DYNAMIC PURITY ANALYSIS FOR JAVA PROGRAMS

by

Haiying Xu

School of Computer Science
McGill University, Montréal

June 2007

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

The *pure* methods in a program are those that exhibit functional or side effect free behaviour, a useful property of methods or code in the context of program optimization as well as program understanding. However, gathering purity data is not a trivial task, and existing purity investigations present primarily static results based on a compile-time analysis of program code. We perform a detailed examination of dynamic method purity in Java programs using a Java Virtual Machine (JVM) based analysis. We evaluate multiple purity definitions that range from strong to weak, consider purity forms specific to dynamic execution, and accommodate constraints imposed by an example consumer application of purity data, memoization. We show that while dynamic method purity is actually fairly consistent between programs, examining pure invocation counts and the percentage of the bytecode instruction stream contained within some pure method reveals great variation. We also show that while weakening purity definitions exposes considerable dynamic purity, consumer requirements can limit the actual utility of this information. A good understanding of which methods are "pure" and in what sense is an important contribution to understanding when, how, and what optimizations or properties a program may exhibit.

Résumé

Les fonctions purs dans un programme sont ceux qui démontre un comportement sans fonctionnalité ou effet secondaire. Ceci s’avère une propriété utile pour une fonction ou du code dans le contexte d’optimisation et de compréhension du programme. Cependant, récolter de l’information de pureté n’est pas une tâche facile, et les techniques existantes pour les analyses de pureté ne fournissent que des résultats statiques basés sur une analyses de la compilation du programme. Nous avons exécuter une analyse détaillée de la pureté dynamique des fonctions dans des applications Java en utilisant une approche basés sur un Java Virtual Machine (JVM). Nous avons évalué multiples définitions de pureté, forte et faible, et considéré les formats de pureté spécifiques à l’exécution, tout en considérant les contraintes qui nous sont imposées par un application consommateur d’information de pureté et de mémorisation. Nous démontrons que malgré la consistance de la pureté dynamique des fonctions parmi certains applications, l’examen du nombre d’invocation pure et le pourcentage de chaîne d’instruction bytecode trouvé dans les fonctions purs nous dévoile l’existante de grande variation. Nous montrons aussi que malgré l’affaiblissement de la définition de la pureté expose considérablement la pureté dynamique, les pré-requis des consommateurs peuvent actuellement limiter l’utilité de cet information. Une bonne compréhension de ce qu’est une fonction “pure” et dans quel sens, est une important contribution à comprendre quand, où, et quelles optimisations ou propriétés une application peut dévoilée.

Acknowledgements

I am very grateful to my advisor, Professor Clark Verbrugge, for his constant help, guidance and encouragements throughout this research work. He was always very kind, patient, and willingness to listen to my ideas. I learned a lot from his valuable comments, and most useful advice. Thank you, Clark.

I would like to thank Professors Laurie Hendren for leading the Sable Research Group so well. I am very respect her, and benefit a lot from her vast knowledge of compiler technology and interesting lectures.

I have a very good time in the Sable Research Group and would like to thank all members in Sable Research Group, in no particular order, Ahmer Ahmedani, Michael Batchelder, Eric Bodden, Diana Gheorghiu, Dayong Gu, Richard Halpert, Chris Pickett, Gregory Prokopski, Nomair Naeem, Sokhom Pheng, Dehua Zhang, Peng Zhang, Xun Zhu. Thank you Alexandre Denault for translating the thesis abstract into French. In particular, I would like to thank Chris Pickett for his continues help on my research. His consistent help and advice are invaluable for me.

Finally, I would like to thank my family for their support and encouragement over the past years.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

In most programming languages, methods can both mutate externally visible state, and access previously available state for input. A *pure* method, depending on the particular definition, either has no externally visible side effects as a result of execution, or the extent of these side effects is limited in some way. The extent to which a pure method depends on previously available state may also be constrained.

1.1 Motivation

The concept of *purity* as a method property has been used in a variety of contexts. It can be useful in program understanding and analysis [DR02], isolating and examining functional or “side effect free” fragments [Rou04], and verification in model checking [CH03, FLL^{+}02]. If pure methods are functional, or static independent, invocation of a pure method will not influence other computations. In model checking knowledge of purity information can be used to reduce the search space by ignoring the interleavings between pure methods, or pure and impure methods. This can help alleviate the problem of state explosion for program verification. When optimizing, improved method purity information allows for less conservative assumptions, and has been used to drive compiler optimization [Cla97, LLH05], novel hardware architectures [BS06], and caching or *memoization* of function calls [HLY00].

Many researchers have use purity information to improve runtime performance of Java program by compiler optimizations [LLH05, Raz99]. Clausen [Cla97] shows the benefit from using purity-based side-effects analysis in dead code elimination and loop-invariant removal without points-to information. Whaley and Rinard [WR99] develop stack allocation optimizations and synchronization removal optimizations that require side-effect properties.

Although these applications demonstrate successful uses of various forms of purity and side effect data, the extent to which programs demonstrate purity has not been fully investigated. Practical exploitation of purity is difficult; subtle language and implementation details exist, and a given usage may impose further constraints. Static analysis have shown the existence of large classes of pure methods [Rou04, SR05], but precise definitions for purity vary. Moreover, static analysis can be quite conservative with respect to runtime behaviour, and the extent to which different kinds of purity are observed dynamically is not clear, nor is whether the different classes of pure methods identified have practical value with respect to application of the information.

We present a detailed examination of method purity in Java programs. We consider several purity definitions that range from strong to weak, and investigate both static and dynamic properties of benchmarks. Our results extend previous work on static analysis, and show that the different forms of purity occur with differing frequencies in a dynamic environment; for statically detectable purity, our dynamic results are significantly less optimistic than previous static data suggest. The experiment data shows that there are few practical opportunities for strong forms of purity. The weaker forms of purity do allow considerably more methods and thus method calls to be dynamically identified, although many methods are small or have other features which make efficient exploitation difficult. However, by introducing a new form of dynamic purity that is not detectable by static analysis, the results are improved, and we observe a significant increase in the amount of available purity.

A purity-specific consumer optimization for purity information contributes constraints on the kind of purity data that is practically useful. To this end, we implement a prototype of a non-trivial method memoization optimization. *Memoization* maps method arguments to return values, and allows for execution of functionally pure methods to be bypassed;

we extend the traditional application of memoization by using dynamic purity data and also considering heap dependences as inputs. This practical and direct application exposes additional constraints that a more abstract model of purity may not consider, and we examine the impact of subtle language and technical concerns on the use of purity information in a VM setting. Of course, our memoization implementation is mainly at verifying our dynamic purity results and ensuring all technical consideration are addressed. As a practical optimization, memoizable methods have stronger further requirements than purity. For improved runtime performance, the memoizable methods should also be repetitive and consume significant amounts of CPU time.

1.2 Contributions

The work in this thesis consists of the design, implementation and deep analysis of different forms of dynamic purity. The results of our experiments are presented using three reasonable, and novel metrics for evaluating behaviours. We also develop a consumer, memoization, for our purity analysis. In summary, we make the following specific contributions in this thesis:

- **Several different purity definitions.**

  We develop and investigate several different purity definitions including ones roughly similar to static approaches as well as a variety of new dynamic purity criteria. We evaluate static purity dynamically, and propose new *moderate*, *weak*, and *once-impure* dynamic purity definitions. Moderate purity criteria is actually still a conservative purity, which does not allow pure methods to access any external heap objects. Weak purity criteria is similar to Rountev's purity definition [Rou04] for side-effect free methods detecting statically. Our once-impure purity is a weak form of purity, that allows a pure method to have an impure operation in its first invocation. This purity criteria can identify some forms of purity that are not observable statically. More general once-impure purity is also possible; reasons for impure designatory are discussed in the experimental results.

- **Dynamic purity analysis.**

We design and develop a dynamic purity analysis framework for Java bytecode in SableVM, a state-of-the-art interpreter for Java bytecodes [Gag02]. To the best of our knowledge, this is the first effort that performs purity analysis at runtime. We also implement an *online escape analysis* for some forms of purity, whose purity depends on the locality of objects. Our analysis is scalable and handles SPECjvm98 at size 100 with acceptable overhead. Our design allows for both online and offline comparisons. An *offline analysis* can be used to explore upper bounds on the use of purity without introducing any overhead from actual purity analysis. An *online analysis* is more flexible and allows purity to be evaluated dynamically on per-input basis. That is, a method can be pure with some particular inputs, but impure with other inputs. Dynamic purity analysis is based on the really executed bytecodes, instead of the whole method body as in static approaches. Given the observed difference between static and dynamic behaviour, it is important to understand dynamic results.

