To create the conservative CFG on all methods in a classfile, they used, in order of measured precision, CHA, pessimistic pRTA, optimistic oRTA and VTA. VTA+VTA pruned the most class type nodes (10-65%) and call edges(17-65%). They observed an runtime improvement of 0% for *javac* and 1% for soot (written in Java) with CHA and a 1% and 3% improvement respectively with VTA.

[BMA03] presents a version of **VTA with multiple iterations and type intersection** to improve precision of the CFG even more. They found the number of fewer classes used vs. CHA was 13% for RTA, for standard VTA 23%, and VTA with the two variations 32%.

[Mil07] looked at **inter-class dependencies** (inheritance, sub-classes, fields) for a test case generator which uses Soot for its CFG. They found inter-class dependencies of 56% for standard VTA; 32% for 0-CFA; 25% for OB, points-to on method objects; and 24% for OBR, points-to data flow to resolve library call-backs. VTA used its standard *pessimistic* implementation. CHA was used for its first conservative CFG. The other algorithms created their CFGs *optimistically on-the-fly.*

Object-sensitive points-to found reductions in spurious edges of 34.2% compared to 75.1% by plain points-to.

3.1.5 Static XTA vs. VTA

Both XTA and VTA are k-iteration algorithms, since there is type flow between the nodes. XTA's first iteration builds class sets and eagerly propagated types during its pass over the program's bytecode. XTA then propagates in further iterations until nothing changes or a practical limit ([TP00]).

The original VTA ([SHR$^+$00]) has five passes over the data, however only the last pass does an *end* type propagation. [BMA03] shows multiple propagation iterations using VTA$_n$ improve the results. Dynamic VTA in [QH05] eager and batch propagation is used. Propagation is covered in more detail in section 3.3.

[TP00] states JAX was never tested with VTA because of the stack analysis required for bytecode analysis was considered too costly in the JAX architecture. VTA is designed to use a single iteration to size linearly. [SHR$^+$00] did not test XTA, but comments "the XTA solver may require iterations".

The SA implementations of JAX's best reported algorithm, XTA, and Sable's best reported algorithm, VTA, are not directly comparable, since:

Analysis is performed in different phases. XTA performed its analysis directly on bytecode (before any stack analysis). VTA was performed on Jimble intermediate code (after stack analysis with variable labels).

XTA produced its CFG on-the-fly and VTA pruned its CFG. XTA starting CFG added reachable methods toward a sound solution. VTA started with a sound CFG and pruned out as many unreachable spurious classes and methods as possible. Due to difficulties in comparing IPTAs and CFGs from various tools, [Lho07] has designed a tool to compare CFGs.

3.2 Analysis in the JVM

The challenge of IPTA is to analyse only truly reachable methods. The initial challenge is *when* and *how* to get the smallest sound CFG and class hierarchy. The *when* is determined by whether the analysis is SA, DA or hybrid analysis as shown in figure 3.1.

SA in JVM must have a low overhead. DA generates a precise solution, but only for one specific run. Hybrid analysis has been most looked at for generating test case solutions, but not in the JVM. SA in the JVM can be used for optimizers. Both SA and Hybrid analysis are used for software verifiers and test code generators. [Fol07] points out that for this reason some have suggested or used a modified classfile formats such as *stackmap* in Java 1.6.

3.2.1 Static Analysis in JVM

[GC01] built and tested a parameterized framework for IPTA algorithms in the Vortex compiler for Java and Cecil. The parameters (method, instance, class, environment (scope)) and their attributes indicated how much type and context information they kept. Initialization parameters determined the starting CFG. For worst cases, CHA was used as a base to compare context insensitive algorithms and 0-CFA was used as a base to compare context-sensitive algorithms for improvements. The result was a lattice of the relative precisions of the CFGs the algorithms produced and as well as scalability information.

3.2.2 Dynamic Analysis in JVM

[QH04] comments that although IPTA is popular, DA IPTA has not yet been widely adopted, but an exception is dynamic devirtualization in JITs.

[QH04] tested **dynamic versions of CHA and XTA** to eliminate unused fields in the Jikes RVM. They tested inlining with lazy loading by adding CG profiling code. This type analysis code was invoked when a method was invoked. Nodes and edges for the *called* and *called by* information was gathered, before allowing the JVM to continue to compile or execute the method. For interfaces two approaches were used to resolve possible receiver methods. The first more exact approach kept a caller index and the second approach used dynamic CHA. Due to few interfaces in their test cases, they used the dynamic CHA approach for final tests. Invalidation techniques were used to determine if the inlined code was still valid whenever the parent method was invoked. The results were mixed. They report 20% to 50% fewer call edges with dynamic XTA than dynamic CHA and a performance degradation in the SPECjvm98 benchmarks.

[QH05] builds on these results. [QH05] reported that comparing **dynamic versions CHA, RTA, XTA and VTA** against the actual runtime results, that CHA gave almost perfect resolution for all *invokevirtual* dynamically monomorphic methods, but not for *invokeinterface.* VTA performed best for resolving *invokeinterface* dynamically monomorphic methods. However there were few opportunities in the standard benchmarks to improve over CHA since they are mostly monomorphic. Only the results for CHA and VTA are reported, since RTA and XTA did not improve on CHA.

VTA was implemented in stages where the nodes and edges were created during the bytecode parse in the Jikes RVM optimizing front-end. Following the design in [QH04] the type resolution occurs from the type information block stub when the method is invoked. The VTA collapsing strongly connected components step did not occur for dynamic VTA.

[Hir04] explores **connectivity-based garbage collection (GC) using Andersen points-to DA**. Since objects, which reference each other, are likely to be no longer needed at the same time, such objects were allocated to a partition. When all objects in a partition were no longer used, the partition could be freed. The hope was that freeing partitions of memory earlier might avoid triggering a full GC of the whole program at once because the analysis was almost out of memory.

3.2.3 Hybrid Analysis

Ernst in [Ern03] encourages exploration of hybrid analysis, using of both SA and DA, in either order, where one analysis gathers the information and the other refines the information for a more precise end analysis.

DA to SA Ryder refers to DA feeding into SA as *blended analysis.* [DRS07] uses a blended analysis, which feeds a CG from a trace of a single DA into a SA tool to load only classes actually used. The main client application is escape analysis of types using points-to object representation for heap analysis and test case coverage. Whole program problems are avoided since all classes loaded and methods called are logged.

DA to SA to DA (DSD) Hybrid analysis can feed information back and forth to gather information, use the information, and verify the resulting use of the information. Csallner et.al in [CS06] use (step 1) DA to capture correct information about a program's run. Step 2 feeds the the captured CG into a SA to use as its starting CG. The step 2 SA results are used to generate test cases. Step 3 DA verifies the SA results. They found DSD to be more precise than tools using just SA or DA and to be especially helpful for avoiding false positives when evaluating exceptions.

3.2.4 No Analysis

[LYK+00] used no analysis for inline devirtualization, but used **polymorphic inline caches (PICs), instead of a virtual method table (VMT)** to dispatch methods. A method stub may be the inlined method, a switch test for multiple inlined methods, or jump to the method. The methods are compiled for the one type as if they were static and cache executes the compiled version based on the type. [LYK+00] found a speed up as much as a geometric mean of 3% for monomorphic inline caches and 9% for polymorphic inline caches. That inline caches match hardware branch prediction in newer hardware gave an extra performance boost over VMTs. Finding in the SPECjvm98 about 85% of virtual calls use a monomorphic type and about 90% have a single target method, they inlined without testing and backed out as needed.

Hind et.al. in [LAHC06] uses statistical profiling, but no traditional SA. The results varied from 10-20% improvement to 10% degradegation. Using hardware performance monitors (hpm) counter and timer statistical profiling to identify method candidates for optimization reports a 5.7% average improvement in [BGH+07]. They considered direct application to inlining as a future application. Using polling as part of the profiling can mediate optimizations applied to a method in a loop, which for inlining may be invalidated when the loop is run a second time.

3.3 Propagation

In [HDDH07] several type propagations were tried with a points-to analysis:

- **eager** propagates as soon as a type change is found;

- **batch** propagates at a specific program point like garbage collection;
- **end** propagates types at the end of the program.

Using SA points-to analysis, propagating types at the *end* was faster. Using DA points-to analysis the batch propagation point was just *before GC*. In [QH05] dynamic VTA found "both eager and batch propagation efficient."

[QH05] found *eager* propagation with dynamic VTA was faster than *batch* propagation.

3.4 Class Loading affect on Analysis

[LAM07] tested eager pre-loading of classes for SA in JVM, because their analysis was being invalidated and repeated too often. They found a better solution was to batch the analysis by delaying any needed analysis until the program was no longer "frequently" loading classes. Batched analysis was invalidated often only if the program gradually loaded classes.

For real-time (RT) Java, [Fol07] reiterates how important keeping the class loading to a minimum for RT systems is. [Fol07] uses an application extractor, Websphere, a second generation tool of JAX, which uses Iteration Type Analysis (ITA) a version of XTA which ignores classes which are set to Null.

Lazy loading delays loading classes until the class is needed, causing the analysis to have to handle classes inserted into a class cone [QH05].

A problem with DA, is it only validate up to the current point in the program due to lazy loading. So DA can be invalidated later in the program and must be redone. [LAM07] found that for *javac* and *jack* programs that classes were loaded gradually throughout the program, caused the IPTA with lazy loading or non-preloading (NP) to be redone 72 and 18 times respectively. Tests showed a 1.87 times average performance improvement for IPTA when classes were pre-loaded.

A non-preloading delayed (NPD) IPTA was introduced, which profiles the time between class loads and delays IPTA until the pattern says no new class loads are expected to immediately invalidate the IPTA. NPD brought significant saving for the programs that gradually loaded classes like *javac* and *jack* and overall an average of 2.30 times performance improvement over non-preloading (NP) lazy loading. However for programs like *compress* and *mtrt*, where all classes are loaded at the beginning of the program, there was a low, in comparison to reanalyze costs, IPTA performance penalty of 25% due keeping the class loading history and delaying IPTA when it was not needed. For *jack* with NPD, the IPTA was only performed once. For *javac* with NPD the IPTA was recalculated 4 times rather than 72 times with NP.

3.5 Whole Program Solutions

Bytecode analysis starting from the *main* method will not be complete. Tomcat servlet programs are a well-known example of use of application class loader. SPECjvm98 invokes the *main* method of its benchmarks dynamically.

The Java ME KVM solution ([KVM00]) is to not allow reflection, dynamic class loading or method invocation, even from native methods. The KNI requires the classes and methods to be loaded to be supplied to the KVM. JAX supplies dynamically invoked methods as entry points [TLSS99].

3.5.1 Supplied Inputs in Static Analysis

XTA ([TP00]) and VTA ([SHR$^+$00])supplied a hand coded analysis of classes and methods missed by the analysis due to Java whole program problems. However CGs relying on a user-provided specification to obtain a conservative CG had 1.43 to 6.58 times more methods than the actual CG ([LWL05]).

3.5.2 String Analysis

For a SA, [LWL05] using points-to analysis looked in the bytecode for a pattern of: the method that performed reflection (*Class.forName*), an *instanceOf* in the bytecode, and invocation of *method.invoke* as hints that reflection was used. They were able to resolved 95% of reflective calls.

For DA, [Hir04] for GC used a similar approach, plus examining the contents of strings for dynamically loaded classes and dynamically invoked methods. They used these hints, together with what classes were actually loaded and what methods were invoked to surmise what methods were called due to reflection or by a native method.

JNI calls can only be examined via their inputs and return values, since the analysis cannot be generalized for all languages and hardware. Jikes is implemented in Java and uses Java reflection calls to implement JNI, so Jikes can gather information from JNI using the same technique used for handling reflection.[HDDH07]

3.5.3 Entry Methods

Jikes optimizes its boot process with the **Jikes boot image tool [AAB$^+$05]**. Besides the *main* method, dynamically loaded methods are treated as entry methods[TLSS99]. The boot process is optimized and the classes and methods used are compiled and saved in a boot image. This boot image is loaded at program begin, rather than repeating optimizations and compilation of the same start-up code for every run.

3.6 Benchmarks Characteristics

[QH05] found in general the commonly used Java Benchmarks have few polymorphic methods. [QH05] test results (with no hot method profiling) showed CHA found most monomorphic methods (99%) and VTA found most of the few monomorphic interface calls (86%-99%) in *db, javac, jack* and SPECjbb2000. Lee et.al. [LYK$^+$00] noted most methods in the SPECjvm98 benchmark were monomorphic and "type feedback cannot improve further over polymorphic inline caches and even degrades the performance for some programs.".

Chapter 4

Motivating Examples

The precision and overhead of a IPTA algorithm determines its usefulness for optimizations. This usefulness is challenged by method polymorphism through virtual methods. Traditional tests ([Dea96]) have been used to determine if a method is monomorphic without using an extensive analysis.

A test case demonstrates how the three CFG type analysis algorithms, (RTA, XTA, VTA) compared in this thesis, as well the algorithm used for comparison (CHA), handle the challenge of resolving virtual method calls. Andersen points-to was not implemented, but is included here for comparison, since it is the basis for many current IPTA algorithms.

The test class hierarchy used for this chapter is:

*class **E** extends class **D** extends class **C** extends class **B** extends class **A***
*class **X** extends class **A***

Every class has a method *m1( )*. Class **A** has a method *m2( )*.

The test case for virtual calls in table 4.1 contains the test code, bytecode call, method actually called and what methods each algorithm consider reachable for each method call. Each *new* is labeled as Objects 1-4 or O_1-O_4 for the Points-To algorithm.

Use of an *optimistic* or a *pessimistic* class hierarchy affects both the precision and overhead of IPTA algorithms. First the use of the optimistic class hierarchy with virtual calls is examined in section 4.2 and then the use of a pessimistic class hierarchy is examined in section 4.3.

4.1 Simple Tests

Some traditional ([Dea96]) *simple tests* to prove a method, $C.m$, is monomorphic are:

1. language constructs (non-virtual);

2. no sub-classes exist (*leaf*);

3. *only 1* available definition of the method, m in $\widehat{C}$.

A *non-virtual method* can be recognized by the appropriate *invoke* bytecode: (sec. 2.7.2)(*static*, *special*, *virtual* for *final* methods);

A method in a *leaf* class, like E, has no subclasses, which could override the method. So a leaf method, like $E.m1$, is always monomorphic. If relative numbering schemes of the class hierarchy are used, which use a base value and depth as described in [KH02] and used in CACAO, a *leaf* class has a depth of zero.

Only 1 definition in $\widehat{C}$ of m occurs when the method is only defined in C or a super class of C. For example, *invokevirtual* $X.m1$ has only 1 method definition of foo in $\widehat{X}$ and always resolves to $X.m1$. Similarly *invokevirtual* $Y.m1$ finds no definition of foo in Y and resolves to $X.m1$ in Y's super class.

In table 4.1 extra analysis information gathered only when using the pessimistic algorithm and class hierarchy are shown in blue.

These simple tests are independent of a specific IPTA algorithm.

4.2 Virtual Calls

Although multiple method definitions exist in reachable classes, only one method definition may be actually reachable by the program. An IPTA must be used to heuristically determine, if **only 1** method definition is **used** in $\widehat{C}$. An IPTA gathers class information, which is used to determine which classes can be *used* by a method invocation. If only 1 class with the method definition is considered *used* by the IPTA, then the method is monomorphic. Further if there is only one method definition in the classes *used*, then the method is monomorphic.

The following examines optimistic versions (section 2.1.2) of the five algorithms using an optimistic class hierarchy. The optimistic class hierarchy loads only classes found reachable, as found in DA due to lazy loading by the JVM. Optimistic IPTA builds its CFG on-the-fly starting with *main*. Methods found reachable by the IPTA are added to the mWL during the analysis and are analyzed until the mWL is empty.

4.2.1 CHA

Class Hierarchy Analysis (CHA) considers all classes in the class hierarchy *used*. In the JVM the class hierarchy is all loaded classes. A method *reachable* and is added to the mWL, if the invoked method is defined in $\widehat{C}$ of the invoking class of the method. The scope of the method CG is: "program calls". There are no edges between methods.

