Interprocedural Side-Effect Analysis and Optimisation in the Presence of Dynamic Class Loading

Phung Hua Nguyen **Jingling Xue**

Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales
NSW 2032, Sydney, Australia

Abstract

We introduce a new approach to computing interprocedural modification side effects for Java programs in the presence of dynamic class loading. When compile-time unknown classes can be loaded dynamically, the points-to and modification sets computed statically based on the analysed program, called internal world, may be incomplete. Thus, the modification side effects cannot be determined based on such sets alone. We present a technique, called internal analysis (IA), to classify the references, objects and call sites in the analysed program so that we can determine which points-to and modification sets are definitely complete and which may be not. By combining the points-to sets and this classification, we obtain a new IA-based interprocedural side-effect analysis for determining the modification side-effects of any statement on any variable with good accuracy. We have implemented our algorithms in Soot. This paper demonstrates two important applications of this work: partial redundancy elimination (PRE) and copy propagation for field accesses. We report significantly increased opportunities in performing these optimisations over benchmarks when compared with a recently proposed technique in these areas. Our techniques are simple since they are flow-insensitive and achieve these improvements at small analysis costs.

1 Introduction

Interprocedural side-effect analysis is an important technique because the information it provides can improve the precision of many analysis techniques such as dependence analysis and liveness analysis. In addition, many compiler optimisation techniques rely directly or indirectly on side-effect analysis when optimising a program. These techniques are PRE, copy/constant propagation, redundant load elimination, null pointer check elimination, array bound check elimination, instruction scheduling, to name just a few. Finally, side-effect analysis also plays an important role in software engineering. One recent example is its adaptation to support generation of summaries of environment behaviour (Tkachuk & Dwyer 2003).

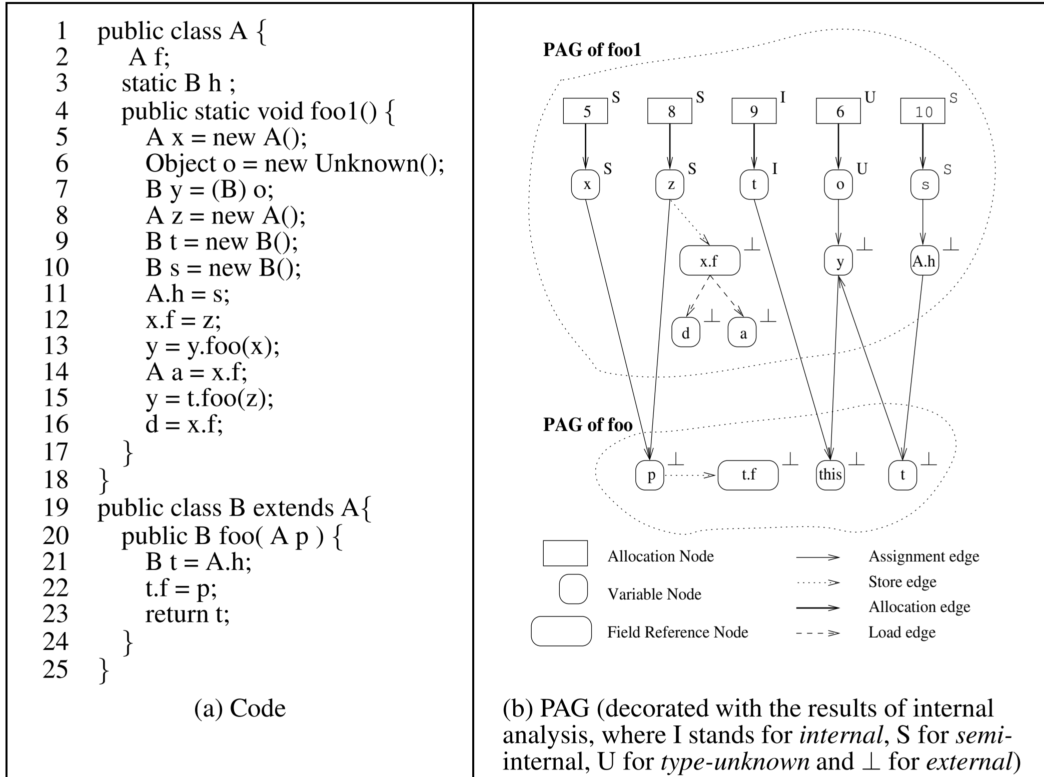
Dynamic class loading, which is an integral part of object-oriented languages such as Java and C#, presents challenges to side-effect analysis. The call graph of the program may be incomplete when some dynamically loaded classes cannot be determined statically. When computed based only on the compile-time known classes in the program, the points-to and modification sets may be incomplete (i.e., do not contain all possible objects found when more dynamically loaded classes become statically available). Thus, the modification side effects of a statement (assignment or call site) cannot be determined based on such sets alone. Consider a Java program given in Figure 1(a). In lines 6 and 7, an object of a compile-time unknown class is created and assigned to `y`. This class may be a subclass of class `B` containing an overriding method `foo` to replace `B`'s implementation. As a result, the invocation in line 13 with `y` as the receiver may call this unknown overriding method `foo`. By using the points-to sets computed based on `A` and `B` only, we cannot determine the modification set of this call site, i.e., whether the call site may modify the five objects created in lines 5 – 6 and 8 – 10.

Some approaches solve this problem by making the conservative assumption that a method invocation may modify any object (Hosking, Nystrom, Whitlock, Cutts & Diwan 2001, Vallée-Rai, Hendren, Sundaresan, Lam, Gagnon & Co 1999). The modification side effects of other kinds of statements are also approximated accordingly. As a result, the constant `z` in line 12 cannot be propagated across the call site in line 13, implying that the possibly redundant load `x.f` in line 14 cannot be eliminated. Other approaches expect the whole program to be available during the analysis (Razafimahefa 1999, Milanova, Rountev & Ryder 2002, Clausen 1997, Lhoták 2003). In order for the statement in line 6 to be analysed, the set of classes that may be dynamically loaded here must be explicitly specified (by the user). As a result, all points-to sets are complete and side-effect analysis can be carried out based on these sets in the normal manner. Suppose that the user indicates that the set just includes `B`. These approaches would conclude that `x.f` in line 12 is not modified by the call in line 13. Thus, the load in line 14 can be eliminated and replaced with the constant value `z` propagated from line 12. However, these optimisations may be incorrect if a new subclass of class `B` is loaded in line 6 at run time.

We introduce a new approach to computing interprocedural modification side effects for Java programs in the presence of dynamic class loading. The compile-time known classes in the analysed program form the so-called *internal world*. The points-to analysis is first applied to the internal world as if it were a whole-program. A new technique, called *internal analysis* (IA), is then applied to the internal world to classify all references, objects and call sites so that we can determine which points-to sets and modification sets are definitely complete and which may be not. By combining the points-to sets and this classification, we obtain a new IA-based interprocedural side-effect analysis for determining the modification side-effects of any statement on any variable with good accuracy.

Consider the program given in Figure 1(a). Figure 1(b) shows its pointer assignment graph (PAG) (Lhoták & Hendren 2003), where each node represents either a reference or a compile-time object identified by its line number. Let us assume that the internal-world consists of classes `A` and `B` only. Suppose we want to find out if the load `x.f` in line 14 can be replaced with the constant `z` propagated from line 12. The points-to sets for all references can be obtained by a visual inspection of the PAG. By performing the side-effect analysis based on these points-to sets alone, we would conclude that the call in line 13 does not mod-

Copyright ©2005, Australian Computer Society, Inc. This paper appeared at the 28th Australasian Computer Science Conference, The University of Newcastle, Australia. Conferences in Research and Practice in Information Technology, Vol. 38. V. Estivill-Castro, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.