- **Three different dynamic metrics for evaluation.**

  Typically, purity is considered a static method property. A dynamic context requires more detailed evaluation. We present three different metrics for evaluating the extent of dynamic purity: *method purity*, *invocation purity*, and *bytecode purity*. *Method purity* considers the number of pure methods dynamically encountered, *invocation purity* describes the invocation frequency for pure methods, and *bytecode purity* actually demonstrates the product of invocation frequency and method size for pure methods. These represent increasingly detailed data, at greater cost. *Bytecode purity* represents most closely the amount of program execution identified as pure. We apply these metrics to the results of a simple static analysis and multiple dynamic analysis corresponding to our purity definitions.

- **A JVM implementation of the memoization.**

  We implement in SableVM an obvious consumer of purity information, memoization. Our memoization is based on the purity information from our *once-impure* analysis. Practical requirements for this as an optimization limit the pragmatic value of purity information. Nevertheless, it serves as a useful functional test module and a good basis for future investigations of the use and nature of purity data.

1.3 Organization

The rest of this thesis is organized as follows. Chapter 2 discusses related work on purity analysis and memoization. For purity analysis, we start by discussing side-effect analysis, which is the base for purity analysis, because the pure method are actually a method without any side-effect. Related work on purity analysis is then discussed in detail.

In Chapter 3, we present different forms of purity we investigate. These purity forms include: *strong purity*, *moderate purity*, *weak purity* and *once-impure purity*. They are discussed in order of conservativeness; each purity form is a weaker form of the purity discussed before.

In Chapter 4, we describe our static analysis and dynamic evaluation environments for *strong purity*. Our static analysis is implemented in Soot, a program analysis framework. The static analysis result is evaluated at runtime in SableVM. Some properties only can be explored at runtime, for example, the invocation frequency, and the method size of pure methods.

In Chapter 5, we implement dynamic purity analyses for all forms of purity discussed in Chapter 3. Dynamic purity analysis can find some forms of pure methods that cannot be identified statically. The dynamic method purity is determined by the actually executed instructions. For all but *strong purity*, *runtime escape analysis* is required to determine the locality of objects, which is the basis for these dynamic analyses. *Runtime escape analysis* is potentially expensive, because we need to check the locality for all objects created during program execution, which itself requires maintaining a complex local object table. Gains in memoizability are then partially affect by implementation costs.

In Chapter 6, we illustrate the basic design of our memoization system, the prototype consumer application for purity data. This design has the added benefit of stress-testing our purity system — improperly memoized methods tend to produce crashes. We developed both *online memoization* and *offline memoization* forms. In *online memoization*, per-input based memoization are possible, but the overall performance will be reduced by the overhead from purity analysis. *Offline memoization* can be used to explore the upper bounds on the performance of memoization without introducing any overhead from purity analysis.

Chapter 7 provides experimental data for static analysis, dynamic analysis, and memoization. The result of purity analysis is described in our three dynamic metrics: *method purity*, *invocation purity*, and *bytecode purity*. Through the comparison between the experimental results from static and dynamic analysis based on the same purity form. We know that the dynamic version could identify more *pure* methods than its static version. And the more flexible the purity form is, the more useful *pure* methods can be found; this is especially true for *weak purity*, for which we get a great improvement compared with *moderate purity*. Although *method purity* is fairly similar between benchmarks, differences in *invocation purity* and *bytecode purity* are greater. From experiments, we know that the purity analysis overhead itself is significant, but compared with existing static analysis, it actually is tolerable.

Finally, in Chapter 8, we conclude and discuss future work on this topic.

Chapter 2
Related Work

This chapter presents previous work done on purity analysis and memoization. Purity analysis itself is a form of side-effect analysis, and so we first present related work in this area. Purity analysis is covered in the second section, while the third section discusses prior work on memoization.

2.1 Side Effect Analysis

Purity definitions are based on the kinds of operations performed in a method, and in particular which classes of data are read and written. Detecting writes in precise detail is typically the task of a *side effect analysis*, a well-known analysis in compiler optimization [Ban79]. Early work on side effect analysis concentrates on determining read and write sets in the context of functional or procedural languages [Bur90, CK88, JG91].

As one of the earliest work on side effects analysis, Banning's algorithm [Ban79] only considered imperative languages and aliasing through parameter passing. The analysis is flow-insensitive and context-sensitive, including two separate flow-insensitive calculations on the call multigraph, one for side effects, another for aliases. Cooper and Kennedy [CK88] improve Banning's approach by dividing the side effect problem into two subproblems: side effects to reference parameters and side effects to global variables. Analysis for reference parameters is based on a graph they call the *binding multi-graph*, which shows the relations between the program's parameters and the call sites. Through this graph, we can find out whether a parameter is modified during a procedure invocation. If a global variable is modified by a procedure, this modification must happen in this procedure or in the call chain after this procedure. Burke showed that these two subproblems for side-effects can be solved by a similar problem decomposition [Bur90]. All these works targeted `FORTRAN77`, a language without pointers.

Modern languages introduce additional concerns through the use of explicit pointers, objects, virtual method dispatch, and intensive use of dynamic memory. Points to analysis assumes a more important role and potentially limits results in such languages. Points-to analysis itself has been studied for a long time [Ste96, WR99, HP00]; and many researchers have considered using points-to analysis for side effect analysis [RLS+01, RR01, MRR02]. Steensgaard [Ste96] developed a flow and context insensitive inter-procedural points-to analysis for C. His analysis is based on type inference, and the principles for this analysis are described through types and typing rules. The main advantage of Steensgaard's points-to analysis is the near linear running-time, which make analysis for large programs possible. Ryder implemented and compared two different algorithms for interprocedural modification side-effects analysis for C program [RLS+01]; one is flow and context sensitive, and another is flow and context insensitive. After comparing these two analysis on cost and precision, she shows that on average the flow- and context-sensitive form will be about 20% more precise than the flow- and context-insensitive version. Unfortunately, the performance for the flow- and context-sensitive approach is also at least in an order of magnitude slower.

More recently side effect analysis has been investigated specifically in the context of Java programs. Razafimahefa presents two different approaches for side-effect analysis in Java program [Raz99]. One is called *type-based analysis*, and another is called *refers-to analysis*. The former uses type information to describe the effect of instructions on variables in the program, and is similar to work done by Clausen [Cla97]. In *type-based analysis*, two variations are considered: class-based and field-based; both of them are based on conservatively assumptions. *Type-based analysis* can be used as a cheap way of approximating the alias relationships. *Refers-to analysis* is derived from Steensgaard's points-to analysis [Ste96] for C, and aims at refining expressions with the same object state. In order to know the refers-to relationship in programs, a model of the program's storage shape graph is built. There are two kinds of nodes in this graph: *reference nodes*, and *abstract heap location nodes*. The *reference nodes* in the graph represents a reference variable in the program, and the *abstract heap location nodes* represent objects that are allocated on the heap during program execution. The edges in the graph will show the relationship between the variable and the object in memory. Therefore, once we get the storage shape graph by refers-to analysis, it is possible to find the side-effect of a statement. Milanova *et al.* explore the use of context sensitive points-to information on side effect information [MRR05]. They develop an *object-sensitive* points-to and side effect analysis, demonstrating the significant impact precise program information can have on side effect data. On the other hand, Le *et al.* show that even reasonably simple points-to information given to a side effect analysis is sufficient to achieve a useful increase in performance, improving the effect of optimizations that use side effect information [LLH05]. These works all focus on precisely identifying read and write sets, and not of course on identifying different notions of purity *per se*.

2.2 Purity Analysis

A study that bridges some of the gaps between purity and side effect analysis is one by Clausen [Cla97]. His work is based on a conservative, static, side effect analysis of bytecode, identifying four classes of instruction and therefore method, arranged in a partial order: *pure*, neither reading nor writing data, *read-only*, only reading data, *write-only*, only writing data, and *read/write* as the least pure. Purity classes are global, exploiting neither type nor points-to information, although this is recognized as a limitation. Clausen demonstrates the impact of this purity information on several standard compiler optimizations. Importantly, Clausen also points out the impact of practical concerns in purity analysis, including the over-identification of pure methods due to language mechanisms such as Java's `<init>`, an empty inherited method in most object instances. In Chapter 5, we further discuss these concerns, particularly in the context of dynamic execution. We also provide a new analysis of impure instructions, informed by our previous work on speculative multithreading for Java [PV05].

Method purity criteria have also been considered in the context of program specification and verification. The Java Modeling Language (JML) is a behavioral interface specification language for Java [BCC$^+$05, LBR06]. JML provides a definition of a pure method as one which does not: 1) perform I/O; 2) write to any pre-existing objects; or 3) invoke any impure methods. However, JML is annotation-based, requiring purity information be provided by users. Static verifiers do exist, although current designs check purity information conservatively [CH03].