In the example in method *f*, *a2.m1* is *A.m1* in the bytecode. Classes **A,B,D,E** are loaded because they are instantiated. Class *C* is loaded because it is the super class of classes *D* and *E*. Since all classes in the $\widehat{A}$ have method definitions for *foo*, all five methods definitions of *foo* (*A.m1, B.m1, C.m1, D.m1, E.m1*) are considered reachable by CHA. In method *g*, variable *b4* has a type of *B*. Class *B* has a method definition for *foo*. **Using $\widehat{B}$, CHA finds for the call *b4.m1* that 4 methods are reachable as shown in figure 4.1.**

4.2.2 RTA

RTA improves on CHA, by only considering classes instantiated {*A,B,D,E*} as used. So in method *f*, the call *a2.m1* resolves to 4 methods. The abstraction is

public class Testcase {	**invoke virtual**	**Calls**	**CHA**	**RTA**	**XTA**	**VTA**	**Pts -To**
public static main()							
{A a1;							
B b1 = new B( );							O_1
g(b1);	g(A)	**g(B)**	b1(B,C,D,E)	b1(B,D,E)	b1(B)	b1(B)	
f(b1);	f(B)	**f(B)**	b1 (B,C,D,E)	b1 (B,D,E)	b1(B)	b1(B)	
}							
static void f(A a2)							
{A a1 = new A( );							O_2
a2.m1( );	A.m1	**B.m1**	A,B,C,D,E,X	A,B,D,E,X	A,B	B	B
a2.m2( );	A.m2	**A.m2**	A	A	A	A	A
}							
static void g(B b2)							
{B b3 = b2;							
B b4 = new E( );							O_3
b3.m1( );	B.m1	**B.m1**	B,C,D,E	B,D,E	B,D,E	B,D	B,D
b3 = new D( );							O_4
b3.m1( );	B.m1	**D.m1**	B,C,D,E	B,D,E	B,D,E	B,D	B,D
b4.m1( );	B.m1	**E.m1**	B,C,D,E	B,D,E	B,D,E	E	E
}							
// not called							
static void h( ) {							
X x1 = new X( );							O_5
x1.m1( );	X.m1						
} }							

Table 4.1: Example Program with IPTA reachable Methods Comparison

Algorithm	**Pgm Classes**
CHA	$\{A,B,C,D,E\}$
RTA	$\{A,B,D,E\}$

Table 4.2: CHA and RTA type propagation set for *b4.m1*

still "program calls". In method *g*, the first *b3.m1* call marks *C.m1* and *D.m1*, since neither class *C* or *D* have been instantiated yet. *D.m1* is added to the reachable mWL after class *D* is instantiated by *new D()*. Since Class *C* is never instantiated, *C.m1* is never added to the reachable mWL. Figure 4.1 shows that since *b4* is of type *B*, that RTA finds 3 methods reachable for the call *b4.m1*. **RTA finds 1 less reachable method, than CHA for all method calls in the example program.**

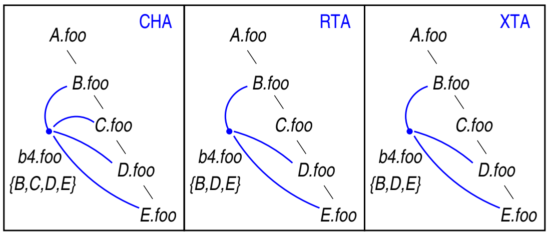


Figure 4.1: CHA, RTA, XTA for *b4.m1*

4.2.3 XTA

XTA improves on RTA by keeping a set of classes used for each reachable method and a field. Classes can be added to a method's type set via parameters in,

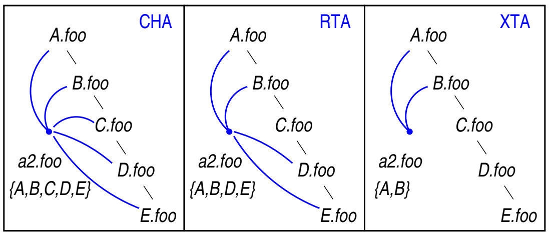


Figure 4.2: CHA, RTA, XTA for *a2.m1*

Method	ParamSet	Rtn	Method Classes
main		{ }	{B}
f	{B}	{ }	{A, B}
g	{B}	{ }	{B,D,E}

Table 4.3: XTA type propagation set

return value out or fields. In the example for method *f*, class *B* is added to type set via an input parameter and class *A* is added via the *new* instantiation. So the class set for method *f* is {A,B} making only *A.m1* and *B.m1* reachable from *f*. For method *g* the class set is {*B,D,E*}. So for *b4.m1*, as shown in figure 4.1, there are three reachable methods. **XTA finds 1 less reachable method for the *a2.m1* (figure 4.2) than RTA**, but has no improvement over RTA for other method calls in this example.

4.2.4 VTA

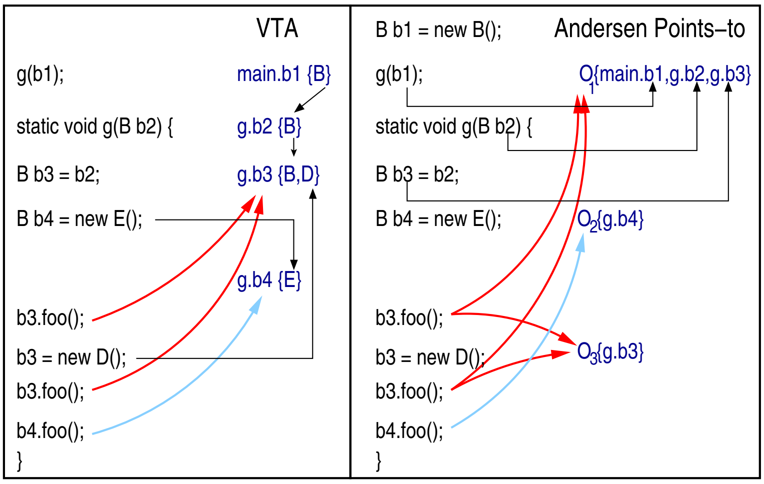


Figure 4.3: VTA and Andersen Points-to for method *g*

VTA follows the flow of types through variables. In method *f*, class *B* is added to *f.b2*'s class set from *main.b1* through the parameter list. Class *A* is instantiated in method *f*, but never used by *b2*. So the *b2.m1* call finds only *B.m1* reachable.

As shown in figure 4.3 in method g, variables *b3* and *b4* have different type sets of {B,D} and {E} respectively. VTA finds *b4.m1* to be dynamically monomorphic since VTA finds only *E.m1* reachable here. So VTA allows *E.m1* to be a candidate to be inlined. VTA finds 1 less reachable method than XTA for all method calls in the example program. **VTA finds 1 method invocation,**

_b4.m1()_ that can be devirtualized. VTA is the MFP (sec.2) of algorithms tested.

Variable	Classes	FlowsTo	FlowsFrom
main.b1	{B}	{f.a2, g.b2}	{ }
f.a1, f.a2, g.b4	{B}	{ }	{ }
g.b2	{B}	{f.b2}	{ }
g.b3	{B}	{ }	{f.b2}

Table 4.4: VTA type propagation set

4.2.5 Andersen Points-to

Andersen points-to was not implemented, but is included for comparison since it is the basis for many current IPTA algorithms. Each *new* is an object reference or creation site. The creation sites are labeled in table 4.1 as Objects 1-4 or O_1-O_4. O_1 is the object created by $newB()$ in the *main* method. O_1 is pointed to by variables: $main.b1$ through a *new* object assignment, *f.b2* as a parameter, *g.b2* as a parameter and *g.b3* through assignment. The points-to set for the $newA()$ creation site, O_2, has an empty points-to set { }. O_3, an instantiation of class *E*, is pointed to by the variable *g.b4*. O_4, an instantiation of class *D* is pointed to by *g.b3*.

Methods are considered reachable by Andersen Points-To, if the variable used to invoke a method are in an object's points-to set. As shown in figure 4.3, for method g, the variable *b3* points to both O_1 of type *B* and O_4 of type *D*. So the *b3.m1* call consider two methods reachable. Variable *g.b4* points to O_3 of type *E*. **The call _b4.m1_ only considers 1 method reachable, so _E.m1_ could be devirtualized by Andersen points-to like VTA.**

It interesting to note the roles of the nodes and sets are reversed for VTA and Andersen Points-To analysis. For this example both VTA and Points-To were able to find 1 dynamically monomorphic method. CHA had the most spurious methods. The next chapter will explain how RTA, XTA and VTA were specifically implemented in CACAO.

Variable v	PointsTo(v)	Object O	FlowsTo(O)
main.b1	$\{O_1\}$	O_1	{main.b1, f.a2, g.b2, g.b3 }
f.a2	$\{O_1\}$	O_2	{f.a1}
f.a1	$\{O_2\}$	O_3	{g.b4}
g.b2	$\{O_1\}$	O_4	{g.b3}
g.b3	$\{O_1, O_4\}$		
g.b4	$\{O_3\}$		

Table 4.5: Andersen Points-to type propagation sets

4.3 Pessimistic vs. Optimistic Class Hierarchy

A pessimistic class hierarchy starts with all classes and methods, that might be used when the class is used, which is all classes in a classfile's constant pool. Method *h* is never called, but the class *X* will still be loaded and in the pessimistic class hierarchy. Therefore although class *X* is never used, pessimistic CHA (pCHA) will find $X.m1$ reachable whenever $A.m1$ is invoked since *X* is in $\widehat{A}$. Similarly since there is a *new* $X()$ in h pessimistic RTA (pRTA) will find $X.m1$ reachable whenever $A.m1$ is invoked.

Pessimistic versions of XTA, VTA and Points-To will start with both h and $X.m1$ in their CFGs and in the end prune both methods out of their CFGs. Additionally VTA is a pessimistic algorithm and would allocate variable nodes for the local variable $x1$. Variables are not shown in the testcase table. Similarly a pessimistic Andersen Points-To would allocate an object, O_5, node due to the *new* $X()$, which will also not be used.

Existing IPTA implementations use a pessimistic class hierarchy for SA and an optimistic class hierarchy for DA.

CHA ([Dea96]) and VTA ([SHR$^+$00]) are pessimistic algorithms, although dynamic CHA is optimistic ([QH05]) due to lazy loading by the JVM. Even in a dynamic implementation VTA prunes a CHA CFG ([QH05]). Both pessimistic and optimistic versions of RTA ([BS96], [SHR$^+$00]) and Andersen's Points-To ([And94],[HDDH07]) are used. XTA is an optimistic algorithm ([TP00]) even when used for SA.

Chapter 5

Analysis in CACAO

This chapter addresses the major issues implementing the three CFG type analysis algorithms, RTA, XTA, and VTA, in four analysis modes (static (SA), hybrid pickup (HPA-SA, HPA-DA), dynamic (DA)) in CACAO. The choice of an optimistic class loading scheme, *delayed class loading*, and class usage by the algorithms is explained next. All algorithms are implemented with same general method worklist (mWL) algorithm to process the methods they find reachable. However there are mWL usage differences between modes.

Figure A.1 shows the three algorithms implemented and highlights their similarities and differences. A method is reachable, if the class is *used*, as defined by the analysis algorithm. The algorithms differ in the scope and discovery of used classes and reachable methods. A major distinction between algorithms is the program entity that types are collected on, which results in differences in how the algorithms recognize and propagate reachable types.

SA whole program approach is explained. The chapter concludes with information about how the devirtualization statistics are gathered and how the implementation is tested.

An overview of the bytecodes was given in section 2.7. In the rest of this thesis, a calling method will always have a lower subscript number than the method it calls. So $C.m_1$ is called by $C.m_0$ and calls $C.m_2$.

5.1 Analysis Modes

Figure 5.1 extends figure 3.1 to show where in CACAO this implementation of static analysis (SA), dynamic analysis (DA), and hybrid pickup analysis (HPA) modes occur in the Java application process.

5.1.1 One source, multiple modes

The analysis modes were implemented in the same source code, but as separate modules by using three C language *#define* macro flags to compile or exclude the mode specific code. These compile-time flags match the mode definitions.

ANALMODE_STATIC, the SA flag, includes code to cause the whole static analysis to occur before any code is compiled.

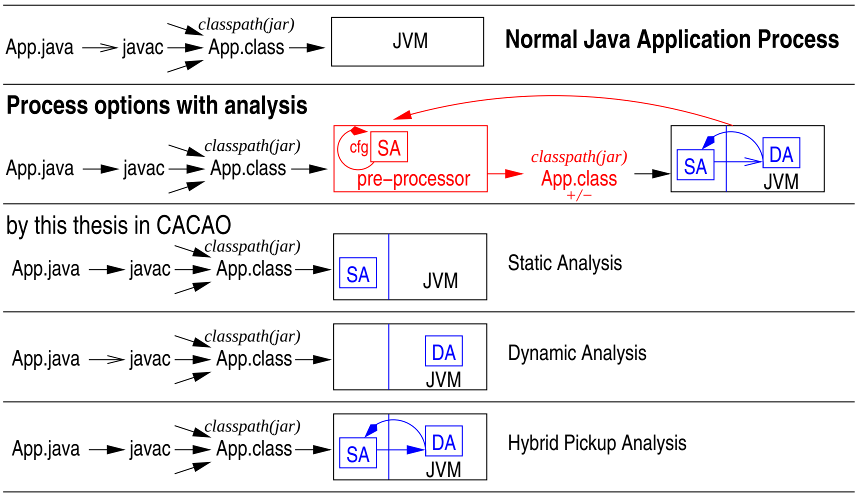


Figure 5.1: Where IPTA in CACAO occurs in Java process (extends fig.3.1)

ANALMODE_DYNAMIC, the DA flag, includes code to cause the dynamic analysis of just one method, just before its normal JIT parse.

ANALMODE_PICKUP_MISSED, the HPA flag, includes or excludes code to cause an analysis to start with this method at runtime, when a method to be JIT parsed has not yet been analyzed. This defines both a static pickup (SA-HPA) and dynamic mixed mode (DA-HPA).

The analysis code must be compiled with either the *SA flag* or *DA flag* defined, but not both. The *HPA flag* may be defined to enable code for a hybrid analysis, which picks up entry points to begin an analysis round.

5.1.2 Static Analysis Location in CACAO

Factors to be considered when deciding where the algorithms are located in CACAO, in order of influence, are:

- the first pass of optimizations using the analysis,
- cost of using a specific compiler pass,
- input suitability and cost to the algorithms,
- the comparability of the algorithms.

The CACAO JIT compiler basic phases consist of a bytecode parser, stack analyzer, type checker, register allocation, and code generation followed by running the method as described in [Kra98] and [Ste07]. Each pass refines its input to an intermediate representation closer to the machine code.

Inlining in CACAO was being implemented in the parse pass at the time this implementation started, making parse the first compiler pass needing the analysis. **A limiting characteristic of SA is that it must be completed for the whole program before any code is generated**, so the analysis:

- must be implemented as a separate first compiler pass, which

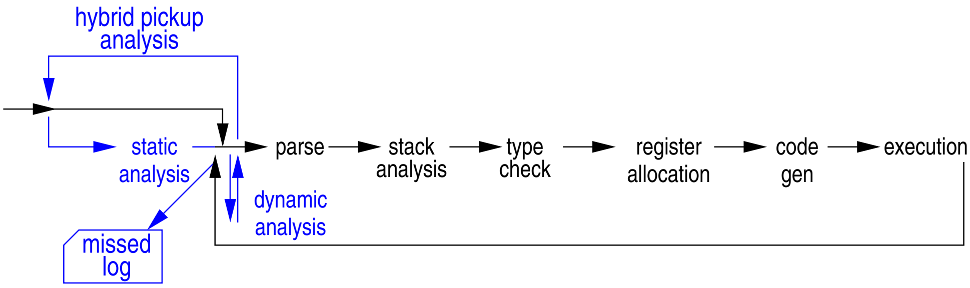


Figure 5.2: Analysis in CACAO Passes

- has bytecode as input and
- could not be mixed in with the code of another compiler pass.

So SA is implemented as a separate pass before the first JIT parse (fig.5.2).

5.1.3 Bytecode or Intermediate code input

The SA pass could still translate its bytecode input into intermediate code like Sable's Jimple ([VRCG$^+$99]) before analysis. Bytecode input is enough for an analysis algorithm, when the type information needed is available in just one bytecode. If type information from more than one bytecode is needed, then reachable types of variables loaded on the stack must be examined. Assignments (sec. 2.7.5) and the variables passed as parameters are examples of code, which in the JVM require stack analysis.

RTA and XTA ignore type flow within a method and only require a *trimmed-down bytecode parse*, since only bytecodes with information needed by the algorithm to determine reachable methods need to be parsed by the algorithm. VTA additionally needs a stack analysis for the type flow between variables. However translation to intermediate code adds to the runtime costs. This implementation is in a JIT, so low runtime overhead is considered important. Using an intermediate translation for RTA and XTA, when it is not needed did not seem justified. Adding the extra overhead of translating bytecode to intermediate code for just VTA, did not seem like a fair comparison of the algorithms, if it could be avoided.

A *compromise* was chosen and a trimmed-down stack analysis is implemented for VTA. The *trimmed-down stack analysis* ignored non-reference types except to track their affect on the stack depth. VTA is able to build its CG and CFG optimistically *on-the-fly* instead of depending on less precise pessimistic starting CG and CFG (sec.s 2.1.2, 4.3), like RTA and XTA.

5.1.4 Method Worklist

Each analysis algorithm takes a method from the worklist of reachable methods (mWL), and performs an analysis parse. Methods found to be reachable from the method being analyzed are added to the mWL. This process continues until all methods in the mWL have been analyzed, which can occur when no new methods are found to be reachable. Methods are added to the mWL in the following three phases for SA:

Phase 1: Initialization The mWL is ***initialized*** with *main* method and the entry methods that the JVM boot image invokes directly.

Phase 2: Reachable by analysis Starting with the entry methods all methods in the mWL are analyzed and methods found ***reachable*** by the analysis algorithm are added to the mWL. Any iteration over the mWL by the algorithm occurs in this phase.

Phase 3: User Supplied Analysis After *halt* method, any methods ***supplied*** by the user that are known to be missed by the analysis are added to the mWL and analyzed.

Phases one and three are required for a sound **SA**, but optional in **HPA** modes and not used in **DA** mode. In HPA mode the missed method is added to the mWL like an entry method and an analysis starts. Methods found reachable are added to the wML and are analyzed until the mWL is empty. This process is repeated again, if another method is missed by the analysis. DA adds no methods to mWL except itself, but it does add edges and keeps track of class usage.

5.1.5 Static Analysis with completeness checks

SA should be complete before the first method is JVM JIT parsed. In CACAO regular parse code before the JIT bytecode parse starts, a test is made to see if the method has been statically analyzed. If the method had not been analyzed the method is logged to a *missed* file. This feedback helps the SA user know what methods they need to supply as inputs to analyze the whole program. Missed methods only need their method flags, like *PUBLIC* deleted for the missed method to be in the correct input format. If the SA is complete, then the *missed* file will be empty.

Additionally this completeness test in *parse* served as a starting point to start both the HPA and DA modes of analysis for the method to be parsed (fig.5.2).

5.1.6 Hybrid analysis: Picking up Entry Points

Rather than just logging that a method has not been statically analyzed before the parse pass, a HPA is started with this method as an entry method. **SA-HPA** analyzes all methods reachable from this entry method and is still finished before the originally missed method is parsed. HPA results are available to any optimizations needing them like pure SA. And whole program issues are handled trivially, since a dynamically invoked method must be parsed by the JVM. For SA-HPA entry methods may be optionally supplied, but are not required. An entry point may reduce precision if not used by the current run. **DA-HPA** works the same but ignores any inputs, so all entry points are always picked-up.

5.1.7 Dynamic analysis of the ideal Call Graph

In the parse pass, rather than logging or starting a SA of all reachable methods, a DA of just the method to be parsed is started. Since only methods executed are analyzed, the CG used for analysis is the ideal CG of the whole application run. DA mode matches what the JVM JIT has available up to the parse phase of the method analyzed, however the complete analysis is not finished until the program finishes.