(a) Code

(b) PAG (decorated with the results of internal analysis, where I stands for *internal*, S for *semi*-internal, U for *type-unknown* and ⊥ for *external*)

Figure 1: A running example program.

ify `x.f`. Thus, the load in line 14 can be eliminated by copy/constant propagation. To account for an unknown class that may be loaded in line 6, our internal analysis classifies all references and objects as shown in the PAG. This classification tells us that while the points-to set of `x` is definitely complete but the modification set of the call site in line 13 may be incomplete. In addition, the modification set may contain an object in the points-to set of `x`. Thus, the call site may modify `x.f`, implying that the load operation in line 14 cannot be safely optimised away. Similarly, the partially redundant `x.f` in line 16 with respect to `x.f` in line 14 cannot be removed safely.

We have implemented our algorithms in Soot, an optimising compiler for Java. We demonstrate two important applications of this work: PRE and copy propagation for field accesses. Both optimisations are guided by our IA-based interprocedural side-effect analysis. We present our experimental results over four benchmarks from SPECjvm98 and five other benchmarks. We compare our work against a recently proposed algorithm which combines PRE and type-based alias analysis (TBAA) (Hosking et al. 2001). In comparison with this algorithm, we eliminate between 0% to 108.0% redundant field accesses and between 0% to 52.9% redundant loads for copy propagation. Our analyses are flow-insensitive. These benefits are obtained at reasonably small analysis costs.

The rest of this paper is organised as follows. Section 2 introduces some background information. Section 3 discusses our internal analysis. Section 4 describes our IA-based side-effect analysis. Section 5 presents our experimental results over benchmarks. Section 6 reviews the related work. Section 7 concludes the paper.

2 Background

By *class* we mean a class or interface in Java. The term *virtual call* means either an `invokevirtual` or `invokeinterface`. The term *static call* means either an `invokespecial` or `invokestatic` call.

Our approach to conducting side-effect analysis and optimisation has four steps. First, the points-to sets are computed for the analysed program by using an Andersen-style points-to analysis for Java (Lhoták & Hendren 2003). Second, our internal analysis classifies all references, objects and call sites in the analysed program so that we can find out the references with definitely complete points-to sets and the call sites where all target methods can be definitely resolved when dynamic class loading is permitted. Third, the modification side effects of a statement are found by combining the results from the points-to analysis and internal analysis. Finally, some optimisations are performed on the analysed program.

Section 2.1 defines precisely the class of internal-world programs to which our approach is applicable. Section 2.2 describes the intermediate representation used for a program. Section 2.3 discusses our equivalent representation of a program using a pointer assignment graph (PAG). Section 2.4 makes precise the points-to analysis done on the PAG. Section 2.5 gives a classification of the references, objects and call sites in the PAG.

2.1 Internal World

It is probably meaningless trying to analyse all arbitrary, incomplete Java program. Let us define precisely the class of programs that can be handled by this work.

Definition 1 Let $\mathbb{C}$ be the set of classes, $\mathbb{M}$ the set of the methods in $\mathbb{C}$, $\mathbb{F}$ the set of fields declared in $\mathbb{C}$. We define the analysed program, i.e. the internal world denoted by IW, as a three-tuple:

$$IW = \langle \mathbb{C}, \mathbb{M}, \mathbb{F} \rangle$$

such that (1) all classes in $\mathbb{C}$ except the root class (i.e., `java.lang.Object`) have all their superclasses in $\mathbb{C}$, (2) there is not a reference appeared in $\mathbb{M}$ with a type not in $\mathbb{C}$, (3) there is not a read/write to a field not in $\mathbb{F}$, and finally, (4) there is not an access (i.e., a call) to a method not in $\mathbb{M}$.

During the execution of an analysed program, a class can be loaded dynamically by using a class loader or `Class.forName()`. A dynamically loaded class may

Stat.	Semantics	Syntax
S_1	object creation	$v = \texttt{new}\ cl()$
S_2	copy	$v = \ell$
S_3	static load	$v = cl.f$
S_4	static store	$cl.f = v$
S_5	instance load	$v = \ell.f$
S_6	instance store	$\ell.f = v$
S_7	virtual call site	$v = \ell_0.op(\ell_1, \ldots, \ell_k)$
S_8	static call site	$v = cl.op(\ell_1, \ldots, \ell_k)$

Figure 2: Statements (i.e., instructions) in intermediate representation.

be any class in $\mathbb{C}$ or a subclass of any class in $\mathbb{C}$. In addition, some methods in $\mathbb{M}$ can be `native` or requested explicitly by the user not to be involved in static analysis.

Definition 2 A class is *internal* if it is in $\mathbb{C}$ and *external* otherwise.

Definition 3 A method is *internal* if (1) it is a method in $\mathbb{M}$, (2) it is indicated (by the user) to participate in static analysis and (3) it is a Java method (i.e. its bytecode instructions are fully available). Otherwise, it is *external*.

2.2 Intermediate Representation

The three-address intermediate representation we use consists of the eight different kinds of statements (i.e., instructions) listed in Figure 2. For efficiency reasons, our internal analysis is flow-insensitive. Thus, the flow control statements in the program are irrelevant to our analysis and are ignored. Every Java program is transformed into such a representation. Every expression of the form `o.newInstance()` in a program is transformed into `new cl()` if `o.newInstance()` is detected statically to be an object of class in $\mathbb{C}$. Otherwise, `o.newInstance()` is replaced with `new` *Unknown*`()`, where *Unknown* denotes a class that is not statically known to be in $\mathbb{C}$.

Accesses to multi-dimensional arrays are represented in the standard manner by accesses to one-dimensional arrays (with the introduction of temporaries). In this work, a one-dimensional array is interpreted as a single, monolithic object. By introducing a special field, say, *sf*, all array accesses can be represented as instance field accesses. We do not distinguish accesses to different components of an array. For example, `a[i]` and `a[j]` are represented by only `a.`*sf*. Therefore, as far as our analysis is concerned, array accesses are expressed in the forms of S_5 and S_6 given in Figure 2.

By a *field access* we mean either a static field access or an instance field access. Let V be the set of local variables (including `this` and temporaries) and parameters. Every instance field can always be expressed in the form of $\ell.f$, where $\ell \in V$ is the *base* and f is an instance field variable. A static field is of the form $cl.f$, where the class name cl is the *base* and f is a static field variable in that class.

For simplicity, the fields in distinct classes are assumed to have distinct names. In addition, every method is assumed to return a value assigned to `v`. If its return type is `void`, `v` is simply considered as a pseudo variable. As a standard transformation, every method is transformed to possess a unique return statement of the form `return r`, where `r` $\in V$ is called a *return variable*. Such a statement does not appear in Figure 2. During the construction of the PAG for the program (see Section 2.3), the semantics of a return statement in the callee *op* is realised as an assignment from `r` to `v` in the standard manner. Finally, ℓ_0 in S_7 is called the *receiver expression* at that call site.

Other language features of Java such as exception, inner class and reflection are all deal with similarly.