Side effect free methods are identified as a form of purity where externally visible writes are not allowed, but reads are permitted. Rountev develops a static analysis to detect side effect free methods, and evaluates the impact of different call graph construction algorithms on detecting these methods [Rou04]. He finds that 22% of methods are side effect free. In comparison with the purity definition given by JML, Rountev's purity definition is conservative. Side effect free methods must guarantee matching pre- and post-states, disallowing them from creating and returning new objects, although they can allocate objects locally. Rountev's work can detect side-effect-free method in incomplete programs using rapid type analysis, which is inexpensive and context insensitive. After comparison with some context sensitive analysis, Rountev proved that this rapid type analysis has near perfect precision in statically detecting side-effect-free methods.

Sălcianu and Rinard [SR05] present a purity analysis based on a previous points-to and escape analysis [WR99]. Their purity definition is much the same as the purity definition given by JML: a pure method can read from or write to local objects, and can also create, modify and return new objects not present in the input state. This allows Sălcianu and Rinard to identify more statically pure methods, 53–65% of methods in their benchmark suite. Compared with Rountev's work, many more pure methods can be found in Java program by removing the restriction on precisely matching the program's its post-state with pre-state. Since there are fewer restrictions on the post-state, a pure method can mutate a new allocated object, and return this new object to its caller. Their analysis also provides some additional information, for example, detecting safe parameters, and regular expressions describing the externally objects mutated by impure methods.

Our work here is partly motivated by an interest in finding the extent to which static results for purity analysis are indicative of dynamic behaviour. A large number of statically identified pure methods suggests a significant optimization opportunity, but only if these methods are both reached and well-exercised at runtime; previous work on dynamic metrics has shown the importance of observing actual runtime behaviour in Java programs [DDHV03], for example, size, data structure, memory use. concurrency, and polymorphism.

Dallmeier *et al.* have examined dynamic purity analysis for Java programs concurrently with this work [DLZ07]. Their `jdynpur` tool uses the ASM bytecode manipulation framework [BLC02] to create program traces, and identifies impure methods based on writes to non-local objects. They also provide a means to compare static and dynamic purity information. Interestingly, they can merge purity information from across different program executions. As of this writing, this related work is in an early phase.

Artzi *et al.* have examined in greater depth a closely related topic that is also concurrent with this work, namely dynamic analysis of parameter mutability for Java programs [AKGE07]. Initially, reference parameters are classified as *unknown* with respect to mutability. A static analysis in Soot [VR00] provides a conservative classification of parameters as *mutable* or *immutable* where possible, and then a dynamic analysis detects further parameter mutability. Their work differs significantly from ours in that they combine static analysis with dynamic analysis in various multi-stage pipelines, and then evaluate the results using static accuracy metrics. Furthermore, although parameter immutability is one aspect of method purity, there may be other factors involved, some of which depend on the consumer.

2.3 Memoization

An important use of purity information is in its application. We demonstrate a practical consumer for purity information through an implementation of method *memoization*. Our design for this in Java, interfacing with purity data and runtime analysis is novel, but the idea of *memoizing* or caching function results is of course quite old, and the basic memoization optimization was first introduced by Michie [Mic68]. Similar ideas have been used for developing dynamic algorithms [Fre97, Mul91, ST81], and, more within the programming languages community, for incremental computation [PT89, ABH03]. Several works have looked at improving function memoization efficiency [ALL96, HLY00, LT95] in the context of functional languages, with most designs based on the strong purity requirements that functional languages provide. Our memoization design is inspired by some of our earlier work on adapting memoization for use by Java-based return value prediction [PV04].

Acar, Blelloch and Harper [ABH03] presents a framework for selective memoization in the context of a small functional language, MFL. Based on this framework, programmers can control the equality, space usage, and identification of dependences to obtain the best performance. Equality tests are critical to the overall performance, and if input arguments are complicated this kind of test could be very expensive. In such situations, the scheme can decide to perform a non exact test, only performing "location" equality tests, or directly skipping this test depending on how expensive the test is. Acar et al. give further programmer control over equality testing in memoization tables, allowing for arbitrary replacement policies [Pug88] in memoization caches.

Memoization for modern languages, such as C, C++, or Java is, for obvious reasons, much more complicated than that in functional languages. Achieving good performance, however, includes similar concerns as finding appropriate memoizable methods or code segments. Ding [DL04] introduce a schema for detecting good candidates for computation reuse in C program. This schema is based on a *cost-benefit analysis* to identify expensive code segments, for example, nested code. Through profiling techniques, information about execution frequencies and input value set repetition for these code segments are collected. They examine the conditions: if the memo table look-up costs is less than repeating code execution, the method is skipped; otherwise, repeating execution is performed. McNamee and Hall [MH98] also developed a tool for memoization in C++. It is based on annotation, which unfortunately requires program rewriting for those methods in which memoization is desired. To the best of our knowledge, our memoization in Java program is the first effort for memoization based on purity information or other high-level data.

Chapter 3
Different Purity Definitions

There can be different levels of purity under different situations, and we consider four levels of dynamic purity which we term *strong*, *moderate*, *weak*, and *once-impure*. These represent a range of exploitable properties of use to program analysis and optimization. In this chapter, we will introduce these different definitions in order of conservativeness; each purity forms is a weaker form of the purity discussed before.

3.1 Strong Purity

Our *strong purity* is a very conservative purity form. Basically, a method that has no side-effects whatsoever is considered pure; even temporary side-effects that not really visible to the caller are not allowed. For example, a method may create an object with purely local scope, never returning it to its caller. Although this operation has no observable side-effect, it has a side-effect on the heap, and it is still considered *strongly* impure.

A method is *strongly pure* iff: it

- does not create any object,
- does not read from or write to any object or static variable,
- does not perform any synchronization,
- does not invoke any native methods,

- does not invoke any strongly impure methods,

In summary a *strongly pure* method cannot touch the heap at all. Therefore, for non-functional languages, a *strongly pure* method will look much more like a functional method: all variables in the method should be primitive, either coming from parameters or the primitive variables declared locally. For example, the program shown in Figure 3.1 is obviously *strongly* pure. Methods in Figure 3.2, however, are all *strongly* impure: the constructor `Obj()` is strongly impure because of writing to the field *f*, `bar()` is strongly impure because of the new object allocation at line 8, `foo()` is strongly impure because of the invocation of impure method `bar()` and reading from an object field, `baz()` is strongly impure because of reading from heap, and `baf()` is strongly impure because of new object allocation at line 20.

```
int foo(int a){ //strong pure
  int b=10;
  return a+b;
}
```

Figure 3.1: *Examples for strong purity.*

3.2 Moderate Purity

Java is an object-oriented language, and the restrictions in *strong purity* that any object operation not be allowed are quite strong. In order to find more useful pure methods, we must allow some kind of object operation. One approach that retains the side effect free property of pure methods is to allow objects to be created and then altered in a pure method, provided such objects do not *escape* the method execution context.

A *moderately pure* method may not:

- read from or write to static or previously existing heap objects,

- perform monitor operations,
- invoke native methods,
- throw exceptions,
- call an impure method, unless the only source of impurity is that the callee method accesses and mutates objects local to the caller,
- allow an object to escape to the caller.

A *moderately* pure method can allocate a new object, provided this new allocated object does not escape. Consider again the program in Figure 3.2, Both `bar()` and `baf()` are *moderately* impure because of objects escaping the local context. The local object in `bar()` escapes by returning the new allocated object to its caller, and the new created object in `baf()` escapes by being stored in a reference field.

A *moderately* pure method can only access *local* objects. An object is considered *local* if this object is created after the method invocation, and does not escape during method execution. The object *o* in method `foo()` of Figure 3.2 is a local object to method `foo()`, and the object *o* for method `baz()` is non-local, because it comes from parameters. Therefore, accesses to the object in `foo()` will not influence its purity, but accesses to the non-local object in `baz()` will make method `baz()` *moderately* impure. The class constructor `Obj()` is also *moderately* impure; it writes to a field of an external object (`this`).

A *moderately* pure method only can call an impure method provided the impurity is contained within the calling context. For example, in Figure 3.2, `foo()` is still considered moderately pure despite calling the *moderately* impure method `bar()`. The impurity for `bar()` is due to an escaping object, but this object is still contained within the calling context of `foo()` leaving `foo()` moderately pure.

In summary, our *moderately* pure methods provide a "contained side-effect" property. With the additional constraint that a pure method does not change behaviour based on the input heap or global state, the result is that the behaviour of a moderately pure method is determined exclusively by its primitive input arguments.

```
class Obj{
  int f;
  Obj x;
  public Obj(){ //strongly impure
    f = 10;
  }
  Obj bar(){ //strongly impure
    Obj o = new Obj();
    return o;
  }
  int foo() { //moderately pure
    Obj o = bar();
    return o.f;
  }
  int baz(Obj o) { //weakly pure
    return o.f
  }
  int baf() { //once−impure pure
    if (x == null) {
      x = new Obj(); //escapes by reference field
    }
    return 42;
  }
}
```

Figure 3.2: *Examples for different purity forms.*

3.3 Weak Purity

A further potential definition of purity allows for arbitrary method behaviour provided that the input state is not altered [Rou04]. This permits at least reading from the heap, as well as making contained modifications. Our *weak purity* is a weaker form of *moderate purity*, and the definition corresponds fairly closely with Rountev's [Rou04].