5.2 Classes

5.2.1 When to Load

The class loading scheme ([QH05],[HDH04]) affects the size of the class hierarchy. CACAO class loader evolved from *eager class loading* to *lazy class loading* with an in-between step of *on-demand class loading.* The in-between step in CACAO loaded classes *on-demand*, as needed during the JIT parse of a method's bytecode. If a class is instantiated in a code branch that is never reached, then a class could be loaded that is not needed. The actual *lazy loading* implementation delays class loading until just before runtime. No unused classes are loaded.

Analysis algorithms need the class file loaded to resolve reachable methods and parse its bytecode. The analysis information is assumed to be used with one application run in a JIT and not for all possible application runs. So only the classes needed by this particular run are needed. Smaller class hierarchies reduce the number of spurious methods being analyzed.

An offline JVM compiler requires an analysis be valid for all possible application paths. Unless all methods are used by every possible application code path, the analysis CG will have spurious methods. But for a JIT JVM, where only one code path is used, considering all code paths causes the number of spurious methods to increase and the analysis precision to decrease.

The **design issues** for choosing between the three class loading options are:

Eager loading of all possible classes from the class file constant pool, can cause extra classes to be loaded. Extra classes in the class hierarchy can mean extra spurious method definitions in the analysis CG, if a parent class is the method's invocation type as shown by example in section 4.2. The analysis may appear to take less time, because the classes have been preloaded. If all classes in the constant pool are not actually used, this pre-load could increases the total time used for class loading.

On-demand loading of a class by the analysis when a method or field in the class is determined reachable, creates a more precise class hierarchy for a run. The analysis must take time to load a class, but the classes will be preloaded for the JIT JVM. However any pre-loading of classes, due to spurious reachable methods in SA or HPA mode, should be hidden from the JVM until the class is actually used by the JVM.

Lazy loading (at runtime) of a class when its constructor is to be compiled and immediately executed means the class is available too late for SA. Any class type analysis and verification is delayed until just before runtime. Class analysis cannot be performed earlier for predictive optimizations because the class has not yet been loaded. DA implementation works because the method has to be parsed before it is executed and DA does not analyze any spurious methods. Analysis information could be collected during type analysis after a used class is loaded. Hotspot profilers can add optimizations for hot methods later using the actual analysis information stand for an on-demand analysis. **On-demand class loading was chosen**, since it allows both SA and DA, as well as limiting the class hierarchy to reachable classes for the analyzed program. This implementation used the last version of CACAO that considered any kind of pre-loading acceptable with no attempt to hide extra classes loaded.

5.2.2 Identifying Used Classes

Each algorithm finds a class to be *used* for a specific scope within the program as summarized in table 5.1. Classes are considered *used* by TA algorithms when a *new* bytecode instantiates the class. In CHA if a class is used in bytecode reachable by the analysis, the class is loaded and considered *used*.

A flag is used by RTA (fig.5.3) to distinguish between classes in the class hierarchy, which have been directly instantiated via a *new* bytecode and loaded for other reasons, like being a super-class of a used class. The RTA class flag is initialized to *notused* when the class is loaded. When a class is instantiated via *new* (or dynamically) the class flag is changed to *used*. For classes which use a static method or field, the class flag is set to *partused*, so not all methods in this class are considered used. The *partused* state may later be changed to *used*, if a non-static method from the class is found reachable.

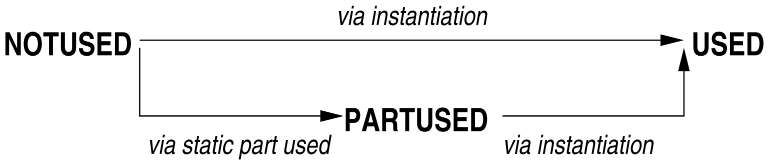


Figure 5.3: RTA Class Flag States

A *partused* flag indicates something in the class is used, but the class has not been instantiated. A class is considered *partused* for:

- **non-virtual invokes** (*static*, *special*, *virtual* for *final* methods);
- **static field bytecodes** (*getstatic*, *putstatic*);
- **cast bytecodes** (*checkcast*, *instanceof*);
- **default super constructor** because the class has not been instantiated, but its default instance constructor, <init>, is invoked by a subclass instance constructor which has been instantiated.

XTA adds instantiated classes to its method class set. A method's class set, $C.m_{cs}$, may be added to via type propagation of its parameters *in* or a return value *out* from a method call.

VTA adds instantiated classes to the class set of the variable. Class types are propagated to other variables via the stack. The VTA analysis stack pushes a VTA analysis variable pointer with all information about the variable.

For all three algorithms, when a class is instantiated, it is added to an *implemented by* list of all interface classes it implements. This allows faster resolution of interface classes during analysis. Additionally for XTA interface classes are kept separately to make class searches faster since classes for multiple variables are grouped together. VTA uses the basic variable class attributes to determine, if the classes are interface classes or not.

5.2.3 Instantiation, Constructors and Destructor

The static class constructor, <*clinit*>, and *finalize* methods are invoked directly from the JVM [LY99]. Except for DA, this must be simulated by analysis. When the analysis finds a class is *used* by the analysis the class constructor <*clinit*> and *finalize* methods are added to the mWL, if they exist. All classes considered *used* or *partused* in CACAO invoke their class constructor.

5.3 Building the Type propagation graph

Algo.	**Scope**	**Class Sets**	**CFG edges**
RTA	program	class flag, $C_{1.classflag}$	method flag, $C_1.m_{1.methflag}$
XTA	field	field, $C.f_{x.cs}$	
	method	method, $C_1.m_{1.cs}$ parameter, $C_1.m_1.p_{cs}$	(calls, calledBy) sets, $(C_1.m_{1.calledBy}$, $C_1.m_{1.calls})$ (marked, markedBy) sets, $(C_1.m_{1.markedBy}$, $C_1.m_{1.marks})$ interfaceCalls set $C_1.m_{1.interfaceCalls}$ fldsUsed set $C_1.m_{1.fldsUsed}$
VTA	variable . fields, $C.f_{x.v}$. method, $C_1.m_{1.v}$. local, $C_1.m_{1.v.loc[n].v}$. parameter, $C_1.m_{1.v.p[i]} = C_1.m_{1.v.loc[i].v}$. return, $C_1.m_{1.v}.r._v$	variable $C.f_{x.v.cs}$ $C_1.m_{1.v.cs}$	(to, from) sets $(C.f_{x.v.to}, C.f_{x.v.from})$ $(C_1.m_{1.v.to}, C_1.m_{1.v.from})$ $C_1.m_{v.1.vCallsites}$ $C_1.m_{v.1.iCallsites}$

Table 5.1: Class sets and CFG edge sets by algorithm

Flags and sets used by the algorithms are summarized in the table 5.1.

RTA identifies classes and methods by the use of its *classUsed* and *methUsed* flag respectively in the class information and method information structures. RTA keeps no call edges, since the class type scope is global over the whole program.

XTA and VTA must propagate types between methods and variables respectively. For XTA and VTA, a pointer to a structure with the analysis information is needed in the method information structure by the algorithm for the analysis. After the first iteration only the propagation graph is used.

XTA uses a class set for the methods and edges, *(Called, CalledBy)*, for type propagation. Since on-demand loading is used, additional sets were needed which record all method calls. Virtual and interface calls and interface classes used are kept separately to speed their resolution. These are needed due to allowing inserted classes. A method keeps a list of fields it uses, $C_1.m_{1.fldsUsed}$.

A method parameter set, $C.m.p$, is kept with all classes used by a method. A pointer into the method's class set and a change flag let the second iteration know if any propagation is needed. For XTA for each field, $C.f_x$, a set of used classes, $C.f_{x.cs}$, is kept.

VTA keeps *(To, From)* sets for variables' type propagation for a method, its local variables, $C_1.m_{1.v.loc[n].v}$, parameters, fields, $C.f_{x.v}$, and return, $C_1.m_{1.v}.r._v$, variables. Parameter 0 for virtual methods is *this* variable. Since parameters map to local variables, after VTA initialization of a method the parameters are treated as local variables.

The base VTA pointer in *methinfo* points to a method variable, which has local variables and a return variable, $C_1.m_{1.v}.r._v$ and may use fields, $C.m_{1.v.fldsUsed}$. Virtual, $C_1.m_{v.1.Callsites}$, and interface, $C_1.m_{v.1.iCallsites}$, method callsites are kept as variables. Callsites are needed due to on-demand loading and inserted classes and because a method may be invoked with a class instantiation, which is never stored in a local or return variable.

5.4 Method reachability

Algo.	**add** $C_2.m_2$ **to mWL** when analyzing $C_1.m_1$
RTA	$C_2.m_2 = used$ $mWL+ = C_2.m_2$
XTA	$C_2.m_2 = used$ $mWL+ = C_2.m_2$ $C_1.m_{1.calls}+ = C_2.m_2$ $C_2.m_{2.calledBy}+ = C_1.m_1$
VTA	$C_2.m_2 = used$ $mWL+ = C_2.m_2$ $C_1.m_{1.v.to}+ = C_2.m_{2.v}$ $C_2.m_{2.v.from}+ = C_1.m_{1.v}$

Table 5.2: Algorithm comparison for adding any reachable method to mWL

5.4.1 Non-virtual methods

Table 5.2 compares how the three algorithms add methods to the mWL by the three algorithms. Non-virtual or instance methods always resolve to one method definition [LY99]. All analysis algorithms use the *non-virtual (static, special, final)* test as described in section 4.1. Invoked non-virtual methods are immediately added to the mWL.

5.4.2 Virtual and interface methods

Virtual methods are invoked by either *invokevirtual* or *invokeinterface.* When $C_1.m_1$ is invoked, all method definitions of m in $\widehat{C}$ whose subclass is considered *used* by the algorithm are added to the mWL. Table 5.3 summarizes the algorithm actions for RTA, XTA and VTA. The same basic technique is used for both kinds of invokes, but *invokeinterface* must consider multiple class cones and is summarized in table 5.4.

The class set associated with the method called is used to resolve a virtual call site. The greater the scope of the class set (table 5.1), the more classes it will have, which can result in more spurious methods. In CACAO, class inclusion checks within a class cone can use a fast range check as described in [KH02]. For both RTA and XTA the class set is a super set of the virtual method's calling class cone and may require a type check over multiple class cones. VTA had a

	***invokevirtual* $\mathbf{C_2.m_2}$** during analysis of $\mathrm{C_1.m_1}$	

RTA	For each subclass $C_{sub}.\mathrm{m}_2$
	in $\widehat{C_2}$,
	where $C_{sub}.\mathrm{m}_2$ is defined,
		if $C_{sub.classflag} = used$
		$\Rightarrow$ then add $C_{sub}.m_2$ to mWL;
		else $C_{sub}.m_2 = marked$

XTA	For each subclass C_{sub} in $(C_1.m_{1.cs} \cap \widehat{C_2})$, $C_1.m_{1.marks} += C_{sub}.m_2$ $C_{sub}.m_{2.markedBy} += C_1.m_1$	$\Rightarrow$ add $C_{sub}.m_2$ to mWL;
VTA	$C_1.m_{1.v.virtcall} += C_2.m_2$ For each subclass C_{sub} in $C_1.m_{1.v.cs}$,	$\Rightarrow$ add $C_{sub}.m_2$ to mWL;

Table 5.3: Algorithm comparison for *invokevirtual*

class set that is limited to a class cone already. So as the scope lessened, the complexity of resolving a virtual call lessened as the following explains closer and as summarized in table 5.3.

RTA could look to see if every *used* class is within $\widehat{C}$ of $C.m$ or look for the method definition, m, via the CACAO function to resolve method in every class in $\widehat{C}$ of $C.m$. Since class cone is smaller than all classes used by a program, the second option is used. Method definitions are added to the mWL, if the class is *used* <u>or</u> *marked* when the class is *notused* or *partused.*

XTA keeps classes used by the method in a class set and edges for type propagation via a list of methods it is *called by* and *calls.* The method's class set contains more classes than those in the class cone of the method's type. Propagation must use an intersection of the *called* method's class cone, $\widehat{C_2}$, and the *called by* method's used class set, $C_1.m_{1.cs}$. The resulting temporary class set is used to resolve reachable method definitions. Using the CACAO class numbering scheme made checking if a class is within the class cone a simple compare.

VTA has a used class sets for each variable. Since the class set, $v.cs$, contains only classes in the class cone of the variable's type, an intersection between the variable types is enough to determine the possible used classes with reachable virtual method definitions.

	invokeinterface $CI_2.mi_2$ during the analysis of $C_1.m_1$
RTA	For each class C_n implementing CI, process like *invokevirtual* $C_n.m_2$
XTA	$C_1.m_{1.interfaceCalls} += CI_2.mi_2$ For each class C_n implementing CI, process like *invokevirtual* $C_n.m_2$
VTA	$C_1.m_{1.v.iCallsites} += CI_2.mi_2$ For each class C_n implementing CI, process like *invokevirtual* $C_n.m_2$

Table 5.4: Algorithm comparison for *invokeinterface*

5.4.3 Method invocation before class *used*

A virtual or interface method call, $C.m$, may occur before the analysis has found all classes in $\widehat{C}$ to be *used.* Using *on-demand class loading* means a class may not be even loaded when the method call would be found reachable in the analysis. The technique used by [QH04] and [QH05] for DA is used in this implementation for inserted classes in SA and HPA.

A method flag added to the class information structure indicates if a class is:

notused The initial value when a class is loaded. Class not yet used at all.

marked The virtual method is reachable, but class is not yet used. So a method is not added to mWL yet, but will be if the class is *used* when the class is instantiated.

used A path to the method exists, so it's definitely reachable

Using *delayed class loading* for SA a method can be *marked* only if the class is loaded only because it is the super class of a *used* class.

In XTA and VTA, the equivalent to *marking* a method is keeping a list of virtual and interface method calls. In VTA the method variable keeps a list of variables used to invoke a method. When a class is found to be used for a method this list is checked, and reachable methods in the new classes are added to the mWL. Intersections of class sets as explained in the previous section can be used to determine reachable methods in the class added to the method or variable class set.

Classes inserted into the class hierarchy for RTA can have a hefty penalty. The only way to know if a method was called before the class was inserted into the class hierarchy is to check for the first super class where the *methodUsed* flag is set for all methods defined in the inserted class. XTA and VTA add methods from newly instantiated classes with the same process they use for adding virtual and interface methods using the list of methods previously invoked. The other option for RTA would be to keep a list of all call sites.

5.5 Field and Variable assignment

Table 5.5 summaries field and variable type propagation for the 3 algorithms.

5.5.1 Analysis Information gathered

RTA keeps no separate information on fields. XTA and VTA add a pointer to the CACAO field information structure to the analysis information gathered for a field. XTA keeps the field's base class type information and a set of classes actually used by the field. VTA keeps the field's base class type information and variable information structure.

5.5.2 Kinds of variables

In VTA all analysis information is kept in the form of a variable.

The following kinds of variables can be stored:

- method $(C.m.v)$
- local variables, $(C.m.l[n].v)$which may be associated with a parameter,
- static fields $(C.f_x)$,
- local fields$(C.f_y)$,

- return variable($C.m.r.v$).

Virtual and interface Method callsites are kept as a variable:

- virtual invocation sites, ($C.m.vCallsites.v$),
- interface invocation call sites, ($C.m.iCallsites.v$),

Virtual and interface methods are invoked with a variable, whose class set determines which method definitions are actually reachable. As stated earlier these callsites are used to determine reachable virtual and interface methods when a class is inserted into the class hierarchy. They also make an Andersen points-to analysis easy to change to or to expand the design for later optimizations.

A variable with reference information is any information that may be stored into a reference type variable or may use reference type information. When pushed on the stack the following kinds of reference types are not associated with an actual VTA variable:

- a new object contains the new type,
- a null object, so assignments are valid when *aconst_null, ifnull, ifnonnull* bytecodes are used,
- string constant, so the variable is given a String type.

Local variables have a positive identifier and other variable types have a negative identifier corresponding to the kind of variable.

5.5.3 Field assignment

$C.f$	**Field and Variable assignment** ($C_1.m_1$, $C.r$)
RTA	"global" so, no action taken
XTA	PUTs: $C.f_{x.cs} += C_1.m_{1.cs} \cap \widehat{C.f_x}$ GETs: $C_1.m_{1.cs} += C.f_{x.cs}$
VTA	PUTs (writes) : $C_1.f_{x.v.cs} += C_1.m_1.v_{cs} \cap \widehat{C.f}$ $C.f_{x.v.from} += C_1.m_1.v$ $C_1.m_1.v_{to} += C.f_{x.v}$ GETs (reads): $C_1.m_1.v_{cs} += C.f_{x.v.cs}$ $C_1.m_1.v_{from} += C.f_{x.v}$ $C.f_{x.v.to} += C_1.m_1.v$

Table 5.5: Field and Variable propagation comparison

Only fields with a reference (class) type are needed for analysis. A write (put) indicates the fields is actually stored into. A read (get) indicates actual use of the field's type information.

RTA stores no separate information for fields. Static fields are initialized in the class initiator method <clinit>, so the class initiator is always added to the mWL when *putstatic* and *getstatic* are parsed. If *notused* previously, the static field's class is considered *partused.* It is possible only a static field will be used from this class, but none of its methods. Local fields can be ignored since they are used by a reachable method within their class and they must instantiated via *new.*

XTA keeps a list of used static fields with a virtual type. So *getstatic* causes the types of the static field to flow into the method. And *putstatic* causes any classes *used* by the method, which are part of the field type's class cone to be added to the field's class set.

VTA field information, both static (*getstatic*, *putstatic*) and local (*getfield*, *putfield*), is kept as a variable node. The types used by the *from* variable and in the class cone of the *to* variable flow into the *to* variable.

5.5.4 VTA Variable assignment

Reference variable assignments use *store* bytecodes for reads and *load* bytecodes for writes. Additionally for VTA a pseudo VTA *reference variable only* stack is kept with just the type of the variable pushed and popped. The VTA stack depth and JVM stack depths are needed to know when a variable should be popped.

For example, table 5.6 shows the VTA reference stack before the invocation of virtual method with 3 parameters, where 2 parameters are reference types and 1 parameter is integer. Assuming nothing was previously on the stack, the VTA reference stack depth is 3, but JVM stack depth is 4 because of the integer parameter has also been pushed on to the stack. After the invoked method pops the parameters into local variables, both stacks are empty.