Stat.	Syntax	Edge
S_1	$v = \texttt{new}\ cl()$	Allocation : $(v \leftarrow \texttt{new}\ cl())$
S_2,S_3,S_4	$v = \ell$	Assignment : $(v \leftarrow \ell)$
S_5	$v = \ell.f$	Load : $(v \leftarrow \ell.f)$
S_6	$\ell.f = v$	Store : $(\ell.f \leftarrow v)$

Figure 3: The four types of PAG edges

2.3 Pointer Assignment Graph

The *pointer assignment graph* (PAG) for a program, with its statements drawn from Figure 2, is a digraph which is a graph representation of the program. Nodes in the graph represents variables, parameters, field accesses and allocation sites. Edges in the graph represents object creation, assignment, load, store statements as well as parameter passing. Figure 3 presents the four types of edges in the graph. We refer to (Lhoták & Hendren 2003) for details. One example is given in Figure 1, where an allocation node represents a compile-time object identified by its line number.

The call graph of the internal world program is built in the usual manner except that all possible target methods contained in the internal classes must be included if they may ever be invoked at a call site. By Definition 1, these target methods can be found using, for example, the class hierarchy analysis (CHA) (Dean, Grove & Chamber 1995). The main reason for building the call graph this way is that all possible methods called at a call site must be examined in order to classify correctly all references and objects in the internal world.

The PAG for a program is constructed as follows:

1. The PAGs for all internal methods in the analysed program are built individually.

2. For every call site in the PAG of every internal method, we connect this PAG with the PAGs of the internal methods invokable at the call site as indicated in the call graph. The node x representing an argument at the call site is linked to the node y representing the corresponding parameter of a callee by an assignment edge $(y \leftarrow x)$. The node r representing the return variable of a callee is linked to the node v representing the LHS of the call statement at the call site by an assignment edge $(v \leftarrow r)$.

2.4 Points-to Analysis

Let O be the set of all compile-time objects created in the analysed program. The points-to set of a reference v found on the PAG of the program is:

$$PTS(v) \quad = \quad \{o \in O \mid v \text{ may point to } o\}$$

Definition 4 Let P be the PAG of a program. Let P' be the PAG of the same program when some external methods invoked at some call sites become internal, implying that P is a subgraph of P'. A points-to or modification set computed based on P is said to be *complete* if it remains unchanged when computed based on every possible P'. Otherwise, it is *incomplete*.

We classify the references, objects and call sites in the analysed program in order to determine which points-to sets and modification sets are definitely complete or not.

2.5 Classification of References, Objects and Call sites

Definition 5 An object may be *externally reachable* if the object may be pointed to by a reference in an external method or by a field of an externally reachable object.

Definition 6 An object is *external* if it is created in an external method. An object which is created in an internal method is

$$\begin{cases} \textit{type-unknown} & \textbf{if } \text{its runtime type is not statically determined,} \\ \textit{semi-internal} & \textbf{elif } \text{it is externally reachable,} \\ \textit{internal} & \textbf{otherwise}. \end{cases}$$

A type-unknown object is created by `java.lang.class.newInstance()` or `java.lang.reflect.Constructor.newInstance()`. Such an object may be externally reachable since its compile-time unknown constructor may make it so.

Therefore, semi-internal, type-unknown and external objects are externally reachable objects. But internal objects are reachable only within internal methods.

An externally reachable object can be manipulated by an external method as if the object were external. Therefore, we make a worst-case assumption that *a reference that may point to an external object may also point to any externally reachable object*.

Definition 7 A reference is

$$\begin{cases} \textit{external} & \textbf{if } \text{it may point to an external object} \\ \textit{type-unknown} & \textbf{elif } \text{it may point to a type-unknown object} \\ \textit{semi-internal} & \textbf{elif } \text{it may point to a semi-internal object} \\ \textit{internal} & \textbf{otherwise} \end{cases}$$

As an external object does not appear in the resolution of the points-to analysis in IW, an external reference may have an incomplete points-to set. However, a reference falling in the other three categories definitely has a complete points-to set. Following from the above assumption, we state that an external reference may be an alias of another external, type-unknown or semi-internal reference, i.e. they may point to the same object.

Definition 8 A call site is *internal* if all the target methods that may be invoked at that call site are internal methods and *external* otherwise.

As the code of an external method is unknown, we make another conservative assumption that *an external call site may modify any externally reachable object*.

The next section describes the analysis technique to classify references, objects and call sites.

3 Internal Analysis

The internal analysis is a data-flow problem for classifying all references and objects in the internal world. The semi-lattice used is $(L, \sqcap)$, where $L = \{\top, I, S, U, \bot\}$. The five lattice values, $\top, I, S, U, \bot$, represent the uninitialised, internal, semi-internal, type-unknown and external states, respectively. They are ordered as $\bot \sqsubset U \sqsubset S \sqsubset I \sqsubset \top$. The meet operator $\sqcap$ is defined as follows: $\top \sqcap e = e$, $\bot \sqcap e = \bot$, $U \sqcap S = U$, $U \sqcap I = U$ and $S \sqcap I = S$, where $e \in L$.

As discussed in Section 2.5, type-unknown objects are externally reachable. Following from the Java language specification, an object created in an internal method as an instance of an internal class becomes externally reachable only if it is

C1. pointed to by a non-private static reference field ,

C2. pointed to by a private static reference field in a class with at least one external method,

C3. returned by a non-private internal method,

C4. returned by a private internal method in a class with at least one external method,

C5. pointed to by an instance field of an externally reachable object,

C6. pointed to by an argument of an external call site.

Let us explain these rules. For Rules C1 and C2, an external method can make a reference point to the object via a static field. For Rules C3 and C4, an internal method may be invoked at a call site in an external method. Therefore, the object returned by the internal method may be pointed by the reference which receives the returned value at that call site. For Rule C6, the object may be pointed to by a parameter of an external method that may be invoked at the call site. As a result, an object that satisfies C1–C4 and C6 may be pointed to by a reference in an external method so it is externally reachable by Definition 5. Rule C5 is derived directly from Definition 5.

Each node in the PAG has a cell, called LAT, to hold its lattice value. The values in these cells may change as our analysis progresses. The analysis terminates once a maximal fixed-point has been reached. The output consists of the points-to sets of all references and an assignment of lattice values to all references, objects and call sites in the PAG.

3.1 Initialisation

All the nodes in the PAG start with $\top$. Every compile-time object o of the form `new cl()` created in an internal method is initialised as follows:
$\mathsf{LAT}(o) = \mathsf{LAT}(\texttt{new cl()}) = \begin{cases} U & \text{if cl} = \textit{Unknown} \\ I & \text{otherwise} \end{cases}$

Let v be a reference. We define:

$$\textit{UpdateP2S}(v) = \forall o \in \textit{PTS}(v) : \mathsf{LAT}(o) \sqcap = S$$

By Rules C1 and C2, *UpdateP2S*(v) is performed for every static field v satisfying the two rules. In addition, these fields are initialised to $\bot$ if they are not declared as **final**. This is because an external method can make v point to an external object.

By Rules C3 and C4, *UpdateP2S*(v) is performed for the return variable v of every internal method defined in the two rules. Furthermore, the node representing every parameter of every internal method defined in these two rules is initialised to $\bot$. This is because such a method may be called from an external method. As a result, the points-to sets of all its reference parameters should contain the objects passed by the external method.

Rule C5 is realised in the transfer functions for loads and stores given in Figure 4.

Finally, let us consider Rule C6. As a special case, a call site `o.newInstance()` in Java implicitly invokes an unknown constructor and is thus regarded as an external call site. By this rule, *UpdateP2S*(v) is performed for all the arguments of the call site. Rule C6 is also realised in the transfer functions for external call sites given in Figure 4.