A *weakly pure* method may not:

- write to static or previously existing heap objects,
- perform monitor operations,
- invoke native methods,
- throw exceptions,
- call an impure method, unless the only source of impurity is that the callee method mutates objects local to the caller,
- allow an object to escape to the caller.

The main difference between *moderate purity* and *weak purity* is that *weak purity* can read from external objects. Consider once again the code in Figure 3.2, but based *weak purity*. Here `baz()` is considered *moderately* impure, but *weakly* pure, because the only impurity for moderate purity comes from reading from the external object *o* at line 16. The constructor `Obj()` remains impure, even *weakly* though because it writes to the field of an external object, as shown at line 5. Similarly, the `bar()` and `baf()` are *weakly* impure due to objects escaping; the object in `bar()` escapes by being returned, and the object in `baf()` escapes by being written to an external reference field.

3.4 Once-impure Purity

It is possible that the behaviour of a method may be different on its first invocation for setup or initialization reasons. *Once-impure purity* purity is equivalent to weak dynamic purity, except that the first invocation of the method during execution may be impure. All subsequent invocations must, however, be weakly pure. Consider the program in Figure 3.3, `bar()` actually is *weakly* impure because it writes to an external object *obj*. However, in this context it only acts in an impure fashion on its first invocation, and thereafter acts weakly pure. A similar situation can be seen in the `baf()` code of Figure 3.2; if *x* is *null* only once in the program's lifetime then `baf()` is once-impure.

```
static void main foo(String[] args)
{
  int sum=0;
  Object obj = new Object();
  ...
  obj.f = 0;
  while()
  {
    sum += bar(obj);
  }
}
static int bar(Object obj) //once−impure pure, weakly impure.
{
  if(obj.f == 0)
  {
    obj.f = 10;
  }
  return obj.f
}
```

Figure 3.3: *Examples for once-impure purity.*

3.5 Summary for Purity Definitions

In Table 3.1, we summarize all purity forms we have discussed in this Chapter. Each purity is a weaker form than its previous line. None of instructions related to *static*, *monitor*, *exception* and *native* are allowed, but allowed once for *once-impure* pure methods. The main difference for different purity forms is in the access to heap and impure method invocation. Heap reading is not allowed at all in *strong purity*, allowed partially in *moderate purity*, and in an unlimited sense for *weak purity* and *once-impure purity*. Heap writing is much more conservative, depending on the object's locality for all other forms of purity except *strong purity*. With the exception of *strongly* pure methods, all forms of pure methods are allowed to invoke impure methods from the same or stronger purity forms, if the only source of impurity is that the callee method accesses objects local to the caller. Thus, a *moderate* pure method may invoke *strongly* or *moderately* impure methods, and a *weakly* pure method may invoke *strongly*, *moderately*, or *weakly* impure methods.

	heap read	heap write	invoke impure	static	monitor	exception	native
strong	no	no	no	no	no	no	no
moderate	maybe	maybe	moderate	no	no	no	no
weak	yes	maybe	weak	no	no	no	no
once-impure	yes	maybe	once-impure	only 1st	only 1st	only 1st	only 1st

Table 3.1: *Different levels of purity.*

Chapter 4
Static Purity Analysis for Strong Purity

Our static work looks for the existence of *strong* purity in Java programs; the definition for the *strong purity* is given in Section 3.1 page 13. A strongly pure method may not read or write the heap or static data, perform synchronization or allocate memory, or conservatively invoke any native method or method not itself strongly pure. This corresponds with forms of purity found in functional languages.

Our investigation into static purity is roughly divided into static analysis and dynamic experimental designs. Our static implementation is based on the Soot program analysis framework [VR00]. And the dynamic evaluation of that static analysis is developed in SableVM, a Java bytecode interpreter [Gag02].

4.1 Overall Framework

Our strong static purity analysis is performed in Soot [VR00], a framework for analyzing, optimizing, and annotating the Java bytecode. The left half of Figure 4.1 shows the overall framework used to evaluate strong purity in a static setting. Java class files are used as input, and we perform a flow-insensitive analysis in Soot. In order to evaluate our strong purity property at runtime, a simple extension of this design is introduced. We use Soot to write out purity information to Java class file as method attributes. This allows SableVM to read this modified class file during class loading, as shown in the right half of Figure 4.1.

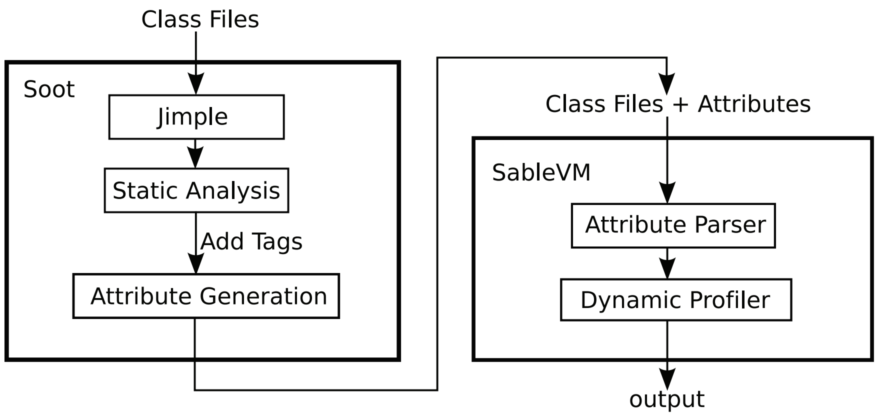


Figure 4.1: *Implementation framework for static analysis.*

4.2 Static Analysis in Soot

Our analysis is flow-insensitive, and will scan all statements in all branches in the method. The first four criteria for *strong purity* in Section 3.1 at Page 13 involve intra-procedural analysis, and the last requires an inter-procedural analysis.

For intra-procedural analysis, the statements can be pure, impure or unknown. A summary of impure statements is given in Table 4.1. Any branch containing any impure instruction will make the method impure. The purity for those methods that do not contain any impure statements but `INVOKE*`, which is treated as unknown statement, depends on the inter-procedural analysis. Only when all its callees are known to be pure is the caller considered pure. Except for impure and unknown statements, all other statements are pure. Statically we assume that exceptions do not propagate unchecked.

A pure method cannot invoke any impure methods, and so we use an inter-procedural analysis to guarantee this criteria. Our inter-procedural analysis is also flow-insensitive, and depends on the call graph construction in Soot [VR00], which itself is based on Class Hierarchy Analysis (CHA). This CHA-based call graph construction is inexpensive because its analysis does not require any flow-sensitive or context-sensitive analysis, and this has the advantage of being safe: a CHA-based call graph will certainly contain all possible methods reachable at runtime assuming the same CLASSPATH of course. Considering the polymorphism inherent to Java language, we expect the methods in call-graph will contain

Class	Statement	Semantic
New	obj `= new C`	create a new object of class C
	v `= new C[k]`	create a array of k reference
Store	obj`.f =` v	store a reference into an object field
	`C.f =` v	store a reference into an static field
	$v_1[i]$ `=` v_2	store a reference into an array cell
Load	v `=` obj`.f`	load a reference from an object field
	v `= C.f`	load a reference from a static field
	v_2 `=` $v_1[i]$	load a reference from an array cell
Monitor	`MONITOR ENTER`	set the lock;
	`MONITOR EXIT`	unset the lock;
Native	`INVOKE NATIVE`	invoke a native method

Table 4.1: *Impure statements for strong purity.*

more nodes and edges than the call graph of a program obtained at runtime.

Figure 4.2 is the pseudo-code for our inter-procedural analysis. First, all nodes in the call-graph are placed onto the worklist in pseudo-topological order. For each node in the worklist, we remove it from the worklist, and perform intra-procedural analysis on it. If the node is pure, we mark the node as pure; otherwise, we mark this node and all its callers as impure. If, due to method calls, we still do not know whether this method is pure or not, we put it back to the end of worklist. This procedure continues until a least fixed point is reached; that is, each method is processed at least once and the methods in the worklist do not change any more.