A separate bytecode pass was needed by VTA to determine basic blocks and their predecessors and successors and entry and exit stack depths.

When the variable is popped, the types in its class set flow into the variable or field it is assigned to. Method invocation and VTA reachability tests use the variable's class set to know what methods are reachable.

newobj A
L1
C.F

Table 5.6: VTA reference stack before *invokevirtual A.m(int, A, B)*

VTA assignments must handle the following *storing into* and *loading from* combinations:

- only load from:
  - new object;
  - NULL object;
  - return variable;
  - string constant (always of type: *class.java.lang.String*);
- both load *from* and store *to*:
  - local field variable
  - static field variable
  - local variable
- store only
  - return variable

5.6 Type Propagation

RTA does not have to explicitly do type propagation since its type scope is the program. For XTA and VTA, type propagation, when analyzing $C_1.m_1$, occurs:

- between methods, via parameters in, return value out;
- within a method, via fields and local variables;
- between and within methods via static fields.

A method's class set is used to identify reachable methods, preferably during the first analysis iteration. Type propagation could occur *eagerly* or *batch* as explained in (sec. 3.3) or a combination of both.

Eager propagation will propagate the classes in the calling method's class set, which are known at the time in the analysis of the call, whenever the method is found reachable by the analysis. Changes to the calling method's class set will be propagated to the called method in the next iteration.

Batch propagation would propagate types later at a logical point, if one exists, as a group. A logical point to batch propagate would be when a method's class set is more current and least likely to change. A method's class set will change most due its analysis, so a batch propagation of its types should occur after the method's analysis. Still the method's class set can change after this, so immediately before use by a called method a logical point to batch type propagation.

5.6.1 Propagation phases

The propagation phases for $C_1.m_1$ are:

1. *pre-* (batch) Parameter types in from all calling methods (table 5.7);
2. *during* (eager) Fields and variable assignments (table 5.5);
3. *post-* (batch) (tables 5.7,5.8)
   (a) (XTA only) Reiteration using the propagation graph after the initial analysis;
   (b) Return type sent to each *calledBy* methods.

When all the mWL has been processed both XTA and VTA do 1 extra propagation iteration.

5.6.2 Pre-analysis propagation

Before bytecode analysis, the method to be analyzed has been *called by* one or more methods, which will have already been analyzed.

A batch propagation of parameters types into a method, $C_1.m_1$, from all *calledBy* methods, $C_0.m_0$, as a pre-analysis step. This starts a method's analysis with the most current parameter class set(s).

5.6.3 Eager Field and Variable propagation

Field and variable read propagation is performed during the analysis to keep the analysis class sets current.

5.6.4 Post-analysis propagation

Although both XTA and VTA had only 1 extra iteration pass, a batch propagation occurred at the end of each method analysis. After analysis the method's class set should change less often.

For XTA the class set has been added to during analysis without type propagation, so the classes in the class sets of the following are propagated:

- used fields to fields and classes using the field;
- parameters are passed to all methods it calls as a kind of initialization;
- the most current return value back to its *calledBy* methods;
- adding all marked methods for classes added since the method was invoked.

For VTA, the return type flow back via *to* edges of the method. The local variable types mapped to the parameters flow into a called method via the *from* edges of the analyzed method.

$\mathbf{p}_n$	**Parameters in pre-analysis of** $C_1.m_1$
RTA	"global" so no action taken
XTA	*for all* $C_0.m_0[n]$ in $C_1.m_{1.calledBy} \Rightarrow C_1.m_{1.cs}+ = \bigcup\limits_n (C_0.m_{0.cs}[n] \cap C_1.m_1.\widehat{p_{cs}})$
VTA	$C_1.m_{1.v.cs}+ = \bigcup\limits_n (C_0.m_{0.v.cs}[n] \cap \widehat{C_1})$

Table 5.7: Pre-analysis *Parameters* propagation in comparison

$\mathbf{p}_n$	**Post-bytecode analysis of** $C_1.m_1$	
RTA	"global" so no action taken	
XTA	*for each* $C.m.f$ in $C_1.m.f_{1.fldused}$	$\Rightarrow C.m.f_{cs}+ = C_1.m_{1.cs}$
	for each $C_2.m_2[n]$ in $C_1.m_{1.calls}$	$\Rightarrow C_1.m_{1.cs}[n]+ = C_2.m_{2.cs}[n] \cap C_1.m_1.\widehat{p_{cs}}$
	for each $C_0.m_0$ in $C_1.m_{1.calledBy}$	$\Rightarrow C_0.m_{0.cs}+ = C_1.m_{1.cs} \cap C_1.m_1.\hat{r}$
VTA	*for each* $C.m.f$ in $C_1.m.f_{1.v.fldused}$	$\Rightarrow C.m.f_{v.cs}+ = C_1.m_{1.v.cs}$
	for each $C_2.m_2$, n a parameter	$\Rightarrow C_2.m_{2.v.cs}+ = \bigcup\limits_n (C_1.m_{1.v.cs}[n] \cap C_2)$
	for each $C_0.m_0$ in $C_1.m_{1.v.to}$	$\Rightarrow C_0.m_{0.v.cs}+ = C_1.m_{1.v.cs}$

Table 5.8: By algorithm: Post-bytecode analysis propagation

5.7 Whole Program Approach

A sound SA must account for classes loaded and methods invoked outside of the bytecode. This implementation attempted to keep user inputs to a minimum.

5.7.1 User Supplied inputs and Feedback

The simplest solution to implement depends on the user providing user supplied inputs from hand analysis introduced in sec. 5.1.5. Besides requiring extra work by the user, system methods can change as the system libraries and JVM further develop along with Java. This causes changes to hand inputs. Examples of changes experienced during implementation that affect user inputs are:

- various system libraries and versions,
- updates to CACAO JNI,
- updates to CACAO boot methods.

It did not seem reasonable to expect every user to supply the methods used during CACAO startup. So a Java program with just an empty *main* method (*null program*), (**In0**), is used to create a list of system methods always missed by the analysis and read in from the file a *xxxMissedIn0*, where *xxx* is *rt, xta,* or *vta* for the algorithm used. The instance class initiator, <*init*>, as input, indicates dynamically loaded classes.

class In0 { *public static void main(String[] s)* { } }

Figure 5.4: The null Java program, *In0.java*

RTA - no edges

RTA user inputs consist of just the class and method called.

Application inputs were supplied in the directory *rtIn* in a file with the name of the *main* class. If the *main* was in a package, then the directory structure must be supplied like for package source code. The *rtMissed* log of missed methods can aid a user creating and testing their input list of missed methods for their application. Even for user application, it may be system classes which cause methods to be missed.

XTA - with edges

For XTA and VTA, a CFG is built. XTA required both the *called* method and the *called-by* method as inputs. A method called from multiple methods might need to be included multiple times in the XTA user input. The test in JIT JVM parse could not catch a specific method call missed by XTA, since it only tests if the method has been analyzed at least once. For XTA reachable methods without a visible caller is supplied with an empty caller and treated as an entry method.

VTA - with edges

VTA would have required all variable references as inputs. VTA reverted back to RTA inputs since local variables in the bytecode cannot be mapped exactly to sources from outside in other languages.

To help the user with inputs, feedback of methods missed by an analysis are logged in a format easy to edit and use as input. However the caller information of the missed method is not available for XTA and VTA.

NATIVE methods

Initially *NATIVE* methods were hand analyzed and the results used when a native method was found to be reachable. However this approach was abandoned when it was found most native methods dynamically loaded classes or invoked methods specific to the application. Classes loaded and methods invoked via JNI are now included in the application specific user supplied inputs.

JNI methods may be included in the RTA mWL via *TA_NATIVE* flag.

5.7.2 Hybrid Pickup analysis - trivial solution

This implementation used HPA to completely avoid user inputs for a specific application run as explained in sec. 5.1.6. Because all methods are analyzed before any code is generated this hybrid analysis still performs a static analysis of a method before any code has been generated for the method. Type propagation proceeds dynamically during the run and is up-to-date to the current place in the application run.

In HPA mode these JVM invoked methods are seen at parse time as not yet analyzed, added to the mWL, and SA is performed until no more reachable methods are found. Using HPA mode makes the Java whole program challenges transparent to the user for SA.

5.7.3 Bytecode hints for dynamic invocation

Patterns in the bytecode provide some hints, but they are incomplete.

String **value hints** were not used. Use of (*Class.forName*), an *instanceOf* in the bytecode, and invocation of *method.invoke* as hints requires the value of the *String* parameters be known. However this is only known at runtime and is not available to SA and HPA. Since the DA here is basically a simulation, the information is not available for this implementation of DA.

The <***init***> method is always invoked when a class is instanciated whether by *new* or dynamically. Instantiation tells the analysis context, that this class' methods are reachable. However <*init*> is also invoked when a subclass is instantiated, requiring the super <*init*> test described in 5.2.2.

A hint that a class has been dynamically loaded is that the instance constructor, <*init*>, is *used* for a *not used* class and it is not invoked from a subclass initiator. When this occurs the class may be considered *used.*

Accuracy is important since adding just one method or class can add many more reachable methods, which will be discussed by example in the results chapter. If an instantiated class is not recognized all virtual invoked methods in the class are missed. However used wrongly this hint will cause all classes to be considered *used* causing RTA to reverts to CHA.

Using super <*init*> and other bytecode hints for dynamically loaded classes will be examined further in sec. 6.2.5. Initial tests favored its use to minimize the number of methods names read in from user supplied inputs.

5.7.4 Boot image and entry methods

Java programs do not start with invocation of the *main* method; instead the JVM directly invokes boot methods first. The **JVM's boot entry and exit**

methods are not reachable from *main*, not standardized by [LY99] and are JVM implementation dependent. They varied during implementation. These methods could be picked up via a HPA, but those methods required by any implementation are hard coded via a C include file for easy modification. CACAO requires the boot image be compiled each time.

The mWL is initialized with JVM's boot methods not reachable from *main* nor other boot methods:

- the first method to be parsed;
- *main* method;
- *java/lang/system/exit(int i)*.

Use of threads requires two more boot methods be added to the mWL :

- the instance constructor, *java/lang/Thread.<init> (java/lang/VMThread t, java/lang/String s, int i)* and
- *java/lang/ThreadGroup.addThread(java/lang/Thread t)*

for the default thread which CACAO thread initiator adds during the JVM boot.

5.8 Devirtualization statistics and tests

5.8.1 Statistics to gather

The CG built should be useful for devirtualization optimizations. When the program halts a post-processing analysis of the information gathered by the algorithm occurs (*TA_STATS*).

Information to compare algorithm analysis results include:

- spurious methods via number of methods analyzed vs. number of methods executed (jit used flag);
- class hierarchy via number of classes loaded with analysis vs. number of classes loaded without analysis (via separate runs) using *cacao –stat*;
- devirtualization statistics about *simple tests* (sec. 4.1) to determine how many more dynamically monomorphic the tested analysis algorithm could find.
- methods only determined to be dynamically monomorphic by the algorithm using the *1 method used* test (sec. 4.2).

Optionally a summary count (*TA_CNTONLY*) or the whole mWL, class hierarchy with method used flags (*TA_CLASSHEIRARCHY*), and for XTA and VTA, the mWL with sets information (*TA_SETS*) could be printed.

5.8.2 Devirtualization tests

Post-processing tested each method in the CG if it was monomorphic or polymorphic. The easiest test to determine, if the method monomorphic was recorded. Tests used were:

1. non-virtual (sec. 4.1)
   (a) language constructs (non-virtual);
   (b) no sub-classes exist (*leaf*);
   (c) *only 1* available definition of the method, m in $\widehat{C}$.
2. 1 method definition used in context of RTA, XTA, or VTA

To aid these tests the first bytecode to invoke the method was recorded during analysis. For system supplied, missed and picked-up methods the invoking method is unknown. For these methods the method names were examined for <*init*> and <*clinit*> and the method flags were examined for *STATIC* and *FINAL*. Similarly interface methods were examined to see if they were *FINAL*. Virtual methods were tested if they were *FINAL* when they were invoked.

The post-processing statistics *ta_stats* function is invoked after the *halt* method has run, if -stat *cacao* option is used. The option to the statistics function is hard-coded. The statistic function is called with *TA_STATS* including the mWL or totals only with *TA_CNTONLY*. The other options were used for more verbose information during testing (*TA_CLASSHEIRARCHY, TA_SETS*) to verify the the mWL, CFG and set information were correct.

Along with the name of methods in the mWL two flags were printed. A *J* indicates the method was actually executed by the JIT, making spurious methods easy to identify in the mWL. The kind of bytecode which invoked first the method is also printed. This flag was also used to gather statistics about whether language constructs were enough to identify a method as monomorphic. Totals were reported and compared for each devirtualization test two ways using by saving the bytecode invoked with and method flags. [1]

5.9 Testing

5.9.1 Algorithm invocation

The algorithms could be invoked one at a time using the *cacao* runtime options: *(–cha,) –rt, –xta, –vta*. The modes (SA, DA, HPA) are determined at compile time, so only one mode is available in a specific *cacao* binary. To ensure the binary for each mode was tested with the current code, test scripts built the executable binary. The script copied a file with the correct flag setting for the mode to be tested and then built the binary to be tested. The binary was copied to name matching the mode, so tests could be repeated with the same binary.

Initially only methods reachable from *main* were analyzed to be sure the analysis worked for smaller test cases with no output so no system methods were called. First in DA mode to be sure the code ran, then in SA mode.

[1] A method can be invoked by multiple *invoke*s - *invokeinterface, invokevirtual, invokespecial.*

Next *system* methods invoked via prints in a *hello world* program. Finally all methods reachable from all boot entry points were analyzed and the analysis checked for completeness by comparing SA results to HPA results. Methods called by the JVM were flagged and the analysis mWL had to contain all JVM invoked methods. The methods flagged as compiled by the JVM in SA and HPA were compared again against the DA results.

A *bytecode trace* in both the analysis and CACAO parse was used to see both opcodes used and the exact method invocations. The *-tracecall* option was used for XTA to hand check and verify the final user inputs.

5.9.2 Addresses, complete, exact

Addresses The most basic test is that the same bytecodes are analyzed and JIT parsed for a method. The address calculation in the analysis parse differed from the JIT parse because fewer bytecodes were parsed. So the bytecodes parsed by the algorithm were compared to the JIT via opcode traces. The classes loaded and invoked were also compared between the two.

Complete If the analysis missed a method it was logged to the Missed file for the algorithm. HPA and DA were very useful during initial testing for finding the minimum numbers of SA entry points, incremental testing of the invoke bytecodes, and debugging reachable methods.

Exactness A log of all methods JIT parsed was compared against the mWL and the *missed log.* For SA with user inputs all methods invoked by the JIT had to be in the mWL and the *rtMissed* file had to be empty for RTA. For XTA and VTA the *xxxMissed* file had to be empty for the NULL program, but no user inputs were supplied for individual programs except for RTA. [2] A program trace was used verify any method missed was due to a Native method, dynamic loading or invocation from the code, or reflection.

5.9.3 Unit Testing

Unit test cases first tested the non-virtual test (*static, special, virtual final*) described in section 4.1. Next methods invoked by *invokevirtual* were tested with test cases similiar to the section 4.2. Then similar test cases were used to test methods invoked by *invokeinterface.*

XTA needed additional test cases to test parameter passing in and return values out. Interface test cases were added where at least one parameter and the return value were variables with interface types.

For VTA the test cases were expanded to test first local variable assignments and then combinations of the various kinds "variables" as described in section 5.5. VTA test cases with branches to method calls and explicit exception handling via try-catch achieved stack misalignments at basic block boundaries.

5.9.4 System library test

The first test with system library methods test was empty main program, *In0* and then a *HelloWorld* like program, which included input/output system libraries dynamically invoked.

[2] XTA application inputs was abandoned when *GNU classpath* was being updated frequently, changing the user system library inputs.So HPA-SA was then used.

5.9.5 Mode testing

Pick-up mode with inputs should also have no Missed methods. Without inputs the Missed list should only contain entry methods as identified by a program call trace. *Dynamic mode* should match the executed methods exactly. All methods analyzed are identified as Missed since the analysis is for one method only at the beginning of parse. So the missed list and mWL should match.

5.9.6 Benchmark Testing

SPECjvm98 invokes each benchmark program application *main* dynamically, so all benchmarks needed at least their *main* as part their *Ins* input. The program class invoked is always the same, so the inputs needed by each method were copied into the *Ins* file for the benchmark *main* by a script. A script executed the algorithm for all benchmarks, plus the Null program and another script gathered the statistics.

Mode flags were kept in an C source code include. The test script copied a file with flags for the mode to be tested into this C include file and built the *cacao* executable. So the most current source code was always tested with the appropriate mode.

5.9.7 Devirtualization

Totals were reported and compared for each devirtualization test two ways using the first bytecode invoked with and method flags. The total of language constructs (non-virtual) must match the total of *static, special, final,* and constructors methods. The virtual total must match the total of the virtual and interface invokes without *final* methods.

The *unknown* total reported is the sum of the user supplied and default methods. Optionally the unknown bytecode inferred by using the method flags. However interface methods cannot be detected after they have been resolved during execution.

Differing sums from calculating the devirtualization total two ways uncovered an interface method which invoked as an interface in the actual application run, but in a reachable method is invoked via *invokevirtual.* A interface method may be invoked via an instance of the implementing class, not just via the interface class. Also a *virtual* method may be invoked via *invokespecial*, if called in source code using *super.*

Chapter 6

Experimental Results

This chapter reports the test results of the RTA, XTA and VTA algorithms against requirements given in section 2.1 and each other. Early tests that affected design and implementation are noted.

JIT refers to the CACAO JVM, which is a JIT compiler and indicates the compiler environment of the analysis. HPA-SA is sometimes referred to as just HPA. Any result of the SPECjvm98 benchmark tests are reported together are their geometric mean. CHA results are used as a maximum solution and JIT results are used as the minimum solution for comparison of how many classes were loaded and how many methods are found reachable.

6.1 Environment Factors

When analyzing the test results, it's important to note that, depending on the programming language they are written in, the JVM and Java system libraries become a part of the test case. The boot loading of the JVM is not standard, so it also affects the analysis. The results of various studies cannot be directly compared unless they use the same JVM and Java system libraries or ignore all system calls.