3.2 Transfer Functions

Figure 4 gives the transfer functions for the four edges given in Figure 3. These transfer functions serve to propagate the lattice values across the edges representing the first six statements and parameter passing in the program. The transfer functions for allocation and assignment edges are self-explanatory. Let us examine the two transfer functions for a store edge $(\ell.f \leftarrow v)$. In (a), the lattice value from v is propagated to all $\ell'.f$, where ℓ' is an alias of ℓ, because ℓ and ℓ' may point to some common objects. In (b), $\mathsf{LAT}(\ell)$ is S, U or $\bot$ if and only if ℓ may point to at least one externally reachable or external object. Therefore, an external method may make $\ell.f$ point to an external object. So $\mathsf{LAT}(\ell.f)$ is set to $\bot$. Similarly, all objects in *PTS*(v) may become externally reachable (Rule C5). Hence, *UpdateP2S*(v) is performed. Note that (b) is applied to ℓ but not to its aliases for two reasons. First, if $\mathsf{LAT}(\ell')$ is S, U or $\bot$, then $\mathsf{LAT}(\ell'.f)$ will be set to $\bot$ when a load from or a store into $\ell'.f$ is evaluated. Second, it is redundant to perform *UpdateP2S*(v) for each of its

Edges			Transfer Functions
Allocation	:	$(v \leftarrow \texttt{new}\ cl())$	$\mathsf{LAT}(v) \sqcap = \mathsf{LAT}(\texttt{new}\ cl())$
Assignment	:	$(v \leftarrow \ell)$	$\mathsf{LAT}(v) \sqcap = \mathsf{LAT}(\ell)$
Load	:	$(v \leftarrow \ell.f)$	(a) $\mathsf{LAT}(v) \sqcap = \mathsf{LAT}(\ell.f)$ (b) if $\mathsf{LAT}(\ell) = \{S, U, \bot\}$ $\mathsf{LAT}(\ell.f) = \bot$
Store	:	$(\ell.f \leftarrow v)$	(a) $\mathsf{LAT}(\ell'.f) \sqcap = \mathsf{LAT}(v)$, for all aliases ℓ' of ℓ, i.e., all ℓ' such that *PTS*$(\ell') \cap$ *PTS*$(\ell) \neq \emptyset$ (b) if $\mathsf{LAT}(\ell) = \{S, U, \bot\}$ $\mathsf{LAT}(\ell.f) = \bot$ *UpdateP2S*(v)

Figure 4: Transfer functions for edges.

Call	Transfer Functions
S_7	Let there be n target methods in IW at this call site found using CHA (a) $\mathsf{LAT}(S_7) = \mathsf{LAT}(\ell_0.op(\ell_1, \ldots, \ell_k)) = \begin{cases} \bot & \text{if } n = 0 \text{ or one of the } n \text{ targets is external} \\ I & \text{elif } op \text{ or the declared class of } \ell_0 \text{ is } \textbf{final} \\ \bot & \text{elif } \mathsf{LAT}(\ell_0) \in \{U, \bot\} \\ I & \text{otherwise} \end{cases}$ (b) $\left\{ \begin{array}{l} \mathsf{LAT}(v) = \bot \\ \textit{UpdateP2S}(\ell_i)\ \forall i = 0, \ldots, k \end{array} \right\}$ if $\mathsf{LAT}(S_7) = \bot$
S_8	(a) $\mathsf{LAT}(S_8) = \mathsf{LAT}(cl.op(\ell_1, \ldots, \ell_k)) = \begin{cases} \bot & \text{if the target method } cl.op \text{ is external} \\ I & \text{otherwise} \end{cases}$ (b) $\left\{ \begin{array}{l} \mathsf{LAT}(v) = \bot \\ \textit{UpdateP2S}(\ell_i)\ \forall i = 1, \ldots, k \end{array} \right\}$ if $\mathsf{LAT}(S_8) = \bot$

Figure 5: Transfer functions for call sites.

aliases. Finally, let us consider the two transfer functions for a load edge $(v \leftarrow \ell.f)$. In (a), the lattice value from $\ell.f$ is propagated to v. In (b), if $\mathsf{LAT}(\ell)$ is S, U or $\bot$, then $\mathsf{LAT}(\ell.f) = \bot$ as explained in (b) of the transfer function for store edge. This part is repeated for both loads and stores because a load from $\ell.f$ or a store into $\ell.f$ may not be present in the program.

Figure 5 gives the transfer functions for the two kinds of call sites given in Figure 2. These functions are necessary since a compile-time unknown method invoked at an external call site behaves like a blackbox. In the PAG of the program, there are no assignment edges connecting the arguments to the corresponding parameters of the unknown method and no assignment edge connecting the return variable of the unknown method to caller's node v representing the return value. Let us examine the transfer functions for a virtual call site, $S7$, given in Figure 2. In (a), there are four cases in determining the new state of a virtual call site. The first case is obvious. The second case is justified because a final method or a method declared in a final class cannot be overridden. The method is the only target of the call site regardless of the runtime type of the receiver. In the third case, the call site is external if its receiver is type-unknown or external. Then an external method may be invoked at the call site. If the last case is reached, the call site is internal. The transfer function (b) comes into play if the call site is external. In this case, v becomes external since it may point to an external object which may be returned from an external method invoked at the call site. In addition, every object pointed to by an argument at the call site becomes external reachable (C6).

The transfer function for a static call site, $S8$, are similar but simpler.

Our analysis problem is solved iteratively until a maximal fixed point is found.

Theorem 1 *Our internal analysis provides a under-approximation of the internal states of references, objects and call sites in the analysed program.*

Proof Follows directly from Rules C1 – C6, the initialisations stated in Section 3.1 and the transfer functions given in Figures 4 and 5. □

3.3 Example

Consider the example IW given in Figure 1, where all compile-time objects are identified by their line numbers. All nodes in the PAG start with $\top$. Then the four allocation nodes, 5, 8, 9, and 10 are initialised to I while 6 to U. The program has one static field `A.h`, where *PTS*(A.h) $= \{10\}$. Thus, the node `A.h` is set to $\bot$ and the state of the node 10 is applied $\sqcap$ with S. The nodes `this` and `p`, which represent the two parameters of the public method `foo`, are set to $\bot$. The variable `t`, where *PTS*(t) $= \{10\}$, is returned by the method `foo`. Therefore, the node 10 is applied $\sqcap$ with S again.

Let us examine the internal analysis on the example briefly. When the statements in lines 6 – 7 are processed, the node `y` becomes U. The node `y` represents the receiver at the call site `y = y.foo(x)` in line 13, where *PTS*(x) $= \{5\}$. Based on the transfer functions for a virtual call site, the state of the node y drops to $\bot$ while the state of the node 5 drops to S. The node `x` becomes S eventually due to the statement in line 5. When the store in line 12 is processed, we apply $\sqcap$ with $\bot$ to the state of the node `x.f` and $\sqcap$ with S to the state of the node in *PTS*(z) $= \{8\}$. The state of the node `x.f` changes to $\bot$ and the state of the node 8 drops to S. Similarly, when the node `t` in `foo` becomes $\bot$ and the store in line 22 is processed, we apply $\sqcap$ with $\bot$ to the state of `t.f` and $\sqcap$ with S to the states of the nodes in *PTS*(p) $= \{5, 8\}$.

The maximal fixed point found eventually is depicted in Figure 1.