Our inter-procedural analysis is performed by propagating impurity up from the leaves of the call graph, and computing a least fixed point for recursive invocations. An example call graph is shown in Figure 4.4, which is constructed based on the program in Figure 4.3. The left hand side of Figure 4.4 shows the purity distribution after intra-procedural and before inter-procedural analysis. Propagating impurity of `c()` up from the leaves to the root will make both `a()` and `foo()` impure, as show in the right hand side of Figure 4.4. Figure 4.5 shows the worklist content as the algorithm proceeds, and the remaining strongly connected component `(b, d, e)` is designated pure.

```
put all nodes in the call graph into the worklist w
while(w is not empty)
{
   w_old = w;
   for each method m in w
   {
      remove m from w;
      perform intra−procedural analysis for m;
      if(m is pure)
         mark m as pure;
      else if(m is impure)
      {
         mark m as impure;
         remove all callers of m from w;
         mark all callers of m as impure.
      }else{
        /* the information for the purity of m is not enough*/
        add m to the end of worklist w;
      }
   }
   if(w_old equals w) //only strong connected components left
      break;
}
/*all methods inside strong connected components are pure*/
if(w is not empty)
   for each method s left in w
        mark s as pure;
```

Figure 4.2: *Pseudo-code for inter-procedural analysis.*

```
void foo( )
{ ...
  if(...)
  { a();}
  else
  { b();}
}
void a( )
{ ...
  if(...)
  { c();}
  else
  { d();}
}

void b( )
{ e();}

void d()
{ b();}

void e( )
{ d();}

void c( )\\impure.
{ o = new Object();}
```

Figure 4.3: *Example programs for inter-procedural analysis.*

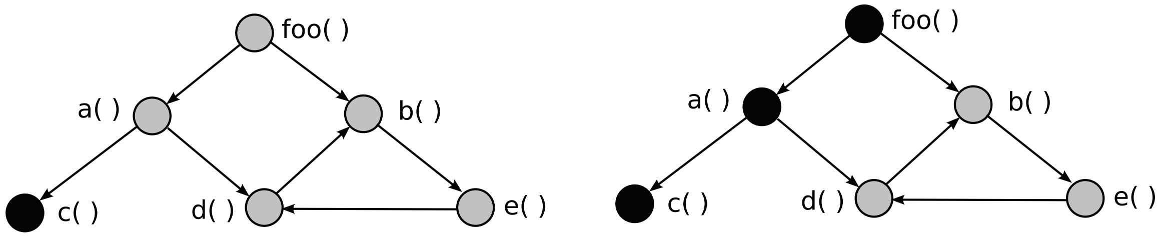


Figure 4.4: *Call graph for inter-procedural analysis.* Black nodes are impure methods, and grey nodes are pure methods. Left and right hand graphs show the before and after result of propagating the impurity of `c()` to its callers.

1: foo, a, b, c, d, e
2: a, b, c, d, e, foo
3: b, c, d, e, foo, a
4: c, d, e, foo, a, b
5: d, e, b
6: e, b, d */∗fixed point for strong connected component∗/*

Figure 4.5: *An example for worklist iteration.*

4.3 Encoding Class Attributes

Runtime Java behaviour may differ from static, conservative estimates. The call-graph might contain more nodes and edges than what we may obtain at runtime, and the purity properties at runtime may be very different from what we see in a static analysis. In order to evaluate the purity properties more precisely, we thus need to evaluate it at runtime. Through runtime evaluation we also can obtain some purity properties that we cannot get statically. For example; How frequently are pure methods invoked? How much bytecode or actual execution time is pure? What is the method size for those pure methods? Soot has a tagging framework [PQVR+01], which can encode our static purity information into class file as method attributes. Reading this attribute at class loading at time is easy. Figure 4.6 shows the data structure for our method attribute. The *attribute_name_index* is a 2 byte unsigned integer value indexing the name of the attribute in the class file's *Constant Pool*, and *info* is the actual purity information. Figure 4.7 gives an example of the attributed bytecode: attached to the end of the `getSum()` method is a purity attribute encoded as *.method_attribute*, and storing the binary data that *getSum()* is pure.

```
method_purity_attribute
{
    u2 attribute_name_index;
    u4 info;
}
```

Figure 4.6: *Data structure for the method purity attribute.*

```
public int getSum(int x, int y)
{
   return x + y;
}
```

```
public int getSum(int, int);
    0: iload_1
    1: iload_2
    2: iadd
    3: ireturn
.method_attribute org.sablevm.purity=1
```

Figure 4.7: *Source and attributed bytecode.* "1": method is strongly pure; "0": method is impure.

4.4 Runtime Evaluation of Strong Purity Property

After writing out purity information from static analysis to Java class file attributes, it is read into the SableVM Java virtual machine during class loading as shown in the right half of Figure 4.1. First we modify the parser to parse the purity attribute in the Class file. And the purity attribute will be stored into the attribute tables. Then for each method, some counters are added, including the number of pure methods reached at runtime, the number of pure method execution, and the bytecodes executed in pure methods. All those counters will give a sense of how well static results correlate with dynamic behaviour. Experimental results of our technique are shown in Chapter 7. In the next chapter we show extensions to our dynamic system that allow dynamic forms of purity to be calculated and compared with our static results, and each other.

Chapter 5
Dynamic Purity Analysis

Previous work has established that statically a significant number of methods have moderate [Rou04] or weaker [SR05] purity properties. Dynamic analysis is a way of complementing existing static purity analysis work. By defining and observing purity as a dynamic property we hope to gain more insight into useful forms of method purity. Below we first give a detailed motivation for dynamic purity followed by descriptions of the various forms we investigate in this thesis.

5.1 Motivation

Statically, a method is conservatively determined to be pure for all possible executions; if not it is necessarily impure. However, for a given program run, a method declared impure statically may actually exhibit only pure control flow. Consider the example code shown in Figure 5.1. From a static perspective, the method `foo()` must conservatively be considered impure. If, however, t is always $>= 0$ at runtime, `foo()` will be dynamically pure. *Dynamic purity analysis* helps identify methods as pure or not based on their actual runtime behaviours, increasing the number of pure methods identified.

```
int x;
public int foo(int t){
  if(t<0)
     return x; //strong impure path
  else
     return t; // pure path
}
```

Figure 5.1: *An example of a method that is impure statically, but maybe strongly pure dynamically.*

5.2 Overall Framework

Figure 5.2 shows our framework for performing, using, and evaluating dynamic purity analysis. Initially, class files are read into SableVM, and method purity is determined by examining the executing instruction stream. Whether a method is considered pure depends on the allowed instructions and system states, and the purity analysis module may even employ an *online escape analysis* sub-module that tracks reads or writes to locally allocated objects. The details of the purity definitions we investigate are detailed in the sections below, from stronger to weaker forms. Purity information can be used to drive client applications, for example, *memoization* to reduce the overhead from dynamic purity analysis. We will discuss this in Chapter 6. Experimental results from our analysis will be presented in Chapter 7.

5.3 Strong Dynamic Purity.

Strong dynamic purity has the same criteria as strong static purity. The definition for *strong purity* is given in Section 3.1 at Page 13. However, we now consider only those instructions that are actually executed, as opposed to the entire static method body. A method may remain pure as long as impure instructions or methods are not encountered.

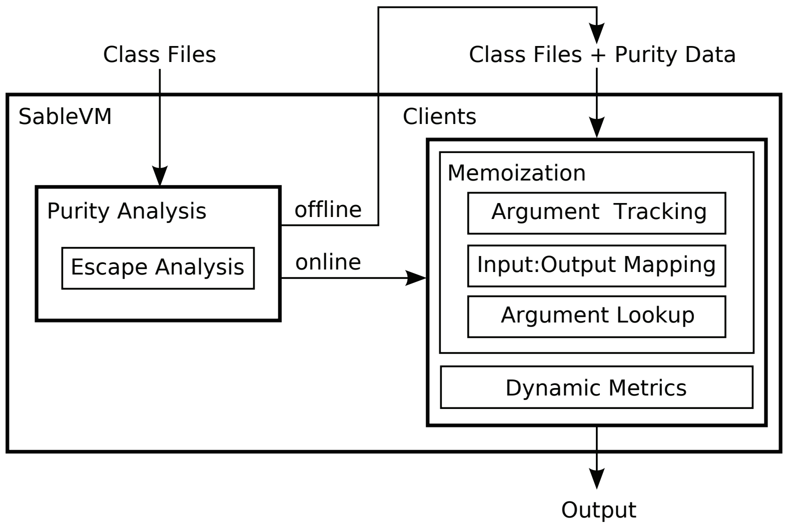


Figure 5.2: *Dynamic purity analysis and memoization.*

Class	Instructions
heap	NEW, NEWARRAY, ANEWARRAY, MULTIANEWARRAY, GETFIELD, PUTFIELD, *ALOAD, *ASTORE
statics	GETSTATIC, PUTSTATIC
synchronization	synchronized INVOKE*, MONITORENTER, MONITOREXIT
exceptions	ATHROW
native methods	native INVOKE*

Table 5.1: *Impure instructions for strong purity.*

The strong purity criteria, reduces to a specific set of impure instructions, as shown in Table 5.1. These impure instructions are divided into five categories according to instruction's function. Initially, all methods have an unknown purity status. As instructions are executed, the status of the containing method is updated if an impure instruction is encountered. Note that the call graph here is dynamic as well. A method found to be impure thus does not update a static set of caller methods, rather it propagate impurity up the call stack to maintain the property that pure methods do not invoke impure methods. Figure 5.4 shows the call stack for program in Figure 5.3, when the program is running line 9. The impure instruction `GETFIELD` in line 9 will make all methods in current call stack impure. So after executing line 9 in Figure 5.3, `baz()`, `bar()`, and `foo()` will be impure. A method is marked as pure if it returns without encountering any impure instruction. At different points in time there will thus be different numbers of strongly pure methods identified. However, once identified as impure, a method conservatively stays impure for the remainder of execution.