6.1.1 JVM implementation language

If the JVM is written in Java, then the JVM and its programming characteristics and style will become a part of the test case results. The number of classes loaded will be greater than when testing with a JVM not written in Java. Both Jikes and Sable are written in Java. Jikes optimizes itself beforehand [TP00], so there is no analysis of the Jikes code although it is written in Java. The CACAO JIT JVM is written in C, so all the benchmarks should load fewer total classes than reported in Jikes [TP00] and Sable [SHR$^+$00] articles.

6.1.2 Java System class libraries

Java's system class libraries are mostly written in Java. The parts of the system libraries not written in Java have use to the JNI, which hides what types and fields a method uses. [TP00] excluded the system libraries in their analysis results. [SHR$^+$00] used system libraries, but also reported results without libraries. This work used GNU Classpath Java libraries version 0.14, but has been run recently with version 97.2 also.

The benchmark program tends to be the smallest part of an application run. Including the system libraries in the analysis tests will include more Java code in the tests. The test results will also indicate how polymorphic the system libraries methods used by the benchmarks are.

6.1.3 Null Program analysis results

The question is open to whether including system methods in the analysis delivers a performance enhancement due to potential optimizations or a performance degradation because it brings unnecessary overhead.

The results for CACAO 0.92 using GNU classpath 0.14 analyzing a null Java program (fig.5.4) consisting of an empty do-nothing *main* method running was that 194 classes are loaded and 342 methods are called.[1] Of these 60 (17%) methods were invoked via *invokevirtual* and 18 (3%) methods via *invokeinterface* of which 7 interfaces methods were declared *final.* Most methods could be proven to be monomorphic via simple leaf (50) and 1 definition tests (18). Only 1 additional method is actually proven monomorphic by the analysis over the simple tests. Only 2 methods remained as possibly polymorphic.

Type analysis algorithms have been shown to be very useful for pre-processing bytecode analysis and significantly reducing jar file sizes. However spurious methods in a pre-processor inflate the results and the algorithms were not compared to the traditional simple tests. The CGs created are useful for other optimizations.

Choosing the RTA mWL as a worse case example in HPA-SA mode with no user supplied inputs for In0, there are 974 methods analyzed. Of these 288 (30%) methods were invoked via *invokevirtual* and 131 (13%) methods via *invokeinterface.* For SA with inputs the results are slightly more inflated. There are 1000 methods analyzed and 146 (15%) methods via *invokeinterface.* The results per mode and algorithm will be compared for benchmarks.

6.2 Whole program issues

The SA algorithm premise that starting with *main*, all invoked methods are reachable from the bytecode is invalid for Java, due to boot methods and dynamic method invocation. If the SA is incomplete, then the analysis may be wrong. Decisions made by a SA for an ahead of time compiler cannot be backed out.[2] This section explains the tests used to prove how complete the analyzes were.

6.2.1 Completeness

For reachability the log of methods missed by the initial SA pass was checked.

- For SA (sec.5.1) no methods should be missed.

[1] With GNU classpath version 97.2, a null program uses 435 methods.

[2] Analysis algorithms are <u>not</u> sound if not complete in the face of whole program issues.

- For HPA (sec.5.1) all methods in the missed method list must be invoked dynamically.
- For DA (sec.5.1) the missed list is the list of all methods executed.

There is one mismatch with the CACAO JIT that still exists. In dynamic mode the analysis lists 3 fewer methods than processed by JIT. These 3 methods (list methods) were called by the JIT for the bootstrap loader and the regular loader since a class is identified by the class loader and the class ([LY99]). However these 3 methods were only JIT parsed and compiled once, but CACAO statistics counted them twice.

6.2.2 CACAO boot initialization

A modified trace identified the 8 boot methods used by CACAO JIT boot initialization, after it stabilized. While CACAO booting was stabilizing, HPA mode was used. It was observed that all CACAO JIT boot invoked methods did not have be to explicitly invoked. Section 5.7.4 explains the four boot entry and exit methods expected.

The first method to be invoked (*String* class initiator) [3] starts the analysis, so it did not have to be hard coded. The remaining 3 boot invoked methods, dynamic loading methods, were found reachable by the previous 5 methods and added to the mWL by the analysis without being hardcoded.

6.2.3 Feedback Help

The log of methods missed by the SA of mWL from *main* and other initial start methods was used to test the completeness of SA and provide information necessary for user supplied inputs. The missed methods were analyzed to find the top level method missed, which caused other methods to be missed. It was important that <*init*> methods are in the CG, otherwise all the methods in that class were considered unreachable and also missed. However too many <*init*> methods can cause more spurious methods.

Table 6.2 Supplied Methods column 2 shows the number of user supplied methods for RTA (and VTA), the minimum inputs was a manageable 7-16 methods for a Java program with an empty *main*, which will be referred to as *In0*. VTA still had 14 picked-up methods for In0 since only missed methods were supplied and not variable references.

Five of the six supplied methods were called from Native methods and one method was due to reflection. The table shows the rtIns application specific inputs for each benchmark program. SA with no supplied inputs will miss these dynamically or JNI invoked methods and all methods they invoke. The *main* methods of the benchmarks are dynamically invoked, so all benchmarks must include their *main* method in user supplied inputs, at a minimum.

Most missed methods were due to the whole program issues in system methods. The benchmark *jess* misses more methods because it dynamically creates classes based on inputs, causing their <*init*> to not be seen by the analysis algorithms as reachable.

[3] It is interesting to note an early version CACAO did not call String class initiator before *main*, if *main* did not have any inputs. Since *Thread.addGroup* has *String* as a parameter, this is no longer possible.

Boot Methods	rtaIn0	xtaIn0	vtaIn0
4	6	16	6 (14)

Table 6.1: User Supplied Methods for JIT Boot

Algorithm	check	compress	db	jack	javac	jess	mpegaudio	mtrt
RTA	4	1	1	1	1	44	1	2
XTA	23	18	21	32	55	142	40	29
VTA	52	43	48	69	372	187	81	80

Table 6.2: Benchmark Methods Missed and picked-up

6.2.4 Class constructors and destructors

Class constructors can also be used as a secondary test, that all classes used by the JIT for an application were loaded. The number of JIT used class constructors in the mWL matched the number of class constructors that a non-analysis *-stat* CACAO JVM run of the application shows.

The analysis must add the destructor method, *finalize*, to the mWL if it exists. However *finalize* is only invoked by the JIT if garbage collection recovers the class instantiation. If the class instantiation is never recovered by the garbage collector, the *finalize* is a spurious method in the mWL. The *finalize* method was included unnecessarily 3 times from system classes used by the JVM boot. The benchmarks additionally used 3 system classes with *finalize* defined. *Jack*, *finalize* is defined for an application class. The garbage collector releases these object instantiations, so these *finalize* methods were invoked by the JIT. Fortunately *finalize* is not used often or the mWL could have many extra spurious methods.

6.2.5 Varying classes considered used

Each class loaded by the analysis could be taken as a hint that a class is used. To test the possible benefits RTA was tested where: casts, <*init*>s, *putstatic*, both *putstatic* and *gettstatic* and with all variations. The results are reported for the SPECjvm98 benchmarks in table 6.3. However the number of spurious methods grew and the number of missed methods did not decrease by more than one method. So the *parent init test* is not included in the end implementation.

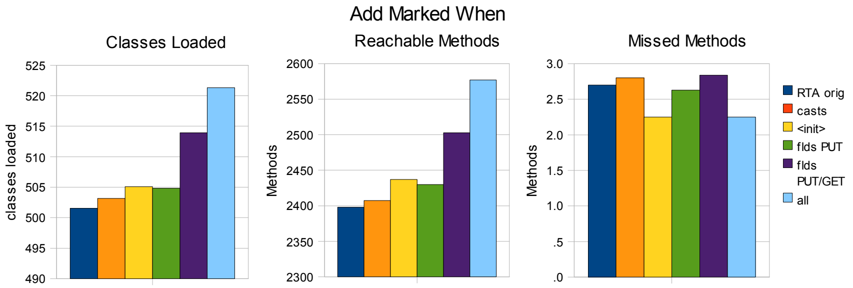


Figure 6.1: Effects of varying when a class is added to class hierarchy

Add Marked When	**RTA orig**	**casts**	<*init*>	**flds**	**flds**	**all**
	orig			***PUT***	***PUT/GET***	
classes loaded	502	503	505	505	514	521
reachable methods	2398	2407	2437	2430	2503	2577
missed methods	3	3	2	3	3	2

Table 6.3: Effects of varying when a class is added to class hierarchy

Leaving out the PARTUSED flag caused a geometric mean of 20 more methods to be RTA analyzed for the benchmark tests. For *In0* only 3 more methods were RTA analyzed.

The original RTA ignores fields completely. A static field could be used and the class constructor would not be added to the mWL. If the class constructor invoked other methods these will also be missed. In SPECjvm98, without adding the class constructor for *getstatic*, there was an improvement of 45 fewer methods analyzed, and 2-3 class constructors were missed and up to 6 more total missed methods. Further tests showed all class constructors missed were due to *getstatic* and the application missed methods due to *putstatic*.

The other option would be to include the missed <*CLINT*> methods in the user supplied inputs. Adding the class constructor when a static field is referenced was kept in the final implementation, since it was thought a method invoked by the JVM should not be a user supplied input.

6.2.6 Native Methods

Native methods are not included in the overall reachable methods statistics because:

- classes are loaded and methods called by native methods are not available to the analysis [HDH04];
- Java classes and methods invoked often differ between applications due to dynamic class loading and method invocation;
- user supplied inputs include classes loaded and methods invoked by native methods.

Native methods can be seen as reachable in the analysis parse via the method flag, *TA_NATIVE*, for RTA. For SPECjvm98, 80 Native methods were found reachable. Native methods are not JIT parsed, so are not flagged as missed. However examining a trace for *In0* found 21 native methods are actually invoked, but the analysis found 45 reachable native JNI methods.

6.3 Analysis Mode Comparison

	Mode: SA	***HPA***	***DA-HPA***	***DA***
User Input	**static**	**static pickup**	**dynamic mixed**	**dynamic**
Both=*wIns*	1 complete	5 complete	9 complete	10 complete
SysBoot=*In0*	2 incomplete	6 complete	9 complete	10 complete
App In=*rtIn*	3 too incomplete	7 complete	9 complete	10 complete
None=*noIns*	4 too incomplete	8 complete	9 complete	10 complete

Table 6.4: Summary of Modes

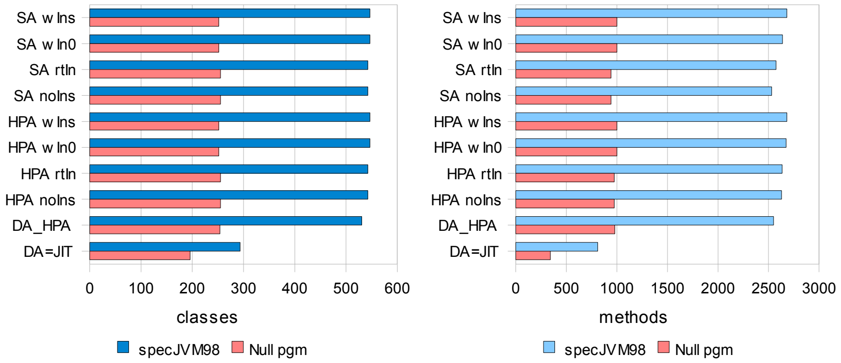


Figure 6.2: By mode: classes and methods analyzed (RTA)

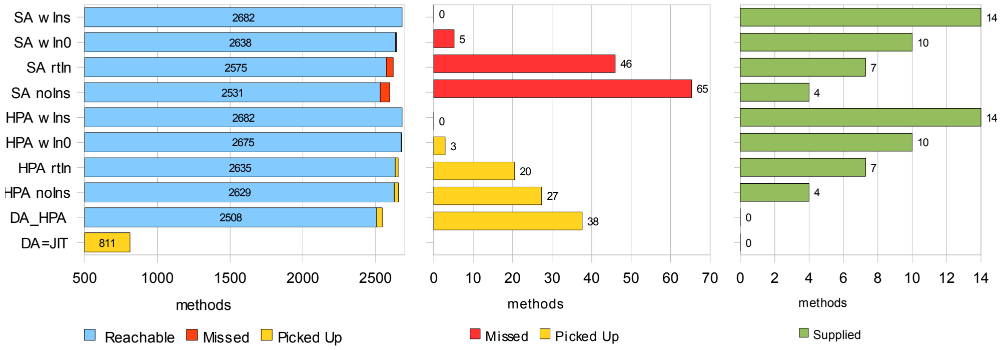


Figure 6.3: By mode: methods supplied, picked-up, missed (RTA)

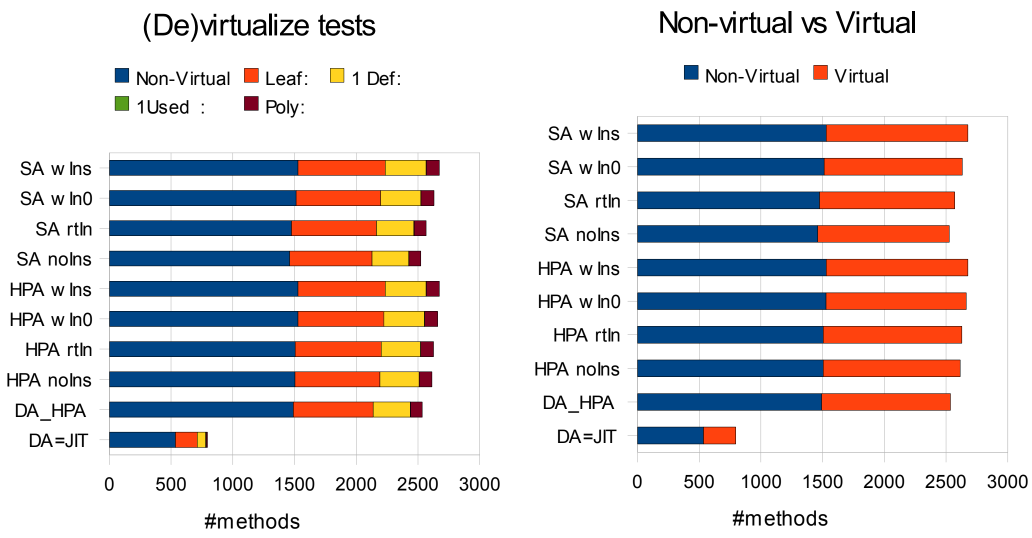


Figure 6.4: By mode: Virtual vs. Non-Virtual

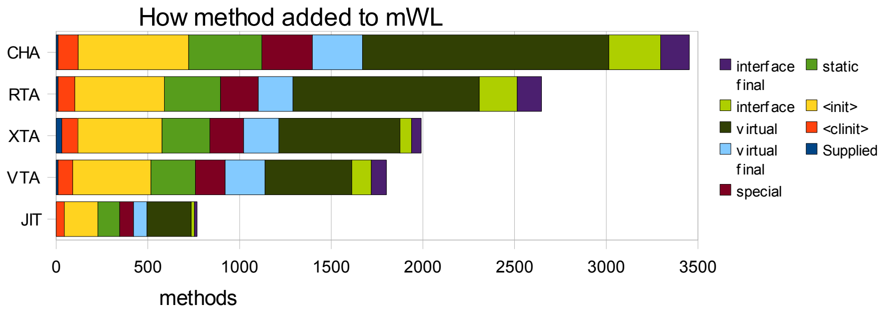


Figure 6.5: How method added to mWL

Ten combinations of the four modes (sec.5.1) and user supplied inputs were tested for RTA as shown in table 6.4. **In0** refers to the user supplied inputs (sec.5.7.1) for system boot in *rtMissedIn0* and **rtIn** to the user supplied inputs for the applications in *rtIn* directory. Each of the four modes could be tested with any of the four combinations of user inputs as shown in the table. Test results are shown in graphs in figures 6.2, 6.3 and 6.4 and tables B.1 and B.2.

Dynamic mode ignores inputs so all four HPA-DA and all four DA combinations are equivalent. SA with inputs and HPA-SA with inputs deliver the same results. Each mode has its applications, so no one mode is chosen. More likely comparing modes, especially the HPA introduced here, can help with a decision between a JIT and off-compiler or combination for an application.

DA and HPA-SA are used to report algorithm comparison. DA is equivalent to the JIT and is always reported. HPA-SA allows a form of SA without the bother of supplying user inputs if the application or system classes change.

6.3.1 Classes loaded

The graph in figure 6.2a shows the classes loaded for each mode compared against each other and *In0*. Figure 6.6 shows RTA supplied inputs and figure 6.7 shows a sample of XTA supplied inputs. *In0* gives an indication of how many of the classes loaded are system methods due to boot initialization. Within SA and HPA-SA modes the classes loaded only differ by 4 classes even when methods are missed or picked-up. HPA-DA loads 3% less classes than HPA-SA, but 45% more than the DA, which is equal to the JIT ideal.

In DA mode the system boot is two-thirds of the classes loaded, showing the overhead the system boot brings to the JIT for benchmarks.

```
gnu/java/io/encode/Encoder8859_1 <init> (Ljava/io/OutputStream;)V
```

Figure 6.6: Sample of supplied rtMissedIn0 inputs

```
java/lang/reflect/Constructor constructNative ([Ljava/lang/Object;Ljava/lang/Class;I)Ljava/lang/Object;
    gnu/java/io/encode/Encoder8859_1 <init> (Ljava/io/OutputStream;)V
```

Figure 6.7: Sample supplied xtaMissedIn0 input (calledBy / called)

6.3.2 Methods found reachable

The graphs in figures 6.2b and 6.3 and tables B.1 and B.2 report test results for methods in all 10 combinations. Figure 6.4b compares the number of total methods found reachable by each mode against the system boot initialization. Figure 6.3a shows the total number of methods analyzed by mode and what part of the methods come from analysis, user inputs or are picked-up. Figure 6.3b compares the number of methods *supplied* in SA and number of method *picked-up* in HPA closer since the totals are small compared to the total number of methods. No methods are found reachable by the DA so it is not included, all methods analyzed are picked-up in JIT parse for analysis. Figure 6.3c shows the methods supplied for analysis. The 4 hard coded boot methods are included in the supplied method totals.