4 IA-based Side-Effect Analysis

In Java, a non-field variable can be modified only by an assignment statement where the variable itself is the left-

hand side (LHS). In contrast, a field variable $v.f$ may be modified by an assignment to $u.f$ if u and v may point to the same object. In the presence of dynamic class loading, the points-to sets may be incomplete. Our internal analysis allows us to determine which points-to sets are definitely complete and which may be incomplete. Based on the information, we compute a modification set for each statement and then the side-effects on any given variable.

4.1 Computing Modification Sets

The side-effect analysis computes for each statement, a modification set consisting of all objects that may be modified by that statement. The modification set of a statement includes the objects modified by the statement itself and those modified by any methods that may be invoked directly or indirectly by that statement.

Let *MOD* be a mapping from $((M \cup S) \times (\mathbb{F} \cup \{sf\}))$ to $(P \times L \times E)$, where M is the set of internal methods in IW, S is the set of statements in these methods, $\mathbb{F}$ is the set of static or instance fields in IW, sf is the special field representing an array access, $P = 2^{O \cup \mathbb{C}}$ is the power set of the set that contains all compile-time objects and classes, $L = \{\top, I, S, U, \bot\}$ is our lattice set, and finally, $E = \{true, false\}$. (Note that $\mathbb{C}$ is defined in Definition 1 and O in Section 2.4.)

The *modification set* *MOD*(x, f) specifies that $x \in M \cup S$ may modify the field $f \in \mathbb{F} \cup \{sf\}$ of all the objects in P. The component L represents the smallest of the lattice values of all references r in the analysed program such that r.f may be modified by x. The E component indicates if s contains an external call site or not. We use *MOD*$(x, f).P$, *MOD*$(x, f).L$ and *MOD*$(x, f).E$ to denote the corresponding components of the modification set.

The *modification set* of an internal method $m \in M$ is simply taken as the union of the modification sets of all statements contained in the method:

$$MOD\,(m, f) = (\emptyset, \top, false) \cup \bigcup_{\substack{s \text{ is a} \\ \text{statement} \\ \text{in } m}} MOD\,(s, f)$$

where the existence of $(\emptyset, \top, false)$ ensures that *MOD*(s, f) is well-defined (in the case when the method is empty). The union of two modification sets is defined as: $(P_1, L_1, E_1) \cup (P_2, L_2, E_2) = (P_1 \cup P_2, L_1 \sqcap L_2, E_1 \vee E_2)$.

Figure 6 gives the rules to compute *MOD* for the 8 basic statements listed in Figure 2. The first two rules are for statements S_4 and S_6 that write static and instance fields, respectively. In the case of S_7, a virtual call site, the modification set includes the union of the modification sets of all the possible internal methods that may be invoked at that call site. To complete the specification of the modification set, there are two cases. If the call site is external, we add $(\emptyset, \top, true)$ to the set since an external method may be invoked at the call site. If the call site is internal, there is nothing to add. The rule for S_8, a static call site, is similar but simpler since whether its unique target method is internal or not can be determined without any flow analysis. Finally, the four remaining statements, S_1, S_2, S_3 and S_5 listed in Figure 2 do not modify any given field g.

Our internal analysis also allows us to determine which modification sets are definitely complete and which may be incomplete. The following results follow directly from Theorem 1 and the rules given in Figure 6. Their proofs are thus omitted.

Theorem 2 *If MOD$(s, f).L = \bot$, then MOD$(s, f).P$ may be incomplete. What are missing in MOD$(s, f).P$ may be any semi-internal, type-unknown or external (but not internal) object. If MOD$(s, f).E$ = true, then MOD$(s, f).P$ may be incomplete. What are missing in MOD$(s, f).P$ may be any internal class $cl \in IW$ in which f is a field variable and any semi-internal, type-unknown or external (but not internal) object. Otherwise, MOD$(s, f).P$ is definitely complete.*

```
Boolean SideEffectChecker(Statement s, Variable x)
 (1)  if x is a non-field variable of the form v
 (2)     if LHS of s is x
 (3)        return true
 (4)  elseif x is an instance field of the form v.f
 (5)     if LHS of s is v
 (6)        return true
 (7)     elseif LAT(v) = ⊥ and MOD(s, f).L ⊑ S
 (8)        return true
 (9)     elseif LAT(v) ⊑ S and MOD(s, f).L = ⊥
(10)        return true
(11)     elseif LAT(v) ⊑ S and MOD(s, f).E
(12)        return true
(13)     elseif PTS(v) ∩ MOD(s, f).P ≠ ∅
(14)        return true
(15)  elseif x is a static field of the form cl.f
(16)     if MOD(s, f).E
(17)        return true
(18)     elseif MOD(s, f).P = {cl}
(19)        return true
(20)  return false
```

Figure 7: IA-based side-effect checker for programs.

4.2 Checking Side Effects

Our algorithm, CHECKER, determines the side-effects of a statement s on any given variable x, where s and x are in internal methods in IW. There are three cases depending on whether x is a non-field variable v, an instance field $v.f$ or a static field $cl.f$. CHECKER returns true in the first case if s modifies v, which is trivial in Java, true in the second case if s may modify either v or $v.f$, true in the third case if s may modify $cl.f$ and false otherwise.

The correctness of our algorithm follows from Theorems 1 and 2. To understand its last two cases, let us see how they can be simplified based on these two theorems. But such a method is inefficient and should not be used. Let $\mathcal{E}$ be a special external object created in an external method. Let $\mathcal{A}$ be the set consisting of all semi-internal, all type-unknown objects and $\mathcal{E}$. If $\mathsf{LAT}(v) = \bot$, then add all elements in $\mathcal{A}$ to *PTS*(v). If *MOD*$(s, f).L = \bot$, then add all elements in $\mathcal{A}$ to *MOD*$(s, f).P$. If *MOD*$(s, f).E = \bot$, then add all elements in $\mathcal{A}$ and all internal classes in IW in which f is a field variable to *MOD*$(s, f).P$. Then lines 7 – 12 in CHECKER are redundant and can be removed and lines 16 – 19 can be replaced by:

if $cl \in$ *MOD*$(s.f).P$
return *true*

4.3 Example

As before we identify each compile-time object by the number of the line where it is created. We also identify a statement by its line number. Below we list the modification sets of all statements in our running example that are not empty, i.e., $(\emptyset, \top, false)$.

$$\begin{aligned} MOD(22, \texttt{f}) &= (\{10\}, \bot, false) \\ MOD(\texttt{foo}, \texttt{f}) &= \bigcup_{s \texttt{ in foo}} MOD(s, \texttt{f}) \\ &= MOD(22, \texttt{f}) = (\{10\}, \bot, false) \\ MOD(11, \texttt{A.h}) &= (\{\texttt{A}\}, \top, false) \\ MOD(12, \texttt{f}) &= (\{5\}, S, false) \end{aligned}$$

Since $\mathsf{LAT}(y) = \bot$ and `foo` is not **final**, we have $\mathsf{LAT}(\texttt{y.foo(x)}) = \bot$. Hence,