Examining the strong purity criteria, we can find that a strongly pure method cannot have any operation related to object, array or field. Java as an object-oriented programming language, has *objects*, and *fields* as fundamental concepts. Therefore, based on these strong purity criteria, we do not expect find many pure methods dynamically.

5.4 Moderate Dynamic Purity.

Our moderate dynamic purity analysis is based on our *moderate purity* discussed in Section 3.2 at Page 14. We make special exceptions for the native `java.lang.VMSystem.arraycopy()` method, which is the native code for SableVM specifically. Otherwise, this method will induce large amounts of impurity. Instead of considering all native methods as unanalyzable methods, which will cause all methods in the call stack impure, we treat this method as an analyzable method, with heap access and allocation instructions, and the purity of these instructions will be decided by our *online escape analysis*.

Based on moderate purity criteria, the instructions from Table 5.1 can be divided into two categories: *impure* and *maybe impure* instructions, as Table 5.2 shows. All other instructions that are not in Table 5.2 are *pure*. The analysis for those *impure* instructions

```
public class Test
{
  Object obj;
  public void foo()
  { bar();}
  public void bar()
  { baz();}
  public int baz()
  { if(obj == null) //contains impure instruction: GETFIELD
        return 0;
      else
        return −1;
  }
}
```

Figure 5.3: *Examples for dynamic strong purity.*

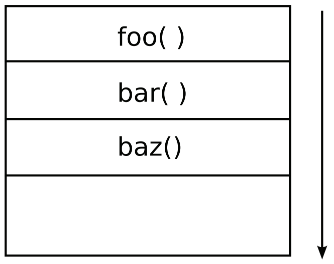


Figure 5.4: *The call stack for example programs.* The stack grows downward.

Class	Instructions
heap *(maybe impure)*	`NEW`, `NEWARRAY`, `ANEWARRAY`, `MULTIANEWARRAY`, `GETFIELD`, `PUTFIELD`, `*ALOAD`, `*ASTORE`
statics *(impure)*	`GETSTATIC`, `PUTSTATIC`
synchronization *(impure)*	synchronized `INVOKE*`, `MONITORENTER`, `MONITOREXIT`
exceptions*(impure)*	`ATHROW`
native methods *(impure)*	native `INVOKE*`

Table 5.2: *Maybe impure and impure instructions for moderate purity.*

is the same as we did for *strong dynamic purity analysis*. And *pure* instructions will not influence the purity of methods. However, for those *maybe impure* instructions, `*NEW*`, `GETFIELD`, `PUTFIELD`, `*ALOAD` and `*ASTORE`, our analysis must now examine the object more closely than was necessary for strong dynamic purity. We need to perform *online escape analysis*.

5.4.1 Online Escape Analysis

The goal of *online escape analysis* is to determine the purity of those *maybe impure* instructions in Table 5.2. After *online escape analysis*, those *maybe impure* instructions must be either *pure* or *impure*. The purity of *maybe impure* instructions is based on object locality. Objects allocated in the current method are *local* if they do not escape the current method; objects allocated by some callee also become local if they escape to the current method. These *maybe impure* instructions will be treated as *pure* instructions only when the target object for the instruction is local. Otherwise, the *maybe impure* instruction will become

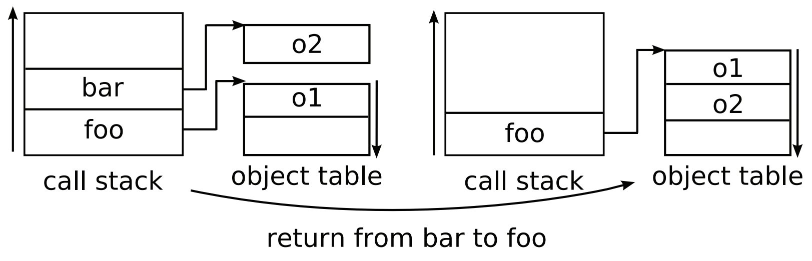


Figure 5.5: *Dynamic escape analysis.*

impure instruction.

Object locality is monitored by storing a *local object table* with each stack frame. The *object table* is created when the method is invoked, and destroyed after the method returns. The *object table* can grow under three situations: first of all, newly allocated objects are stored in the table for the current frame. Secondly, any object allocated and returned by a callee is merged into the object table of its caller. Finally, any callee `PUTFIELD` instruction with a reference argument, if this reference argument is non-local, can allow an object local to the callee to escape to the caller, adding the object argument to the caller's *object table*. Figure 5.5 shows the states for call stacks and object tables for the Java program in Figure 5.6. The left hand side of Figure 5.5 shows the call stacks and their corresponding object tables after `Test.bar()` invocation and before its return. At this time, `Test.foo()` has a new allocated object, *o1*. And `Test.bar()` also has a locally object *o2*. When `Test.bar()` returns, this new allocated locally object is merged into it caller `Test.foo()`, as show in the right hand side of Figure 5.5. `Test.<init>()`, an example for the object escaping through `PUTFIELD` with reference. Before `Test.<init>()` is invoked, an *Test* object will be created in its caller, and inside `Test.<init>()`, a *Point* object will be created, and merged into its caller due to the `PUTFIELD` instruction that stores the *point* into *pp* as a field of `Test` object, which is created in the caller. Therefore, *pp*, the local object of `Test.<init>()`, will escape and become a local object of its caller.

The `GETFIELD`, `PUTFIELD`, `*ALOAD`, and `*ASTORE` instructions can now be easily classified depending on the contents of the object table for the current frame. If a read or write occurs to a non-local object, the stack is searched for that object, marking the current

```
class Point{ public int x, y; }
class Test{
  Point pp;
  public void Test(){
    pp = new Point(0,0);
  }
  int equals(Point p1, Point p2){
    if(p1.x == p2.x && p1.y == p2.y)
      return 1;
    else
      return 0;
  }
  int baz(Point p1, Point p2){
    if(p1 == null || p2 == null)
       return −1;
    else
       return equals(p1, p2);
  }
  Point bar(int x, int y){
    Point o2 = new Point(x, y);
    return o2; //note1
  }
  int foo(int x, int y){
    Point o1 = new Point(x,y);
    Point o2 = bar(x,y);
    int sum = baz(o1, o2); //note2
    return sum;
  }
}
```

Figure 5.6: *Example programs for different purity forms.*

method and all intermediate methods as impure; otherwise the instruction is considered pure. The `*NEW*` will add the new object to the method's object table. It is an impure instruction only if the object created is returned to its caller. Otherwise, `*NEW*` is a pure instruction.

5.4.2 Computing Moderate Dynamic Purity

Consider the example Java program in Figure 5.6. Both `Test.equals()` and `Test.baz()` are impure because of the use of the instruction `GETFIELD` for statements *p1.x, p1.y, p2.x, and p2.y*. Because objects *p1* and *p2* for instruction `GETFIELD` are not in the object table of `Test.equals`, we mark this method as an impure method. After searching the call stack, we know that this object is created at it's caller's caller `Test.foo()`. Therefore, we also mark the intermediate methods of `Test.equals()`, `Test.baz()`, as impure. `Test.bar()` is impure because of `*NEW*`. We know that `*NEW*` can only be treated as pure in the situation that the new created object is not returned to its caller, but here the new object *p* is returned to its caller `Test.foo()`. After `Test.bar()` returns, this new object is added to object table of its caller, `Test.foo()`, as the right part of Figure 5.5 shows. Finally, `Test.foo()` is pure, because there are no impure bytecodes in it, and all *maybe impure* instructions are on its local objects.

5.4.3 Limitation for Moderate Dynamic Purity

Given that external heap reads are disallowed, a moderately pure method often does not have object parameters; or if it does, it is unable to make any use of them outside of object reference comparisons. A moderately pure method also cannot make any use of implicit parameters from method's receiver. In a Java context, this can greatly reduce the observable purity, even given the ability to access and mutate locally allocated objects, many methods read input heap data, and object parameters are common. We thus developed a third and weaker form of purity that permits heap reads.

5.5 Weak Dynamic Purity.

In this section we will discuss how the dynamic purity analysis is performed based on the *weak purity* discussed in Section 3.3 at Page 17. Reading from external objects will alter the input state, but will give some kind of overhead for our purity consumer, *memoization*. When reading external objects is allowed, that means the result of the method may depend on the content of the input objects and when memoizing this method we need to save the content of external objects. This will definitely introduce overhead for memoization, but this may be necessary in an object-oriented language, like Java where useful pure methods are likely to inspect object state to at least some degree. This motivates our *weak dynamic purity*.