As expected, the more supplied inputs the more methods are found reachable. The class of a supplied method is considered *used* causing more spurious methods to be reachable.

For SA, missing methods means the analysis is invalid (combinations 2,3,4). However combination 2 is still small enough to be useful to help a user create their SA user supplied inputs. Combinations 3 and 4 will have methods which are not entry points in their missed files, but called by entry points. Making non-entry point method's classes reachable will cause even more spurious methods to be analyzed and should be avoided.

The graph in figure 6.5 shows how methods were added to mWL. This is also an indication of devirtualization possibilities by how many *invokevirtuals* and *invokeinterfaces* were used, as well as what adds the most spurious methods.

6.3.3 Methods devirtualization

The graph in figure 6.5 shows how methods were added to mWL. This is also an indication of devirtualization possibilities by how many *invokevirtuals* and *invokeinterfaces* were used, as well as what adds the most spurious methods. The graph in figure 6.4a shows how many methods are non-virtual due to language constructs. The graph in figure 6.4b shows which test proves a method to be monomorphic. Methods which are monomorphic due to language constructs are also leaf methods. HPA-DA not only has the fewest methods analyzed, but also fewer virtual methods.

6.4 Algorithm Precision Comparison

The algorithms precision matched the progression of the program entity the types were collected on. The algorithm precision for classes and methods is compared in figure 6.8 for the individual SPECjvm98 benchmarks. CHA varies more wildly than the IPTA algorithms. The difference between the other algorithms is similar and would be predictable. If all classes in most branches of the class hierarchy are used, then RTA precision will be the same as CHA.

The SPECjvm98 combined class (fig.6.9) and method (fig.6.10) precision comparison is presented in three graphs: as the **totals** in first graph (a); **spurious** classes and methods in the second graph (b); and the **percent savings** over the next closest algorithm in third graph (c).

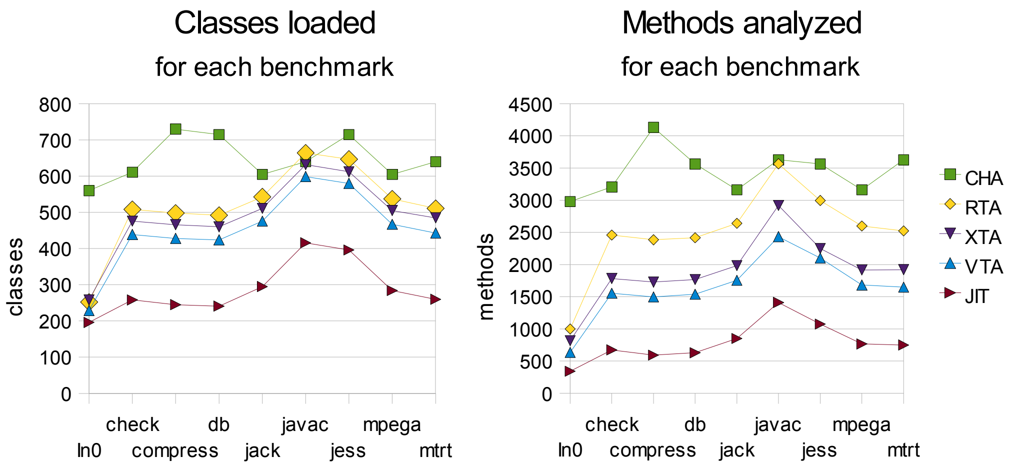


Figure 6.8: By algorithm/benchmark: Methods Analyzed and Classes loaded

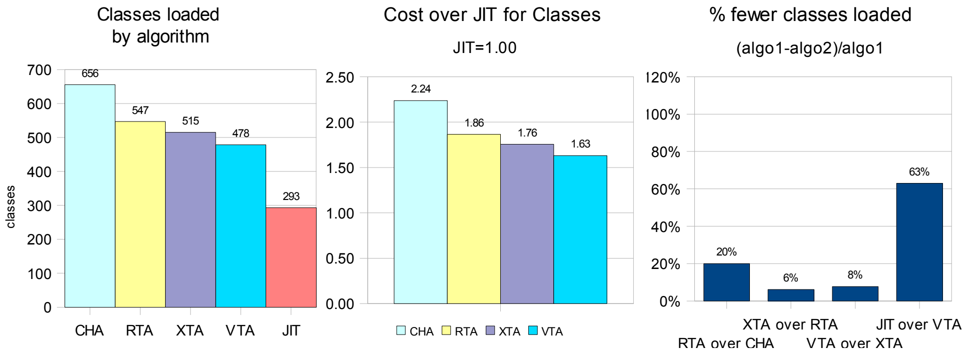


Figure 6.9: Classes loaded, Cost over JIT, and Savings over next algorithm

The classes loaded are only 1.6 to 2.2 times more than without IPTA. More classes loaded make more methods reachable. This causes a higher cost for the analyzed reachable methods, which vary from 2.2 to 4.3 times more than with no analysis. The variations presented in section 6.2.5 would fit between RTA and CHA.

Savings over the next closed algorithm is measured by $(algo1 - algo2)/algo1$. Savings for classes analyzed is:

- RTA had a 20% saving over CHA
- XTA had a 6% saving over RTA
- VTA had a 8% saving over XTA
- JIT JVM had a 63% saving over VTA

Savings for methods analyzed is:

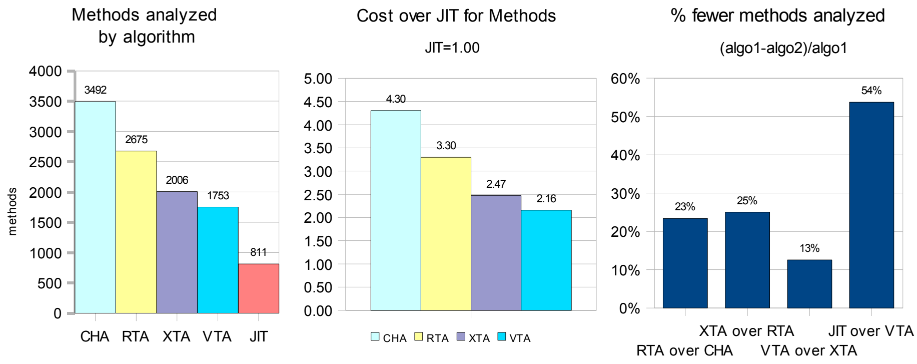


Figure 6.10: Methods analyzed, Cost over JIT, and Savings over next algorithm

- RTA had a 23% saving over CHA;
- XTA had a 25% saving over RTA;
- VTA had a 13% saving over XTA;
- JIT JVM had a 54% saving over VTA.

The results emphasize how many more classes and methods a SA uses over both the JIT JVM with no analysis and a DA.

6.5 How virtual is SPECjvm98?

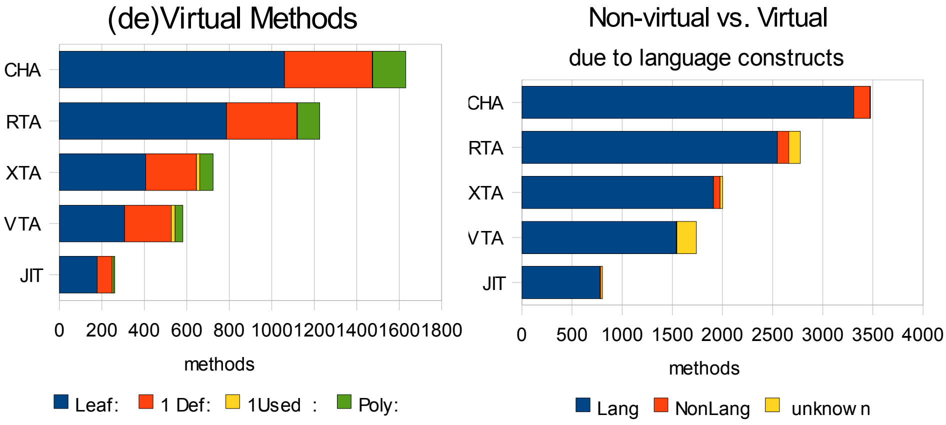


Figure 6.11: By algorithm: Virtual vs. Non-Virtual

As noted in section 3.6, SPECjvm98 is not very virtual, which explains why CHA often works almost as well as the more precise CG algorithms for DA. Only 1-2 methods were found to be monomorphic by the analysis algorithms over the traditional non-virtual *simple* tests (sec.4.1). But what are the characteristics of the SPECjvm98 method definitions that prove these methods are monomorphic? This question is answered in detail in the graphs in figure 6.11 and tables B.3 and B.4.

Method calls were broken-down into: system called methods; <*clinit*>; <*init*>; *invokestatic*; *invokespecial* (not <*init*>), which includes private and protected methods; *final* methods; simple virtual, which are dynamically monomorphic methods; *invokeinterface*; and default flag, which indicates a method was picked-up and caller is not known from the bytecode. Methods supplied to the system(SY), picked-up(DF), or application user supplied(MI) are not counted as monomorphic. All other methods categories, (<*clinit*> (CL), <*init*> (IN), static(ST), special(SP), virtual final(VF)), except the invokevirtual (VS) and invokeinterface (II) are known to be static due Java language constructs.

To summarize, 98% of the SPECjvm98 methods invoked are monomorphic. The largest part (82%) can be proven to be monomorphic through Java language constructs as being non-virtual. Of the 18% remaining methods, 90% are in a leaf class that was not declared *final* or the method has no methods definitions in a subclass. This left 2%(pickup SA)-0.1%(DA)1 of methods to need further examination by an IPTA to determine if the method is monomorphic.

Since SA and HPA analysis find more classes used and more methods reachable than are really in the JIT JVM, reporting only the analysis results skews the results. For example, 98% of methods in the benchmark with library methods are monomorphic. But for RTA and XTA, monomorphic reachable methods are respectively only 96% and 95%. The geometric mean of how many methods can be recognized through Java language constructs to be monomorphic is actually 66%, but 54% using RTA results.

6.6 Algorithm Costs

Algorithms costs vary in the number of bytecodes that must be parsed, the number and size of CG and CFG related sets, and the runtime or number of instructions run.

6.6.1 Parse Costs

One indication of how well an algorithm's runtime costs will size is how many bytecodes must be analyzed during the analysis parse[TP00]. Figure 6.12 and table 6.5 shows the number of bytecodes an algorithm must analyze during its parse compared to the JIT JVM parse.

The analysis bytecode parse may separated into bytecodes requiring only an address calculation and those analyzing the type flow information (sec.2.7). RTA and XTA must only analyze nine bytecodes. VTA must parse almost all bytecodes the JIT JVM must parse (116 vs. 131), which indicates VTA might not size well.

Some bytecodes require the same analysis and can be grouped together. The number of unique analyzes indicates the size of the code needed to implement the algorithm. RTA must uniquely analyze the invokes bytecodes. RTA must only simulate the class initiator invocation for other analyzed bytecodes(sections 5.2.3,6.2.4). XTA must uniquely analyze the class flow for static field references.

Algorithm	*Unique Analyses*	Bytecodes analyzed	Address calculation	Total
RTA	*5*	9	16	**25**
XTA	*6*	9	16	**25**
VTA	*44*	115	1	**116**
CACAO JIT	*55*	130	1	**131**

Table 6.5: Algorithm costs by number of bytecodes parsed

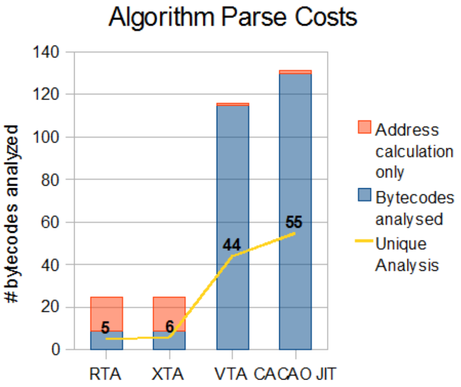


Figure 6.12: Algorithm Parse Costs png

Although there are 201 defined bytecodes, the CACAO JIT JVM first processes some bytecodes during stack analysis. VTA must analyze bytecodes for local variables with a reference type and perform a scaled down stack analysis. This reduced stack analysis can be viewed as extra cost when compared to RTA and XTA or savings over the SOOT VTA [SHR+00], which performs a full stack analysis when translating to Jimple.

6.6.2 Sets and Edges

The number of sets is stated in [TP00] as factor for the IPA's scalability. All 3 algorithms cost on the order of $O(n^3) = O(n^2) \times C$ where C is the number of classes in the program. Again VTA costs the most.

	n=#sets	**Description**
CHA	0	no sets all loaded classes are used
RTA	1	set of used classes by the program
XTA	$M + F + 1$	set of types used for each reachable method and field, plus 1 parameter class set
VTA	$2M + F(+P) + L$	set of types for each: reachable method(this+return) fields, (parameters) and local variables.

Table 6.6: Set Costs

VTA's extra memory requirements were evident. CACAO does an explicit garbage collection at the end of each major pass. VTA required an explicit recovery of dump memory after its analysis. RTA's and XTA's memory requirements fit into the parse phase, even for SA where the whole program is analyzed before the CACAO parse of the first method.

Table 6.6 summaries the class set costs for each algorithm. CHA has no explicit class set. Methods in loaded classes are considered reachable. RTA has one class set for the whole program, which is implemented by one flag in the class information in this implementation. XTA has class sets for each class used by all reachable methods with edges between the reachable methods. For type propagation to and from fields, XTA keeps a class set for each field in a method reachable by the analysis. One combined class set is kept for parameters. Return types are propagated via an intersection with the return type.

For VTA each variable has a class set. A method is a variable with a class set of what classes it is invoked with plus other method variables it uses (return, locals, fields). Parameters are mapped to local variables. So each parameter has its own class set. VTA uses (flowTo, flowFrom) edges between variables for type propagation. **Unless all reachable methods have no more than one parameter and no return variables, VTA has more class sets than XTA.** This holds true even for the original VTA implementation in [SHR$^+$00], where circular assignments caused variables to be combined within a method.

VTA propagation chains at the end of return were short with the majority less than 3 variables long. However in one case a propagation chain was 46 variables long!

6.6.3 Instructions Used

Using *perfex* profiler ([SGI],[Ert]) to find the number of instructions used with HPA, XTA always used the fewest instructions. However VTA was second 5 times and RTA 3 times and almost the same once for the 9 test cases (see table 6.7 and fig.6.7). The geometric means show:

- XTA uses 14% more instructions than the cacao JVM with no analysis
- RTA uses 13% more instructions than XTA
- VTA uses 1.5% more instructions than RTA

The instruction count standard deviation was zero except for *mtrt.* The instruction count standard deviation for *mtrt*, which uses threads, was between 0.5% and 1.5% of the total instructions. For *Null* program and *check* the instruction counts were close and in some cases less than the JVM without analysis.

Instructions	JIT	RTA	XTA	VTA	Fewest
In0	181	277	225	266	XTA
check	283	721	472	706	XTA
compress	14718	15173	14980	15172	XTA
db	21438	22014	21920	22035	XTA
jack	9191	9680	9506	9742	XTA
javac	18623	18623	17974	21767	XTA
jess	11358	12233	11616	13134	XTA
mpegaudio	19142	19570	19324	19562	XTA
mtrt	230	666	420	643	XTA
geomean	3726	4990	4354	5066	XTA

Table 6.7: By algorithm (HPA): Comparison of the number of instructions in millions

DA compares how the algorithms compare when the algorithms analyze the same number of methods with the same class hierarchy. However with DA, XTA always used the most instructions. RTA used the fewest instructions except for *jess* (see table 6.8 and fig.6.8). For the smaller programs the difference was more

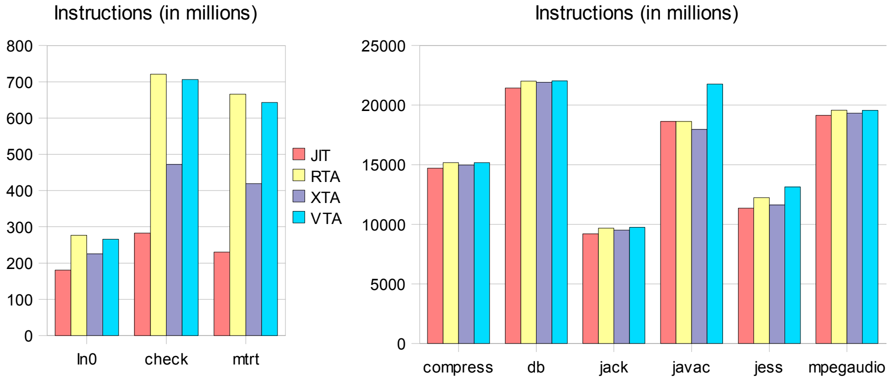


Figure 6.13: Algorithm Instruction Costs - HPA

noticeable. VTA had a propagation problem with *mtrt* due to not all classes loaded when needed as expected sometimes for DA.

Instructions	JIT	RTA	XTA	VTA	Fewest
In0	181	206	225	222	RTA
check	283	337	472	365	RTA
compress	14718	14772	14980	14814	RTA
db	21438	21489	21920	21694	RTA
jack	9191	9249	9532	9297	RTA
javac	17681	17801	18707	18517	RTA
jess	11358	11493	11795	11380	VTA
mpegaudio	19142	19194	19324	19228	RTA
mtrt	230	273	420	269	VTA
geomean	3726	3919	4382	4002	RTA

Table 6.8: By algorithm (DA): Comparison of the number of instructions in millions

6.7 CACAO inlining

Initially the CG algorithms tested were for devirtualization for use with inlining in CACAO. CACAO has tested two inlining implementations. After examining the two inlining implementations, the usefulness of IPTA for inlining in CACAO will be discussed.

6.7.1 Inlining in parse

The first attempt at inlining in CACAO was implemented as part of the parse phase. The heuristic used to select methods to inline was the first n methods, which were smaller than a specific size, determined by the number of bytecodes a method had. This first implementation of inlining was also tested with the SA implementations of RTA and XTA. No dynamically monomorphic methods were inlined by any of the JVM benchmarks. With or without IPTA analysis, this implementation of inlining was always slower than without inlining. The heuristic was picking non-critical methods to inline and reaching its limit for the number of methods to inline before finding any dynamically monomorphic virtual method to inline.