Statement	*MOD*
$S_4 : cl.f = v$	$MOD(S_4, g) = \begin{cases} (\{cl\}, \top, \mathit{false}) & \text{if } g = cl.f \\ (\emptyset, \top, \mathit{false}) & \text{otherwise} \end{cases}$
$S_6 : \ell.f = v$	$MOD(S_6, g) = \begin{cases} (PTS(\ell), \mathsf{LAT}(\ell), \mathit{false}) & \text{if } g = f \\ (\emptyset, \top, \mathit{false}) & \text{otherwise} \end{cases}$
$S_7 : v = \ell_0.op(\ell_1, \ldots, \ell_k)$	$MOD(S_7, g) = \bigcup_{m \in M \text{ may be invoked from } S_7} MOD(m, g) \cup \begin{cases} (\emptyset, \top, \mathit{true}) & \text{if } \mathsf{LAT}(\ell_0.op(\ell_1, \ldots, \ell_k)) = \bot \\ (\emptyset, \top, \mathit{false}) & \text{otherwise} \end{cases}$
$S_8 : v = cl.op(\ell_1, \ldots, \ell_k)$	$MOD(S_8, g) = \begin{cases} (\emptyset, \top, \mathit{true}) & \text{if } \mathsf{LAT}(cl.op(\ell_1, \ldots, \ell_k)) = \bot \\ MOD(op, g) & \text{otherwise} \end{cases}$
S_1, S_2, S_3, S_5	$MOD(S_1, g) = \ldots = MOD(S_5, g) = (\emptyset, \top, \mathit{false})$

Figure 6: Rules for computing *MOD* for all the statements in Figure 2.

$$\begin{aligned} MOD(13, \texttt{f}) &= MOD(\texttt{foo}, \texttt{f}) \cup (\emptyset, \top, \mathit{true}) \\ &= (\{10\}, \bot, \mathit{true}) \\ MOD(13, \texttt{g}) &= (\emptyset, \top, \mathit{true}), \text{ where } \texttt{g} \neq \texttt{f} \end{aligned}$$

Since $\mathsf{LAT}(\texttt{t.foo(z)}) = I$, the modification set for the statement in line 15 is:

$$\begin{aligned} MOD(15, \texttt{f}) &= MOD(\texttt{foo}, \texttt{f}) \cup (\emptyset, \top, \mathit{false}) \\ &= (\{10\}, \bot, \mathit{false}) \\ MOD(\texttt{foo1}, \texttt{f}) &= \bigcup_{s \text{ in } \texttt{foo1}} MOD(s, \texttt{f}) \\ &= (\{5, 10\}, \bot, \mathit{true}) \\ MOD(\texttt{foo1}, \texttt{A.h}) &= \bigcup_{s \text{ in foo1}} MOD(s, \texttt{A.h}) \\ &= (\{\texttt{A}\}, \top, \mathit{true}) \end{aligned}$$

Let us apply CHECKER to conduct some side-effect analysis. The call in line 13 may modify `x.f`, i.e., $\text{CHECKER}(13, \texttt{x.f}) = \mathit{true}$ because $\mathsf{LAT}(\texttt{x}) = S$ and $MOD(13, \texttt{f}).E = \mathit{true}$. Since `y` may point to an object whose runtime type is a subclass of `B`, this call site may invoke a compile-time unknown overriding method of `foo` in the subclass. When `x` is passed as an argument to the method, `x.f` may be modified inside.

The call in line 15 may modify `x.f`, i.e., $\text{CHECKER}(15, \texttt{x.f}) = \mathit{true}$ because $\mathsf{LAT}(\texttt{x}) = S$ and $MOD(15, \texttt{f}).L = \bot$. Let us give a scenario in which such a modification takes place. The target of the call in this statement is certainly the method `foo` contained in `B` because the declared type of `t` is `B` and the internal state of `t` is I. The external call made in line 13 can make `A.h` point to the same object that `x` points to, i.e. the object 5. When `t` is assigned in line 21, `t` will point to the object that `x` points to. So the statement in line 22 that modifies `t.f` also modifies `x.f`.

5 Experiments

We have implemented our internal analysis and IA-based side-effect analysis algorithms in Soot (Vallée-Rai et al. 1999), a bytecode to bytecode optimiser. In Soot, only whole-program analyses and optimisations are supported. There is a preprocessing translator that converts Java bytecode into a three-address representation called *Jimple*. All the statements in Jimple relevant to our analysis are listed in Figure 2. Recall from Section 2.2 that array accesses are interpreted as instance field accesses during the analysis.

A program being analysed is represented by its PAG. The points-to sets for the analysed program are computed using a points-to analysis pass in Soot (Lhoták & Hendren 2003). We have implemented our internal analysis and side-effect analysis algorithm as separate passes.

We present our experimental results using nine benchmarks. The first four are from SPECjvm98, `jasmin` (version sable-1.1) is a Java assembler interface , `jlex` (version 1.2.6) and `javacup` (version 0.10k) are Java scanner and parser generators from Princeton University, `jb` (version 7.0) is a parser and lexer generator for Java from Colorado University and `jtar` (version 1.21) is GNU's `tar` ported to Java. In our experiments, the internal-world for a benchmark consists of classes and their methods in both the application and the library that can be reached statically from any non-private method of the application.

We measure the precision of an side-effect analysis with the average number of field accesses modified by a statement (assignment or call site) in a program. We compare our analysis technique with the type-based alias analysis (TBAA) (Diwan, McKinley & Moss 1998), which is recently used in (Hosking et al. 2001) for optimising Java programs. Table 1 shows that our analysis yields more precise results across all the benchmarks used. We have considered only the field accesses and the statements appearing in the internal methods from the application part of a benchmark. Our analysis provides a more precise estimate about the average number of field accesses that may be modified per statement than the TBAA approach. The array accesses, which are discussed in Section 2.2, are included in the statistics for instance field accesses.

We also demonstrate two important applications of IA-based side-effect analysis in compiler optimisations: partial redundancy elimination (PRE) and copy propagation for eliminating redundant field accesses, i.e., loads. PRE is an important optimisation that subsumes loop-invariant code motion and common subexpression elimination (Morel & Renvoise 1979). Existing PRE algorithms (Horspool & Ho 1997, Knoop, Rüthing & Steffen 1994, Cai & Xue 2003) deal with arithmetic expressions involving only local variables (e.g., $a+b$). Recently, Hosking *et al* (Hosking et al. 2001) combine PRE with TBAA to deal with field accesses. In comparison with this earlier approach (Section 6), our IA-based side-effect analysis allows a more effective PRE to be performed for field accesses. In our second application, we show further that our analysis is also more effective in performing copy propagation for field accesses. In our experiments, both optimisations are applied only to the internal methods in the application part of a benchmark.

To the best of our knowledge, this paper is the first demonstration of these optimisations in Java programs by considering the interprocedural modification side effects in the presence of dynamic class loading.

Table 2 compares the two PRE algorithms and the two copy propagation algorithms in terms of the number of redundant loads eliminated. We see convincingly that our IA-based side-effect analysis has successfully enabled sig-

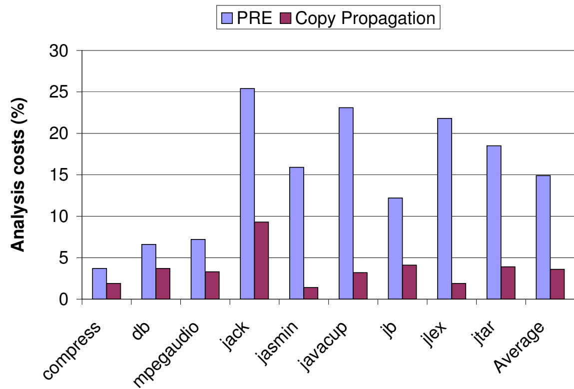


Figure 8: Analysis costs of PRE and copy propagation using IA-based approach over the TBAA-based approach.

nificantly more redundancies to be eliminated. In PRE, the improvements over the TBAA-based algorithm range from 0% to 108.0%. In copy propagation, our algorithm eliminates between 0% and 52.9% more redundancies.