Weakening moderate purity by allowing heap reads enables a method to inspect its object parameters and any data structures reachable from them. This maintains the property that the method is functional on its input, even if the input is quite large and in the worst case constitutes the entire heap. The weak purity criteria is showed at the third line of Table 3.1. This purity definition corresponds fairly closely with Rountev's [Rou04]. We make special exceptions for the native `java.lang.VMSystem.arraycopy()` methods as we did for moderate criteria, treating them as heap access and allocation instructions respectively, as these methods otherwise induce large amounts of impurity.

For *weak* dynamic purity, the `GETFIELD` and `*ALOAD` operation are now always safe, and safe, or pure, instructions will not influence the purity of methods. As with moderate purity, *impure* instructions will cause all methods in the current call stack to be marked impure. The instructions `*NEW*`, `PUTFIELD`, and `*ASTORE` must still be considered in the context of our *online escape analysis*, as we did for moderate purity.

As we did for moderate purity, we perform *online escape analysis* to determine the purity of *maybe impure* instructions in Table 5.3. `*NEW*` is only impure when the new allocated object is escaped. `PUTFIELD` and `*ASTORE` are impure when the target object is a non-local object. Now consider the code in Figure 5.6. We know that only `Test.foo()` is pure based on moderate purity, and others impure. Under *weak purity*, however, `Test.equals()` is found to be pure, and thus so are `Test.baz()`, `Test.bar()`, and `Test.foo()`.

Class	Instructions
heap *(maybe impure)*	`NEW, NEWARRAY, ANEWARRAY, MULTIANEWARRAY, PUTFIELD, *ASTORE`
statics *(impure)*	`GETSTATIC, PUTSTATIC`
synchronization *(impure)*	synchronized `INVOKE*, MONITORENTER, MONITOREXIT`
exceptions*(impure)*	`ATHROW`
native methods *(impure)*	native `INVOKE*`

Table 5.3: *Maybe impure and impure instructions for weak purity.*

5.6 Once-Impure Dynamic Purity.

The preceding definitions require purity over the entire course of execution. After examination of the impure methods identified using the weak criteria, we found that some of them are weakly pure, but only after the first invocation.

For example, In Figure 5.7, both `Test.bar()` and `Test.foo()` are weakly impure. When `Test.bar()` is invoked for the first time, *obj* is *null*, and the method executes the impure `PUTFIELD` instruction on an external object. However, different behaviour after initialization is common. In our example, after the first execution, *obj* will not be *null* anymore, and `Test.bar()` will remain pure. Hence both `Test.foo()` and `Test.bar()` are *once-impure* pure methods.

Once-impure dynamic purity is equivalent to weak dynamic purity, except that the first invocation of the method during execution may be impure.

```
public class Test
{
  Object obj;
  public void bar()
  {
    if(obj==null)
          obj = new Object(); //impure because of PUTFIELD
  }
  public void foo()
  {
    for(int i=0; i<5; i++)
        bar();
  }
}
```

Figure 5.7: *Example programs for once-impure purity.*

Chapter 6
Memoization

Memoization is an optimization that caches argument to return value mappings, jumping past actual method execution for repeated invocations with the same arguments. A method is memoizable only when it has a functional property: there is a unique result for any given input. The forms of purity we define all ensure that pure methods have this functional property. Even methods identified as weakly pure are thus candidates for *memoization*. In fact, as far as memoization is concerned, our once-impure definition fits perfectly: a method is always invoked at least once before being memoized. This has the further benefit that mandatory class loading and initialization during a first invocation does not spuriously cause methods to be rejected as impure. In our case, *memoization* is also treated as a practical, if heuristic, validation for our purity analysis. If *memoization* is applied to methods that are not really pure, the program tends to fail as memoized results skip side-effects or incorrect return values are used.

In the sections below, we describe our memoization design. We begin with a discussion of basic design concerns and practical goals of the system. Section 6.2 presents the main framework for our investigation of memoization, and Section 6.3 describes the actual implementation.

6.1 Developing Memoization in a JVM

At its core our basic approach to memoization follows the traditional idea of substituting method invocation by table-lookup. A typical memoization scheme maintains a *memo table* used for mapping input arguments to previously computed results. A new method invocation will *lookup* this table to find whether this method has been invoked before with the same arguments. If it has, the previous stored result from the table is returned and execution of the method skipped. Otherwise, this method must be invoked as normal; when it returns, the new mapping of invocation arguments and return value are added to the memo table. From an algorithm perspective, the structure of the *memo table* and the scheme for the *table lookup* are critical to the overall performance. Efficient memoization in a Java context implies further concerns and constraints. This needs a more obvious structure.

6.1.1 Argument Caching

Argument caching is simple in principle, but the presence of not only objects but garbage collection as well method memoization in Java particularly challenging. Saving the object arguments directly to the *memo table* is not practical. In Java object or reference variables could be destroyed or moved by automatic garbage collection, invalidating object arguments in *memo table*. Finding and purging the table appropriately is expensive, and may reduce memoization opportunities depending on the frequency of GCs. For example, in Figure 6.1, if garbage collection happens between the first invocation of `Test.foo(p1)` (line 9) and the second invocation of `Test.foo(p1)` (line 12), all objects in the *memo table* maybe invalid, and `Test.foo(p1)` (line 12) cannot be skipped anymore. Of course we can maintain a valid *memo table* even after garbage collection by modifying GC to update all objects in the *memo table* after garbage collection. But this is an invasive procedure given the weak guarantees on GC behaviour provided by Java. Our actual scheme for handling objects and arbitrary GC is to recursively *flatten* input objects, maintaining only type structure and primitive values. The detail of this *flatten algorithm* will be discussed in Section 6.3.1. Here, we will give an brief example to show how this algorithm works. For example, consider the method invocation in line 11 at Figure 6.1, *p2* is used as the argument for the first time, but actually memoization can be applied, because *p2* has exactly the same content and structure as *p1*, in the previous invocation at line 9. Our memoization for Java programs uses the content of object arguments as the mapping input and does not include deep addresses. This scheme will not be influenced by garbage collection and can even allow memoization for different, but equivalent in content, objects. Naturally, this scheme can consume significant time and space. The process for big objects maybe quite expensive, and we will need to balance the benefit and cost in practice.

```
   class Point
 2 { public int x,y; }
   public class Test{
 4   public int foo(Point p) //pure method with object argument
     { return p.x+p.y; }
 6   public void bar()
     {
 8     Point p1 = new Point(1,2);
       int sum = foo(p1); //invoked the first time.
10     Point p2 = new Point(1,2);
       sum = foo(p2); //invoked by the different object.
12     sum = foo(p1); //invoked by the same object.
     }
14 }
```

Figure 6.1: *A pure method with object arguments.*

6.1.2 Memoizable Methods

As suggested above, not all methods are worth memoizing even if they are sufficiently pure. The benefit obtained from jumping past method execution must exceed the cost of looking up the return value for memoization to have any positive effect. In our case we use a few heuristic rules. The *memoizable* methods are those methods that are pure and satisfy the following rules:

1) the method must execute for long enough to be worth skipping.

2) There must be a good hit rate. If a method is rarely invoked with the same arguments, the cost of input caching and mapping may exceed any benefit.

3) The amount of input data to be processed cannot be too large. Otherwise the cost of *argument tracking*, flattening input reference arguments could be too expensive.

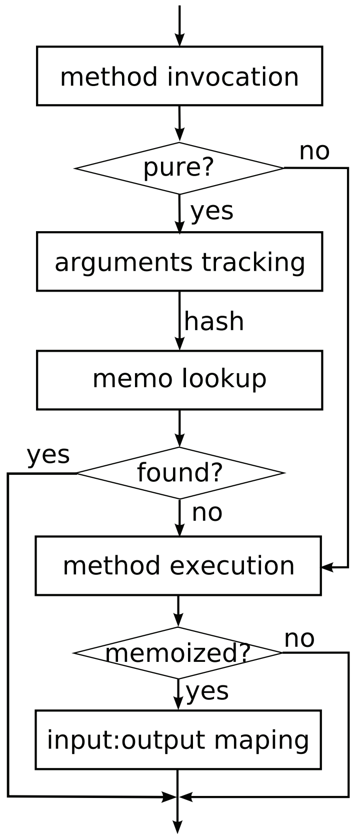
4) No cyclic references in the arguments. A looping reference inside arguments could make our *argument tracking* fall into an infinite process. The detail for why it will fail are explained in Section 6.3.1.

5) The return value is primitive type only. This restriction is aim at simplifying the implementation for our memoization.