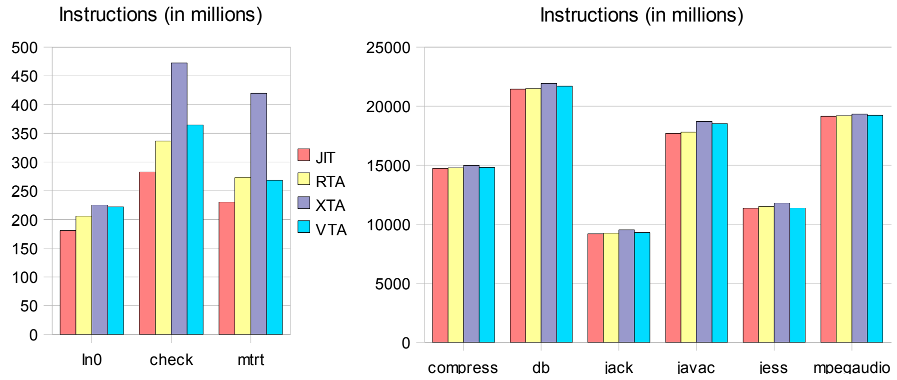


Figure 6.14: Algorithm Instruction Costs - DA

6.7.2 Inlining with online stack replacement

An important improvement in the current implementation of inlining [Ste07] in CACAO is finding critical methods with a profiler. No IPTA was used. If there is only one method in the dynamic dispatch table, then it is considered dynamically monomorphic and a candidate to be inlined. The information available to the JVM is the same as dynamic CHA test with the traditional simple tests (see Chap.4). Since a method has been executed often enough to be a hot method, the probability is higher, if a method is polymorphic, it is known when making the inline decision. Wrong decisions were corrected with online stack replacement.

This version of inlining with no IPTA did bring runtime savings with SPEC jvm98 benchmark programs. The SPECjvm98 characteristics in section 6.11 show that few methods, require a more extensive analysis to prove they are dynamically monomorphic.

6.7.3 Inlining with dynamic inter-procedural type analysis

The test results of inlining in CACAO agree with [LYK$^+$00] that when using a Virtual Method Table, like CACAO uses, an adaptive optimization framework using a hot-spot profiler is needed to avoid performance degradation.

Although IPTA can be used by other optimizations and IPTA is needed for type analysis, it's doubtful IPTA can improve inlining CACAO performance. Further performance improvement in CACAO due to inlining in SPECjvm98 cannot be expected due to XTA and VTA. The analysis of SPECjvm98 in section 6.5, shows most virtual and interface methods invoked can be proven to be monomorphic using traditional *simple* tests (sec.4.1) as documented by [Dea96].

Currently CACAO 99.2 explicitly uses the *language construct test* for its leaf test. Implicitly the JVM compiler sees when only one method definition has been used so far via the VMT. This is different from the second and third simple tests. Checking for *no subclass* proves the method cannot currently be overridden and is fast. No check is made if *only one method definition exists*, however this test and IPTA (RTA, XTA, VTA) tests are more costly. The VMT test used shows a method is currently monomorphic. CHA class and method costs are less predictable from the actual number used than the IPTA algorithms(fig. 6.8). Using IPTA tests would show if an inlined method might have to be backed out, but not if it will be backed-out.

Chapter 7

Summary

7.1 Modes and class loading

DA in the JVM is appropriate for an online JVM compiler such as CACAO JVM JIT. The analysis information can be collected at runtime, but only used on-demand ([QH05]). Only DA mode is compatible with lazy loading in an online JVM. Both SA and HPA modes must load classes, before its known if the class is really needed by the application run.

SA and HPA in the JVM are appropriate for use in an off-line compiler or for test and verification applications. Offline applications still need the smallest set of reachable methods, since spurious classes and methods decrease analysis precision. On-demand class loading limits the class hierarchy to reachable classes.

The differences found between this implementation and traditional pre-processing SA and other hybrid analysis implementations (fig.s 7.1, 3.1, 5.1) are:

- in the JVM so all IPTA information not just hints are available for optimizations;
- smaller class hierarchy by using on-demand class loading instead preloading all classes in the classfile constant pool;
- no separate trace run needed for test tools;
- HPA picks-up all entry methods without user supplied inputs;
- HPA starts a SA analysis immediately from an entry method; [1]
- DA mode uses the ideal CG;

For SPECjvm98 most methods are monomorphic, so few corrections due to DA having incomplete information would be expected. More generally, use of the DA information by an optimizer can be improved by using characteristics of the application to delay optimizations to a logical time when more information will be known. (see sec.s 6.7, 3.4)

[1] Since this SA is finished before the method's parse, the results may be used by optimizations of all methods reachable from this entry method.

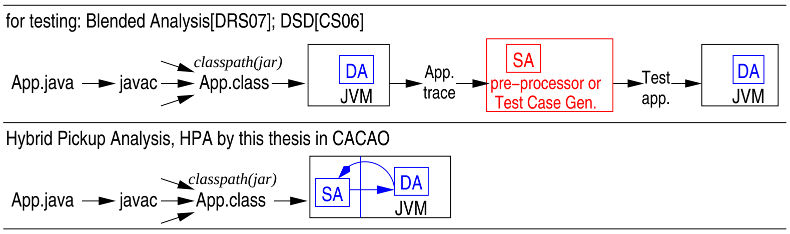


Figure 7.1: HPA combines the 3 stages of DSD / Blended analysis into 1 step

7.2 Algorithm Comparison

7.2.1 CHA and simple tests enough?

[QH05] found it surprising for DA, that dynamic CHA found dynamic monomorphic virtual methods almost as well as dynamic VTA. It is not surprising, since unlike SA, DA has a more precise class hierarchy with no spurious classes loaded. CHA precision is determined by the class loading scheme, whether DA or SA. Methods definitions visible varied more wildly and making it be less easy to generalize its costs and precision, section6.3.2. More precise algorithms have fewer spurious methods to mark.

CHA will perform best with the smallest possible class hierarchy, as with lazy loading and DA. However CHA is not precise enough to find dynamically monomorphic interface method calls from both [QH05] and the tests here. But a more expensive analysis is only needed if the program has hot methods, which are interfaces. This is not true of SPECjvm98 from the tests here nor SPEC2000 and other standard Java benchmarks [QH05].

The tests here showed most singleton method definitions can be found using leaf tests (90%). Only 2% for Pickup SA and 0.1% for DA of the SPECjvm98 methods require more extensive IPTA like RTA, XTA and VTA to determine if they are dynamically monomorphic. **Showing leaf tests with CHA is enough for SPECjvm98.**

None of the algorithms found more than 2 method callsites, which only used 1 definition. The rest could be proven to be monomorphic using one of the non-virtual tests.

7.2.2 Costs

XTA used fewer instructions, but VTA was second 5 times and RTA 3 times. Less detailed information gathered caused more spurious methods to be analyzed. More detailed information gathered, gained back time by analyzing fewer methods. VTA can distinguish better between types within a method.

As the precision of the entity the class set is associated with increased(table 2.2), the number of spurious classes and methods decreased (sec.6.4).

Spurious classes/methods VTA < XTA < RTA < CHA

As the precision of the entity the class set is associated with increased, the number of sets needed and bytecodes analyzed increases. (sec. 6.6).

Precision /Implementation costs VTA > XTA > RTA > CHA

When spurious classes and methods are in the analysis results, the effectiveness of the algorithms is inflated. The runtime tests suggest, that gathering the type information for an analysis is often small compared to the total runtime and might be able to be run on demand, if a profiler deemed the analysis information to be useful.

7.3 An algorithm for Dynamic Analysis?

A DA is not always needed. More expensive CG IPTA algorithms are only needed, if multiple methods called at the program level being used (examples: graphic program, visitor pattern). The more precise the class hierarchy is the fewer false positive for used classes and reachable methods.

Each of the algorithms fits a specific style best as shown in Chapter 4. If the application is mostly monomorphic then traditional *simple* tests are enough. If the application has similar objects as described in section 1.1.2, then the extra precision RTA, XTA, and VTA bring is important enough to be worth their costs.

If there is a need, rather than choosing just one algorithm a framework would be suggested, set-up like Vortex ([GDDC97], [CDG96]) and Sable's current framework or at least multiple options would be suggested.

7.4 Future work

Promising areas for future work are:

- monomorphic statistics for CACAO boot with various classpath libraries,
- gather DA information during required type analysis,
- the use of IPTA for other optimizations perhaps in a framework to allow testing of various IPTA algorithms;
- to recognize the patterns in the hotspot methods which implement multiple interfaces, which might be monomorphic;
- the use of CACAO generated information for program verification.

HPA mode with delayed class loading looks promising for application with offline compilers and test tools.

Bibliography

[AAB$^+$05] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocch, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research VM project: building an open-source research community. *IBM Sys. Journal*, 44(2):399–417, 2005.

[AH95] O. Agesen and U. Hölzle. Type feedback vs. concrete type inference: a comparison of optimization techniques for OO languages. In *OOPSLA '95: Proc. of the 10th annual Conf. on OOP systems, languages, and applications*, pages 91–107, NY, NY, USA, 1995. ACM.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language, Ph.D. Dissertation.* PhD thesis, DIKU, Univ. of Copenhagen, 1994.

[BGH$^+$07] Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, and Koen De Bosschere. Using hpm-sampling to drive dynamic compilation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 553–568, New York, NY, USA, 2007. ACM.

[Bin07] D. Binkley. Source code analysis: A road map. In *FOSE '07: 2007 Future of SWE*, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Soc.

[BMA03] P. Brunelle, E. Merlo, and G. Antoniol. Investigating Java type analyses for the receiver-classes testing criterion. In *ISSRE '03: Proc. of the 14th Int. Symp.on Software Reliability Engineering*, page 419, Washington, DC, USA, 2003. IEEE Computer Soc.

[BS96] D.F. Bacon and P.F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96: Proc. of the 11th ACM SIGPLAN Conf. on OOP, systems, languages, and applications*, pages 324–341, NY, NY, USA, 1996. ACM.

[CDG96] C. Chambers, J. Dean, and D. Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical Report TR-96-11-02, University of Washington, 1996.

[Cha01] S. Chang. Performance profiling and optimization on the sgi origins. Technical Report MS 258-6, NASA Ames Research Center, June 2001. http://people.nas.nasa.gov/˜schang/origin_opt.pdf, accessed Aug.2008.

[CHK02] K. D. Cooper, T. J. Harvey, and K. Kennedy. Iterative dataflow analysis, revisited. Technical Report TR04-100, Rice Tech. Report, Nov. 2002.

[CS06] C. Csallner and Y. Smaragdakis. Dsd-crasher: a hybrid analysis tool for bug finding. In *ISSTA '06: Proc. of the 2006 Int. Symp.on Software testing and analysis*, pages 245–254, NY, NY, USA, 2006. ACM.

[Dea96] J. Dean. *Whole-Program Optimization of OO Languages.* PhD thesis, Univ. of Washington, Nov. 1996.

[DGC95] J. Dean, D. Grove, and C. Chambers. Optimization of OOP using static class hierarchy analysis. In *ECOOP '95: Proc. of the 9th European Conf. on OOP*, pages 77–101, London, UK, 1995. Springer-Verlag.

[DRS07] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA '07: Proc. of the 2007 Int. Symp.on Software testing and analysis*, pages 118–128, NY, NY, USA, 2007. ACM.

[Ern03] M. Ernst. Static and dynamic analysis: synergy and duality. In *ICSE Workshop on Dynamic Analysis (WODA)*, pages 118–128, May 2003.

[Ert] Anton Ertl. http://www.complang.tuwien.ac.at/anton/lvas/skriptum-effizienz.html, accessed Aug.2008.

[FKU75] A. Fong, J. Kam, and J. Ullman. Application of lattice algebra to loop optimization. In *POPL '75: Proc. of the 2nd ACM SIGACT-SIGPLAN Symp.on Principles of programming languages*, pages 1–9, NY, NY, USA, 1975. ACM.

[Fol07] S. Foley. Tactics for minimal interference from class loading in real-time Java˙ In *JTRES'07 Proc.*, pages 23–32, NY, NY, USA, 2007. ACM.

[GC01] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685746, 2001.

[GDDC97] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in OO languages. In *OOPSLA '97: Proc. of the 12th ACM SIGPLAN Conf. on OOP, systems, languages, and applications*, pages 108–124, NY, NY, USA, 1997. ACM Press.

[Gea74] C. W. Gear. *Computer Organization and Programming.* McGraw-Hill C.S. series, 1974.

[HDDH07] M. Hirzel, D. Von Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29(2):11, 2007.

[HDH04] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in presence of dynamic class loading. *ECOOP04*, 2004.

[Hin01] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01).* ACM Press, Jun. 2001. http://www.infosun.fmi.uni-passau.de/st/paste01.

[Hir04] M. Hirzel. *Connectivity-Based Garbage Collection.* PhD thesis, Dept. of C. S., Univ.of Colorado at Boulder, Jul. 2004. http://domino.research.ibm.com/comm/research_people.nsf/pages/hirzel.index.html/hirzel.index.html/FILE/dissertation.pdf, accessed Aug. 2006.

[HvDDH05] M. Hirzel, D. von Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. Technical Report RC23638, IBM Research Report, Watson, Jun. 2005.

[IKY+99] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Java '99: Proc. of the ACM 1999 Conf. on Java Grande*, pages 119–128, NY, NY, USA, 1999. ACM.

[KC07] R. Kumar and S. S. Chakraborty. Precise static type analysis for OOP. *SIGPLAN Not.*, 42(2):17–26, 2007.

[KH02] A. Krall and N. Horspool. Optimizations and machine code generation. *The Compiler Design Handbook*, pages 219–246, Sept. 2002.

[Kil73] G. A. Kildall. A unified approach to global program optimization. In *POPL '73: Proc. of the 1st annual ACM SIGACT-SIGPLAN Symp.on Principles of programming languages*, pages 194–206, NY, NY, USA, 1973. ACM.

[Kra98] A. Krall. Efficient jvm just-in-time compilation. *Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 205–212, Oct. 1998.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC '92: Proc. of the 4th Int. Conf. on Compiler Construction*, pages 125–140, London, UK, 1992. Springer-Verlag.

[KU76] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, 1976.

[KVM00] White Paper KVM. J2me building blocks for mobile devices, white paper on kvm and the cldc, May 2000. http://java.sun.com/ products/cldc/wp/KVMwp.pdf.

[LAHC06] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online performance auditing: using hot optimizations without getting burned. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 239–251, New York, NY, USA, 2006. ACM.

[LAM07] K. Lee, Q. Ali, and S. P. Midkiff. Efficient classloading strategies for interprocedural analyses in the presence of dynamic classloading. *Fifth Int. Workshop on Dynamic Analysis (WODA'07)*, 2007.

[LH06] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In A. Mycroft and A. Zeller, editors, *Compiler Construction, 15th Int. Conf.*, volume 3923 of *LNCS*, pages 47–64, Vienna, March 2006. Springer.

[Lho07] O. Lhoták. Comparing call graphs. In *PASTE '07: Proc. of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 37–42, NY, NY, USA, 2007. ACM Press.

[LWL05] B. Livshits, J. Whaley, and M. Lam. Reflection analysis for Java, 2005.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification,2nd Ed.* Sun Microsystems Inc., 1999. http://Java.sun.com/docs/books/ second_edition/html/VMSpecTOC.doc.html, accessed May 2007.

[LYK$^+$00] J. Lee, B. Yang, S. Kim, K. Ebcioğlu, E. Altman, S. Lee, Y. C. Chung, H. Lee, J. H. Lee, and S. Moon. Reducing virtual call overheads in a jvm just-in-time compiler. *SIGARCH Comput. Archit. News*, 28(1):21–33, 2000.

[Mil07] A. Milanova. Light context-sensitive points-to analysis for Java. In *PASTE '07: Proc. of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 25–30, NY, NY, USA, 2007. ACM.

[MRR02] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA '02: Proc. of the 2002 ACM SIGSOFT Int. Symp.on Software testing and analysis*, pages 1–11, NY, NY, USA, 2002. ACM.

[MRR05] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[Pro02] C. W. Probst. *A Demand Driven Solver for Constraint-Based Control Flow Analysis.* PhD thesis, Universität des Saarlandes, Oct. 2002.

[QH04] F. Qian and L. Hendren. Towards dynamic interprocedural analysis in jvms. In *JVM Research and Technology Symp.*, 2004. http://www.sable.mcgill.ca/ publications/papers/2004-3/ sable-paper-2004-3.pdf, accessed Aug.2006.

[QH05] F. Qian and L. Hendren. A study of type analysis for speculative method inlining in a jit environment. In *Lecture notes in Computer Science*, pages 255–270. Springer Berlin/Heidelberg, 2005.

[RK02] D. Rayside and K. Kontogiannis. A generic worklist algorithm for graph reachability problems in program analysis. In *CSMR '02: Proc. of the 6th European Conf. on Software Maintenance and Reengineering*, pages 67–76, Washington, DC, USA, 2002. IEEE Computer Soc.

[RMR03] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. In *ICSE '03: Proc. of the 25th Int. Conf. on Software Engineering*, pages 210–220, Washington, DC, USA, 2003. IEEE Computer Soc. http://www.cs.rpi.edu/˜milanova/docs/icse03.pdf, accessed Aug.2006.

[Ryd03] B. G. Ryder. Dimensions of precision in reference analysis of OOP languages. In *Proc. of the Int. Conf. on Compiler Construction*, pages 126–137, Berlin / Heidelberg, Apr. 2003. Springer-Verlag. http://www.prolangs.rutgers.edu/refs/docs/cc03.pdf, accessed Aug.2006.

[Ryd07] B. Ryder. Adv. program analyses for OO systems, acaces 2007: 3rd int. summer school on adv. computer arch. and compilation for embedded sys. adv. pgm. analyses for OO systems, Jul. 2007. http://RydJul.07 http://www.cs.rutgers.edu/˜ryder/, lectures 1-5, accessed Feb.2008.

[SGI] http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi ?coll=0650&db=man&fname=/usr/share/catman/u_man/cat1/perfex.z& srch=perfex, accessed Aug.2008.

[SGSB05] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. *SIGPLAN Not.*, 40(10):59–76, 2005.