Table 3 shows that our internal analysis is extremely efficient. By computing the lattice values of all references and objects, we are able to determine which points-to and modification sets are definitely complete or not, providing sufficient information to compute the modification side effects. In comparison with the compile time spent by the points-to analysis pass, the average overhead of our internal analysis pass for all the benchmarks is only 2.6%.

Finally, Figure 8 presents the analysis costs of our IA-based approach relative to the TBAA-based approach in performing both PRE and copy propagation. The costs for PRE increase from 3.7% to 25.4% with an average of 14.9% over the TBAA-based approach. The costs for copy propagation increase from 1.9% to 9.3% with an average of 3.6%. PRE is more expensive among the two optimisations since several data-flow passes are required in order to find the optimal placements for inserted temporaries.

6 Related Work

We review the existing work related to internal analysis and side-effect analysis below.

6.1 Internal analysis

Points-to analysis (Rountev & Ryder 2000, Harrold & Rothermel 1996) for incomplete programs on imperative languages has been studied for a long time. Unlike these approaches, our approach takes into account some new features of object-oriented languages such as polymorphism, dynamic binding and dynamic loading.

Chatterjee et al. (Chatterjee & G.Ryder 2001) present a points-to analysis for library modules to find def-use relations. The analysis evaluates a parameterised points-to solution for each method and propagates conservative assumptions about the clients of the library in top-down manner. The limitation of the approach is that it does not examine the affects of statically unknown codes such as native codes or dynamically loaded codes.

Extant analysis (Sreedhar, Burke & Choi 2000) is designed for the purposes of specialising Java programs in the presence of dynamic class loading. The technique partitions references into two categories: *unconditionally extant references* when they just point to objects whose runtime types are in the closed world and *conditionally extant references* otherwise. The internal analysis can obtain the information about the completeness of a points-to set by which we apply to side-effect analysis.

Immutability analysis (Porat, Biberstein, Koved & Mendelson 2000) is a technique for detecting mutability of fields and classes in a Java program. Field analysis (Ghemawat, Randall & Scales 2000) exploits the declared access restrictions placed on fields in order to determine useful properties of these fields, such as `exact_type`, `nonnull`, `may_leak` and `only_init`.

Escape analysis (Blanchet 1998, Blanchet 1999, Choi, Gupta, Serrano, Sreedhar & Midkiff 1999, Whaley & Rinard 1999, Vivien & Rinard 2001) detects the objects that never escape out of a method or thread. An object escapes a method M if its lifetime may exceed the lifetime of M. An object that does not escape a method can be possibly allocated on the method's stack frame. If an object does not escape a thread, no other threads can access the object. The synchronisation operations associated with the object can be eliminated. In general, our internal analysis can be regarded as a kind of escape analysis for detecting objects escaping out of the analysed program. The major differences are that (1) the access properties of fields and methods are taken account, (2) the completeness of points-to sets is determined, and (3) the completeness of virtual call sites is evaluated in our technique.

Some dynamic points-to analysis techniques for Java (Pechtchanski & Sarkar 2001, Hirzel, Diwan & Hind 2004) restrict themselves only to loaded classes during program execution. The analysis and optimisation techniques that make use of the points-to information may require runtime invalidation and recompilation mechanisms, which can hurt performance. In addition, side-effect analysis can not be easily done when some points-to sets are incomplete. The proposed technique in this paper allows side-effect analysis and other analysis and optimisation techniques to be applied without runtime invalidation and recompilation.

6.2 Side-Effect Analysis

Side-effect analysis for imperative languages has been studied for a long time. Banning (Banning 1979) is one of the earliest providing a systematic treatment of this problem. His approach works on imperative languages without pointers and where aliasing occurs only through parameter passing. Cooper and Kennedy (Cooper & Kennedy 1988) improve Banning's work by dividing the side-effect problem into two subproblems: the side effect analysis for formal parameters and the side-effect analysis for global variables. Ryder et. al. (Ryder, Landi, Stocks, Zhang & Altucher 2001) present an interprocedural side-effect analysis algorithm with pointer aliasing. Like these existing approaches, our approach is flow-insensitive. However, our approach works for object-oriented languages that support dynamic class loading.

Some interprocedural side-effect analysis techniques have been proposed for Java programs (Clausen 1997, Razafimahefa 1999, Lhoták 2003, Rountev 2004). However, they expect the *whole program* to be present during the analysis. Sălcianu and Rinard (Sălcianu & Rinard 2004) present a static analysis for identifying pure methods. Their technique is based on a specialised pointer/escape analysis from (Whaley & Rinard 1999). Our approach evaluates side-effect based on any flow- and context-insensitive points-to analysis. This permits the use of a variety of existing points-to analysis (Lhoták 2003, Berndl, Lhoták, Qian, Hendren & Umanee 2003).Rountev and Ryder (Rountev & Ryder 2001) analyse separately client modules and library modules implemented in C. They construct summary information conservatively about the library modules and then use the information when they analyse the client modules.

There are some intraprocedural side-effect analysis techniques that 'support' dynamic class loading (Hosking et al. 2001, Vallée-Rai et al. 1999). However, they simply make the worst-case assumption that a call may modify all objects. A simple side-effect analysis, called a naive side-effect analysis, is implemented in (Vallée-Rai et al. 1999), that assumes that `a` and `b` may point to the same object

Benchmark	(Instance Field Accesses)/Statement		(Static Field Accesses)/Statement	
	TBAA-based	IA-based	TBAA-based	IA-based
compress	80.4	59.2	8.2	5.3
db	59.6	53.8	10.9	9.8
mpegaudio	542.8	392.0	7.6	5.0
jack	414.3	395.3	16.1	15.4
jasmin	552.7	455.5	9.0	7.0
javacup	505.8	423.7	56.9	47.5
jb	134.0	110.1	34.4	28.0
jlex	553.4	461.8	1.8	1.5
jtar	386.0	297.5	30.6	23.8

Table 1: A comparison of the precisions of TBAA-based and IA-based side-effect analysis techniques.

Benchmark	Partial Redundancy Elimination (PRE)			Copy Propagation		
	TBAA-based	IA-based (%)		TBAA-based	IA-based (%)	
compress	25	52	(108.0)	13	15	(15.4)
db	10	10	(0.0)	0	0	(n/a)
mpegaudio	421	493	(17.1)	122	143	(17.2)
jack	115	178	(54.8)	216	227	(5.1)
jasmin	71	93	(31.0)	13	13	(0.0)
javacup	39	78	(100.0)	15	20	(33.3)
jb	19	29	(52.6)	17	26	(52.9)
jlex	476	570	(19.7)	25	25	(0.0)
jtar	58	98	(69.0)	28	37	(32.1)

Table 2: A comparison of PRE and copy propagation algorithms w.r.t, redundant loads eliminated.

for any two field accesses `a.f` and `b.f`. Hosking *et al* (Hosking et al. 2001) perform PRE for field accesses by using the type-based alias analysis (TBAA) (Diwan et al. 1998) to provide a more accurate side-effect analysis. Let the declared types of v and w be T_1 and T_2, respectively. Then v and w may be aliases iff T_1 and T_2 are identical or one is a subclass of the other. Therefore, $v.f$ and $w.g$ may be aliases iff $f = g$ and v and w are aliases. Our experimental results over benchmark programs show that our IA-based side-effect analysis provides a more accurate information than the approach.