[Shi91] O. Shivers. *Control-Flow Analysis of Higher-Order Languages, Ph.D. Dissertation.* PhD thesis, Carnegie Mellon Univ., May 1991.

[SHR+00] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *OOPSLA '00: Proc. of the 15th ACM SIGPLAN Conf. on OOP, systems, languages, and applications*, pages 264–280, NY, NY, USA, 2000. ACM Press.

[Sri92] A. Srivastava. Unreachable procedures in OOP. *ACM Lett. Program. Lang. Syst.*, 1(4):355–364, 1992.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proc. of the 23rd ACM SIGPLAN-SIGACT Symp.on Principles of programming languages*, pages 32–41, NY, NY, USA, 1996. ACM.

[Ste07] E. Steiner. Thesis: Adaptive inlining and on-stack replacement in a JVM, Feb. 2007.

[TLSS99] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *OOPSLA '99: Proc. of the 14th ACM SIGPLAN Conf. on OOP, systems, languages, and applications*, pages 292–305, NY, NY, USA, 1999. ACM Press.

[TP00] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA '00: Proc. of the 15th ACM SIGPLAN Conf. on OOP, systems, languages, and applications*, pages 281–293, NY, USA, 2000. ACM Press.

[TSL03] F. Tip, P. F. Sweeney, and C. Laffra. Extracting library-based Java applications. *Commun. ACM*, 46(8):35–40, 2003.

[VRCG+99] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proc. of the 1999 Conf. of the Centre for Adv. Studies on Collaborative research*, page 13. IBM Press, 1999.

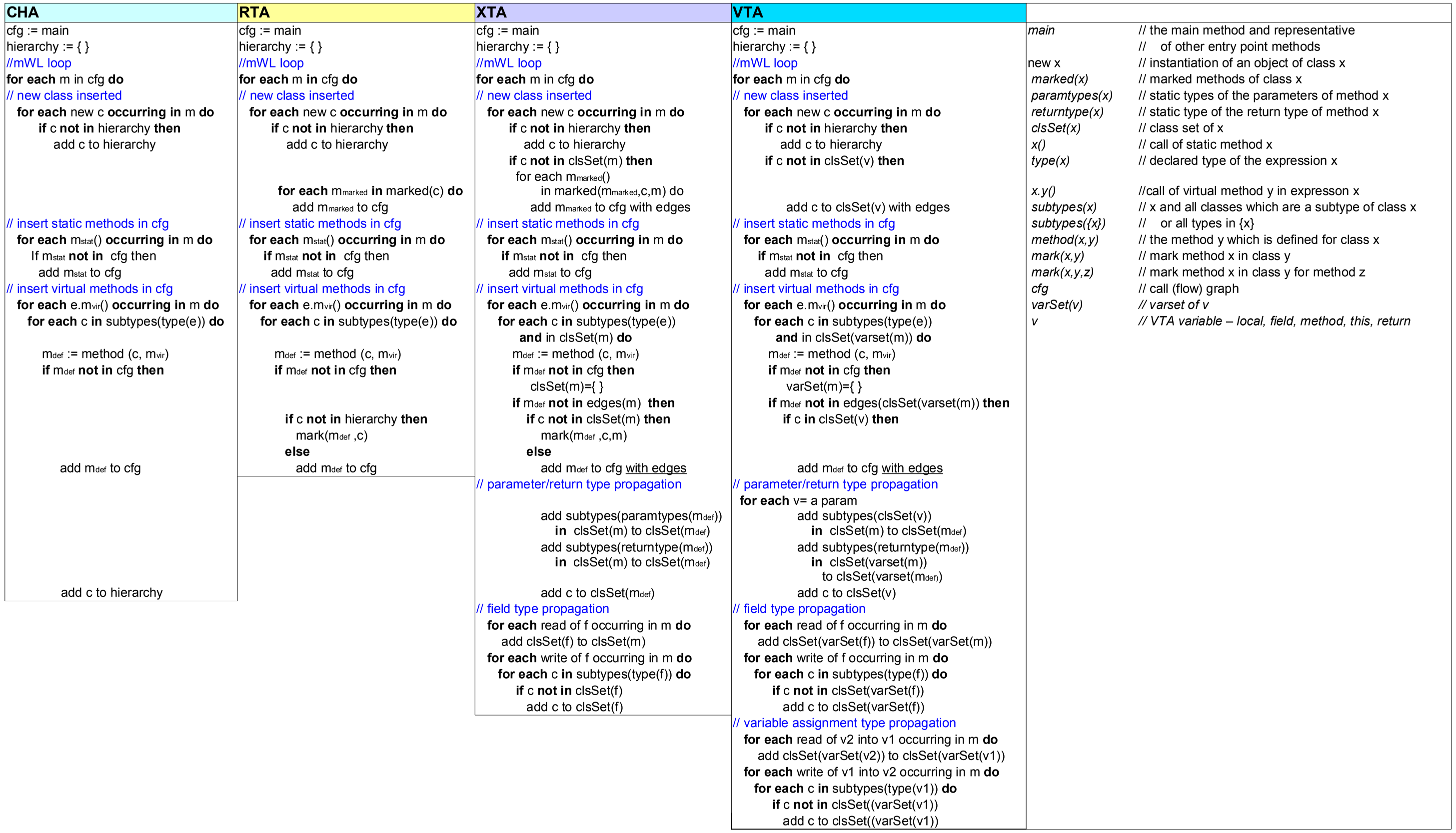
[ZR07] W. Zhang and B. G. Ryder. Discovering accurate interclass test dependences. In *PASTE '07: Proc. of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 55–62, NY, NY, USA, 2007. ACM.

Appendix A

Algorithm Comparison

Figure A.1 contains a high level comparison in pseudo code of the algorithms implemented. CHA and RTA are based on figures in [KH02]. Actions that are same or similiar are on the same line. So differences are easily seen additions or deletions by an algorithm are on different lines. Symbols are explained on the right in comments.

More details of the algorithm implementations are contained in chap. 5.



CHA

```
cfg := main
hierarchy := { }
//mWL loop
for each m in cfg do
// new class inserted
  for each new c occurring in m do
      if c not in hierarchy then
         add c to hierarchy



// insert static methods in cfg
  for each $m_{stat}$() occurring in m do
     If $m_{stat}$ not in  cfg then
      add $m_{stat}$ to cfg
// insert virtual methods in cfg
  for each e.$m_{vir}$() occurring in m do
    for each c in subtypes(type(e)) do

       $m_{def}$ := method (c, $m_{vir}$)
       if $m_{def}$ not in cfg then



          add $m_{def}$ to cfg







          add c to hierarchy
```

RTA

```
cfg := main
hierarchy := { }
//mWL loop
for each m in cfg do
// new class inserted
  for each new c occurring in m do
      if c not in hierarchy then
         add c to hierarchy


        for each $m_{marked}$ in marked(c) do
           add $m_{marked}$ to cfg
// insert static methods in cfg
  for each $m_{stat}$() occurring in m do
     if $m_{stat}$ not in  cfg then
      add $m_{stat}$ to cfg
// insert virtual methods in cfg
  for each e.$m_{vir}$() occurring in m do
    for each c in subtypes(type(e)) do

       $m_{def}$ := method (c, $m_{vir}$)
       if $m_{def}$ not in cfg then


         if c not in hierarchy then
           mark($m_{def}$ ,c)
         else
           add $m_{def}$ to cfg
```

XTA

```
cfg := main
hierarchy := { }
//mWL loop
for each m in cfg do
// new class inserted
  for each new c occurring in m do
      if c not in hierarchy then
         add c to hierarchy
      if c not in clsSet(m) then
        for each $m_{marked}$()
            in marked($m_{marked}$,c,m) do
          add $m_{marked}$ to cfg with edges
// insert static methods in cfg
  for each $m_{stat}$() occurring in m do
     if $m_{stat}$ not in  cfg then
      add $m_{stat}$ to cfg
// insert virtual methods in cfg
  for each e.$m_{vir}$() occurring in m do
    for each c in subtypes(type(e))
        and in clsSet(m) do
       $m_{def}$ := method (c, $m_{vir}$)
       if $m_{def}$ not in cfg then
          clsSet(m)={ }
       if $m_{def}$ not in edges(m)  then
          if c not in clsSet(m) then
            mark($m_{def}$ ,c,m)
          else
            add $m_{def}$ to cfg with edges
// parameter/return type propagation

            add subtypes(paramtypes($m_{def}$))
               in  clsSet(m) to clsSet($m_{def}$)
            add subtypes(returntype($m_{def}$))
               in  clsSet(m) to clsSet($m_{def}$)

            add c to clsSet($m_{def}$)
// field type propagation
  for each read of f occurring in m do
     add clsSet(f) to clsSet(m)
  for each write of f occurring in m do
    for each c in subtypes(type(f)) do
       if c not in clsSet(f)
         add c to clsSet(f)
```

VTA

```
cfg := main
hierarchy := { }
//mWL loop
for each m in cfg do
// new class inserted
  for each new c occurring in m do
      if c not in hierarchy then
         add c to hierarchy
      if c not in clsSet(v) then


           add c to clsSet(v) with edges
// insert static methods in cfg
  for each $m_{stat}$() occurring in m do
     if $m_{stat}$ not in  cfg then
      add $m_{stat}$ to cfg
// insert virtual methods in cfg
  for each e.$m_{vir}$() occurring in m do
    for each c in subtypes(type(e))
        and in clsSet(varset(m)) do
       $m_{def}$ := method (c, $m_{vir}$)
       if $m_{def}$ not in cfg then
          varSet(m)={ }
       if $m_{def}$ not in edges(clsSet(varset(m)) then
          if c in clsSet(v) then


            add $m_{def}$ to cfg with edges
// parameter/return type propagation
 for each v= a param
            add subtypes(clsSet(v))
               in  clsSet(m) to clsSet($m_{def}$)
            add subtypes(returntype($m_{def}$))
               in  clsSet(varset(m))
                 to clsSet(varset($m_{def}$))
            add c to clsSet(v)
// field type propagation
  for each read of f occurring in m do
     add clsSet(varSet(f)) to clsSet(varSet(m))
  for each write of f occurring in m do
    for each c in subtypes(type(f)) do
       if c not in clsSet(varSet(f))
         add c to clsSet(varSet(f))
// variable assignment type propagation
  for each read of v2 into v1 occurring in m do
     add clsSet(varSet(v2)) to clsSet(varSet(v1))
  for each write of v1 into v2 occurring in m do
    for each c in subtypes(type(v1)) do
       if c not in clsSet((varSet(v1))
         add c to clsSet((varSet(v1))
```

main	// the main method and representative
	// of other entry point methods
new x	// instantiation of an object of class x
marked(x)	// marked methods of class x
paramtypes(x)	// static types of the parameters of method x
returntype(x)	// static type of the return type of method x
clsSet(x)	// class set of x
x()	// call of static method x
type(x)	// declared type of the expression x
x.y()	//call of virtual method y in expresson x
subtypes(x)	// x and all classes which are a subtype of class x
subtypes({x})	// or all types in {x}
method(x,y)	// the method y which is defined for class x
mark(x,y)	// mark method x in class y
mark(x,y,z)	// mark method x in class y for method z
cfg	// call (flow) graph
varSet(v)	*// varset of v*
v	*// VTA variable – local, field, method, this, return*

Figure A.1: CHA, RTA, XTA, VTA algorithms as implemented

Appendix B

Detailed Statistics

B.1 Graph data

	Classes	**loaded**	**Reachable**	**methods**	**Missed**	**methods**	**Picked–up**	**methods**
	spec-		spec-		spec-		spec-	
	JVM98	In0	JVM98	In0	JVM98	In0	JVM98	In0
SA wIns	547	(252)	2682	(1000)	0	(0)	0	(0)
SA wIn0	547	(252)	2638	(1000)	5	(0)	0	(0)
SA rtIns	543	(255)	2575	(943)	46	(29)	0	(0)
SA noIns	543	(255)	2531	(943)	65	(29)	0	(0)
HPA wIns	547	(252)	2682	(1000)	0	(0)	0	(0)
HPA wIn0	547	(252)	2672	(1000)	0	(0)	0	(0)
HPA rtIns	543	(255)	2614	(961)	3	(0)	20	(13)
HPA noIns	543	(255)	2601	(961)	0	(0)	27	(13)
DA-HPA	531	(254)	2513	(961)	0	(0)	38	(19)
DA	293	(196)	0	(0)	0	(0)	811	(342)

Table B.1: By mode/inputs: classes / methods(reachable, missed, picked-up)

Test	Leaf	1 Def	1 Used	Poly	=	Virtual	Non-Virtual
SA wIns	707	330	2	105		1147	1529
SA wIn0	685	325	2	105		1119	1513
SA rtIns	690	303	2	97		1094	1475
SA noIns	668	297	2	97		1066	1460
HPA wIns	707	330	2	105		1147	1529
HPA wIn0	696	330	2	105		1135	1528
HPA rtIns	698	317	2	103		1122	1506
HPA noIns	687	317	2	103		1111	1505
DA-HPA	646	301	3	92		1044	1491
DA	175	68	6	9		260	536

Table B.2: By mode/inputs: mono (by test) vs. poly (fig.6.4)

Test	Leaf	1 Def	1 Used	Poly
CHA	1060	414	2	156
RTA	787	332	2	105
XTA	407	239	15	63
VTA	307	220	18	37
JIT	178	68	6	9

Table B.3: By algorithm: mono (by test) vs. poly (fig.6.11a)

	Non-virtual	**Virtual**	**Unknown**
CHA	3309 (95.2%)	162 (4.7%)	5 (0.1%)
RTA	2546 (91.7%)	116 (4.2%)	116 (4.2%)
XTA	1911 (95.5%)	64 (3.2%)	25 (1.3%)
VTA	1541 (88.5%)	4 (0.3%)	196 (11.2x%)
JIT	777 (96.6%)	9 (1.2%)	18 (2.3%)

Table B.4: By algorithm: Non-virtual vs. Virtual (fig.6.11b)

Total Time	In0	check	compress	db	jack	javac	jess	mpega	mtrt	geomean
SA wIns	0.19	0.48	9.36	21.13	8.74	13.49	8.78	10.29	8.34	7.35
SA wIn0	0.19	0.48	9.35	21.22	8.71	13.12	8.75	10.26	8.28	7.31
SA rtIn	0.18	0.47	9.35	21.04	8.65	13.2	8.77	10.3	8.15	7.27
SA noIns	0.18	0.47	9.43	21.21	8.65	13.2	8.67	10.26	8.41	7.31
HPA wIns	0.19	0.48	9.36	20.94	8.72	13.05	8.78	10.26	8.49	7.32
HPA wIn0	0.19	0.48	9.33	20.91	8.71	13.12	8.81	10.28	8.23	7.29
HPA rtIn	0.18	0.47	9.37	21.12	8.67	13.19	8.77	10.30	8.42	7.31
HPA noIns	0.18	0.47	9.42	21.05	8.67	13.19	8.67	10.28	8.12	7.27
DA-HPA	0.18	0.40	9.25	20.75	8.6	13.17	8.65	10.17	8.26	7.08
DA	0.16	0.27	9.16	20.60	8.45	13.84	8.57	10.51	7.67	7.62

Table B.5: By mode: Run times in seconds

classes	In0	check	compress	db	jack	javac	jess	mpega	mtrt	geomean
CHA	560	611	730	715	605	640	715	605	640	656
RTA	252	508	498	492	543	664	647	537	511	547
XTA	258	476	466	460	511	632	612	505	485	515
VTA	229	439	428	424	476	599	580	467	443	478
JIT	196	258	245	241	295	416	396	284	260	293

Table B.6: By classes by benchmark and algorithm (fig.6.8a)

methods	In0	check	compress	db	jack	javac	jess	mpega	mtrt	geomean
CHA	2979	3206	4132	3563	3162	3628	3563	3162	3628	3492
RTA	1000	2458	2385	2417	2641	3566	2996	2599	2524	2675
XTA	818	1784	1728	1768	1985	2919	2251	1917	1921	2006
VTA	635	1555	1498	1539	1756	2434	2101	1682	1650	1753
JIT	342	675	594	630	851	1412	1076	766	750	811

Table B.7: By methods by benchmark and algorithm (fig.6.8b)

B.2 Run-time comparison

Table B.8 shows the runtime comparisons of RTA, XTA, VTA and the JIT in SA-HPA mode with In0 inputs in seconds on 1.5GHz centrino with 496 MB RAM using *-time cacao* option. Bold means *cacao* was faster with the analysis, than without.

Run times were measured two ways (*cacao -time*, *perfex* time stamp counter (tsc) ([Cha01])) and gave similar results. However the runtime standard deviations, shown in parenthesis, were too high to use. Sometimes the geometric mean runtime with the analysis ran faster than without analysis. The geometric mean runtime of the algorithm with the least instructions was sometimes slower.

The benchmarks which ran sometimes faster without analysis than with it, may be due I/O patterns when loading classes in spurts or gradually [LAM07].

	RTA	**XTA**	**VTA**	JIT=no analysis	Min. Algo
In0	0.164(.003)	0.171 (.003)	0.159 (.002)	0.135(.002)	VTA
check	0.273(.003)	0.370 (.005)	0.346 (.004)	0.219(.003)	RTA
compress	**9.178** (.208)	**9.320** (.097)	**9.200** (.131)	9.476 (.257)	RTA
db	**20.645** (.241)	20.854 (.218)	**20.593**(.201)	20.763 (.276)	VTA
jack	8.408 (.008)	8.586 (.093)	**7.263** (.231)	8.230 (.063)	VTA
javac	13.344 (.097)	**12.621** (.133)	**12.611** (.041)	12.630 (.118)	VTA
jess	8.519 (.014)	8.659 (.094)	8.688 (.009)	8.474 (.092)	RTA
mpegaudio	10.404 (.139)	10.208 (.103)	10.397 (.379)	10.167 (.235)	XTA
mtrt	7.749 (.248)	7.995 (.301)	8.118 (.336)	7.658 (.182)	RTA
geomean (SPECjvm98)	6.68	6.96	6.77	6.44	RTA
num. wins 1st	4	1	4		tie
2nd	3	3	3		tie
3rd	2	5	2		XTA

Table B.8: By algorithm: Run time comparisons in seconds