7 Conclusion

In this paper, we solve an important problem for efficient execution of Java programs. We describe a framework for interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. When classes that are dynamically loaded are unknown at compile time, the points-to and modification sets may be incomplete. We present an internal analysis technique for classifying references in the internal-world program into internal, semi-internal, type-unknown and external references. By combining points-to analysis and internal analysis, we present an algorithm for computing interprocedural modification side effects when dynamic class loading is permitted. We have evaluated the precision of our side-effect analysis and demonstrated its effectiveness in two important applications — PRE and copy propagation for field accesses. Our experimental results using benchmarks show significant benefits of our analysis techniques in compiler optimisations at reasonably small analysis costs.

References

Banning, J. P. (1979), An effecient way to find the side-effects of procedure calls and the aliases of variables, *in* '6th Annual ACM Symposium on Principles of Programming Languages', San Antonio, Texas, pp. 29–41.

Berndl, M., Lhoták, O., Qian, F., Hendren, L. & Umanee, N. (2003), Points-to analysis using BDDs, *in* 'ACM SIGPLAN '03 Conference on Programming Language Design and Implementation', pp. 103–114.

Blanchet, B. (1998), Escape analysis: Correctness proof, implementation and experimental results, *in* '25th Annual ACM Symposium on Principles of Programming Languages', pp. 25–37.

Blanchet, B. (1999), Escape analysis for object-oriented languages: application to Java, *in* '14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications', pp. 20–34.

Cai, Q. & Xue, J. (2003), Optimal and efficient speculation-based partial redundancy elimination, *in* '1st IEEE/ACM International Symposium on Code Generation and Optimization', pp. 91–102.

Chatterjee, R. & G.Ryder, B. (2001), Data-flow-based testing of object-oriented libraries, Technical Report DCS-TR-433, Rutgers University.

Choi, J.-D., Gupta, M., Serrano, M. J., Sreedhar, V. C. & Midkiff, S. P. (1999), Escape analysis for Java, *in* '14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications', pp. 1–19.

Clausen, L. R. (1997), 'A Java bytecode optimizer using side-effect analysis', *Concurrency: Practice and Experience* **9**(11), 1031–1045.

Cooper, K. D. & Kennedy, K. (1988), Interprocedural side-effect analysis in linear time, *in* 'ACM SIGPLAN '88 Conference on Programming Language Design and Implementation', Atlanta, Georgia, pp. 57–66.

Dean, J., Grove, D. & Chamber, C. (1995), Optimization of object-oriented programs using static class hierarchy analysis, *in* '5th European Conference on

Benchmark	Points-to Analysis (secs)	Points-to Analysis + IA (secs)	Increase (%)
compress	152.6	157.8	3.4
db	151.9	157.1	3.4
mpegaudio	166.0	171.2	3.1
jack	162.0	167.3	3.3
jasmin	49.2	50.0	1.6
javacup	45.5	46.3	1.8
jb	38.0	38.7	1.8
jlex	40.7	41.4	1.7
jtar	156.8	162.4	3.6

Table 3: Analysis costs of internal analysis (IA).

Object-Oriented Programming', Vol. 952, Springer, pp. 77–101.

Diwan, A., McKinley, K. S. & Moss, J. E. B. (1998), Type-based alias analysis, *in* 'ACM SIGPLAN '98 Conference on Programming Language Design and Implementation', pp. 106–117.

Ghemawat, S., Randall, K. H. & Scales, D. J. (2000), Field analysis: Getting useful and low-cost interprocedural information, *in* 'ACM SIGPLAN '00 Conference on Programming Language Design and Implementation', pp. 334–344.

Harrold, M. J. & Rothermel, G. (1996), 'Separate computation of alias information for reuse', *IEEE Transactions on Software Engineering* **22**(7), 442–460.

Hirzel, M., Diwan, A. & Hind, M. (2004), Pointer analysis in the presence of dynamic class loading, *in* '18th European Conference on Object-Oriented Programming'.

Horspool, R. & Ho, H. (1997), Partial redundancy elimination driven by a cost-benefit analysis, *in* '8th Israeli Conference on Computer System and Software Engineering', pp. 111–118.

Hosking, A. L., Nystrom, N., Whitlock, D., Cutts, Q. & Diwan, A. (2001), 'Partial redundancy elimination for access path expressions', *Software Practice and Experience* **31**(6), 577–600.

Knoop, J., Rüthing, O. & Steffen, B. (1994), 'Optimal code motion: Theory and practice', *ACM Transactions on Programming Languages and Systems* **16**(4), 1117–1155.

Lhoták, O. (2003), Spark: A flexible points-to analysis framework for Java, Master's thesis, School of Computer Science, McGill University, Montreal.

Lhoták, O. & Hendren, L. (2003), Scaling Java points-to analysis using Spark, *in* G.Hedin, ed., 'Compiler Construction, 12th International Conference', Vol. 2622 of *LNCS*, Springer, Warsaw, Poland, pp. 153–169.

Milanova, A., Rountev, A. & Ryder, B. G. (2002), Parameterized object sensitivity for points-to and side-effect analyses for Java, *in* 'International Symposium on Software Testing and Analysis', pp. 1–11.

Morel, E. & Renvoise, C. (1979), 'Global optimization by suppression of partial redundancies', *Communications of the ACM* **22**(2), 96–103.

Pechtchanski, I. & Sarkar, V. (2001), Dynamic optimistic interprocedural analysis: a framework and an application, *in* '16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications', pp. 195–210.

Porat, S., Biberstein, M., Koved, L. & Mendelson, B. (2000), Automatic detection of immutable fields in Java, *in* 'Proceedings of CASCON 2000'.

Razafimahefa, C. (1999), A study of side-effect analyses for Java, Master's thesis, School of Computer Science, McGill University, Montreal.

Rountev, A. (2004), Precise identification of side-effect-free methods in Java, *in* '20th IEEE International Conference on Software Maintenance', pp. 82–91.

Rountev, A. & Ryder, B. G. (2000), Practical points-to analysis for programs built with libraries, Technical Report DCS-TR-410, Rutgers University.

Rountev, A. & Ryder, B. G. (2001), Points-to and side-effect analyses for programs built with precompiled libraries, *in* '10th International Conference on Compiler Construction', Vol. 2027 of *LNCS*, pp. 20–26.

Ryder, B. G., Landi, W. A., Stocks, P. A., Zhang, S. & Altucher, R. (2001), 'A schema for interprocedural modification side-effect analysis with pointer aliasing', *ACM Transactions on Programming Languages and Systems* **23**(2), 105–186.

Sreedhar, V. C., Burke, M. & Choi, J.-D. (2000), A framework for interprocedural optimization in the presence of dynamic class loading, *in* 'ACM SIGPLAN '00 Conference on Programming Language Design and Implementation', pp. 196–207.

Sălcianu, A. & Rinard, M. (2004), A combined pointer and purity analysis for Java programs, Technical Report MIT-CSAIL-TR-949, MIT.

Tkachuk, O. & Dwyer, M. B. (2003), 'Adapting side effects analysis for modular program model checking', *SIGSOFT Softw. Eng. Notes* **28**(5), 188–197.

Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E. & Co, P. (1999). Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot`.

Vivien, F. & Rinard, M. C. (2001), Incrementalized pointer and escape analysis, *in* 'ACM SIGPLAN '01 Conference on Programming Language Design and Implementation', pp. 35–46.

Whaley, J. & Rinard, M. (1999), Compositional pointer and escape analysis for Java programs, *in* '14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications', pp. 187–206.