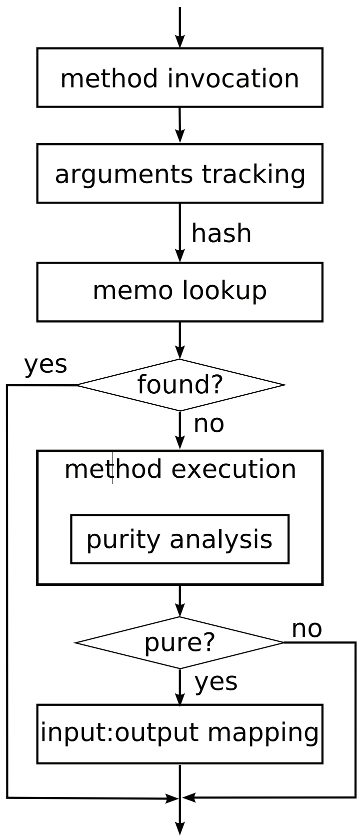
6.2 Overall Framework

Our overall framework including memoization has illustrated in Figure 5.2 (page 31). Memoization can be driven by either online or offline dynamic purity analysis, as shown on the right hand side of Figure 5.2. In *online analysis*, memoization uses the purity information from purity analysis immediately. For *offline analysis* purity information comes from a previous program run; that is, *offline analysis* requires at least two rounds of program running, one for purity analysis only, and another for memoization only. Purity information gathered in the first run is written out as a file. In subsequent runs this file will be loaded and used at the memoization stage.

Figure 6.2 shows the control flow for both *online* and *offline* memoization upon method execution. They have the same main components and similar control flow, but *online analysis* need of course the online component for supplying purity information. As showed in the left hand side of Figure 6.2, the method will be executed if the current arguments are not found or the method is impure. When a memoizable method is returned, the mapping between input argument and return value will be added to the *memo table* for possible reuse later. The right hand side of Figure 6.2 shows the control flow for *online analysis*. When



(a) offline analysis



(b) online analysis

Figure 6.2: *Control flows for memoization.*

the method is invoked, we may have no information about the method's purity, because this property depends on the result of a successful *memo lookup* or previous *purity analysis* on this method's current argument. If we find the method is impure after *purity analysis*, the argument tracking process could be a waste, and becomes part of the general overhead of the online system. Note, however, that *online analysis* has an advantage over offline in that purity can be considered on a per-input basis. *Memoization* can then be performed on all possible pure invocations, even in those methods which keep changing their purity states, and so will be treated as impure even based on *once impure purity*. For example, considered the code in Figure 6.3. `Test.foo(int)` is an impure method based on the *once* *impure purity*. Since it does not remain pure after the first invocation. But *memoization* can in fact happen on the second invocation of `Test.foo(2)`, since with an argument of 2 `Test.foo()` is pure.

```
public class Test{
  public int x;
  public int foo(int x)
  {
    if(this.x <= x)
    { this.x = x; }
  }
  public int bar()
  {
    this.x = 5;
    foo(10); //impure invocation;
    foo(2); //pure invocation; input state added to memo table.
    foo(10); //impure invocation;
    foo(2); //pure invocation; can be skipped.
  }
}
```

Figure 6.3: *Sample programs for per-input memoization.*

Argument tracking for all methods at all times is expensive, and for the online algorithm in practice, we need to keep a balance between this cost and its benefit. The *offline analysis* eliminates all the overhead of the purity analysis and allows for better evaluation of the memoization client in isolation.

6.3 Memoization Implementation

As described earlier, the memoization client consists of three main components: 1) argument tracking; 2) mapping between input and output; 3) argument lookup. Below we will describe these three processes and how they are implemented.

6.3.1 Arguments Tracking

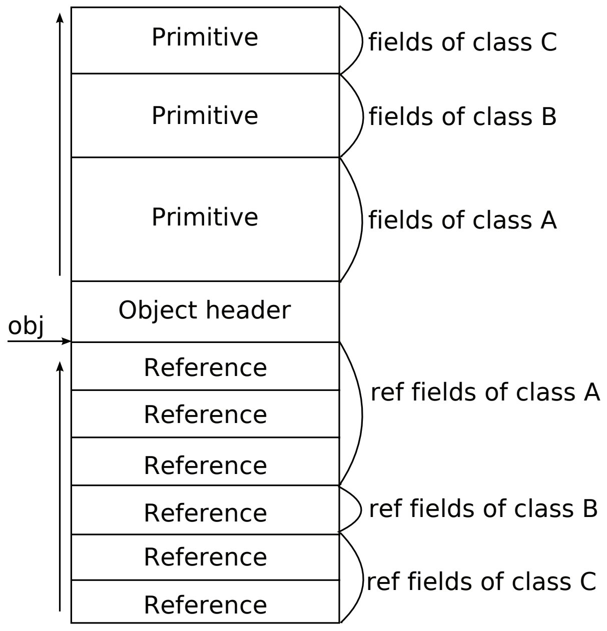


Figure 6.4: *Bidirectional object layout in SableVM.*

For the *once-impure* purity, the parameters of a pure method can be references or primitive variables. Generally, primitive arguments will be saved directly, and reference arguments will be “flattened” recursively until no references are left. Each reference variable is saved in the format of $< TYPE, (CONTENT) >$, where $TYPE$ is a Java class and $CONTENT$ is its recursively flattened data. No type information is stored for primitive variables because their type are guaranteed by parameter types at compile time already. Internally, SableVM [Gag02] uses a bidirectional object layout as show in Figure 6.4. *Obj* is an object instance of class C, which extends from B, and B extends from A. As shown in Figure 6.4, the fields in this layout are grouped as reference fields and non-reference fields. All primitive fields, whether from the class itself or its super class, are put consecutively after the object header. These primitive fields can be copied as a contiguous chunk of memory. Reference fields are put consecutively before the the object header, and so can be tracked recursively until no reference left in a manner similar to GC tracing. Consider the code in Figure 6.5. If we have an object *obj* with type A, it will be saved as: $obj =< A,(x,y,b =< B,(a,b) >) >$. Unfortunately, flattening in this manner means, circular data structures cannot presently be memoized. This kind of structure shown in Figure 6.6 would force our object tracking process into an infinite loop.

```
class A {
  int x, y;
  B b;
}
class B {
  int a, b;
}
```

Figure 6.5: *Sample code for object structure.*

In order to avoid infinite looping during arguments tracking, we develop an algorithm for circular reference detection. The basic idea of this algorithm is to check for the duplication of reference types during content “flattening”. Each memoizable method has a list called its *parameter reference type list*, which is used to store the reference types of a method’s parameters during flattening. Once any duplicated reference type is found, this method will be marked as a circular reference, cannot therefore be stored, and memoization from these arguments is abandoned. Otherwise, this set of parameters is memoizable without circular references. For example, suppose we have a method *foo(A a, B b)* and its structure of reference parameters *A* and *B* as shown in Figure 6.6. When flattened the first

```
class A {
  int x, y;
  B b;
}
class B {
  int a, b;
  A a;
}
```

Figure 6.6: *Sample code for circular reference.*

```
public class A {
  Object o1, o2;
  boolean foo() {
    return this.o1==null; //IFNULL
  }
  boolean bar() {
    return this.o1==this.o2; //ACMP
  }
}
```

Figure 6.7: *Sample code for ACMP.*

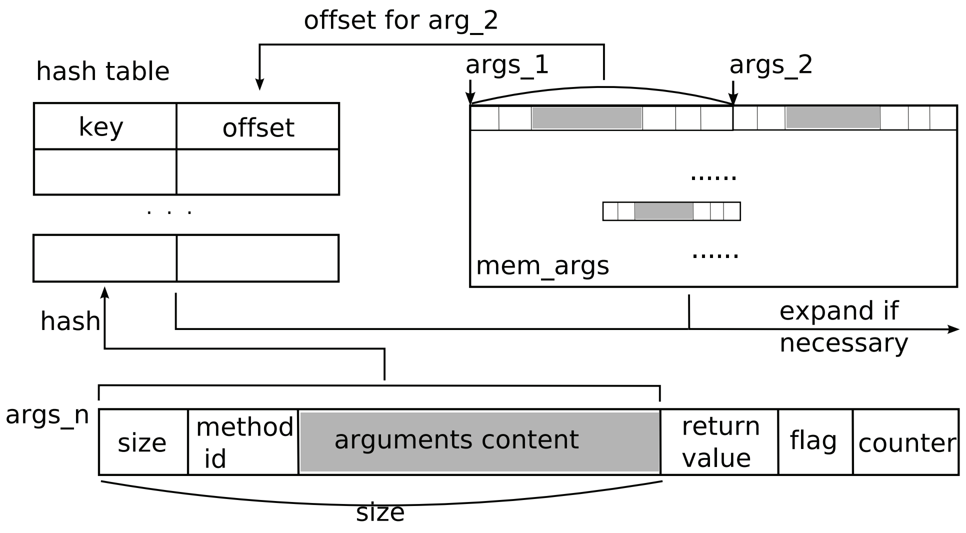


Figure 6.8: *Memory structure for arguments and hash table.*

parameter *a*, will get a *parameter reference type list* as *A B A*. That is, there exists duplicate reference types, and hence circularity.

Despite the inability to handle circular references, there are some advantages for storing only the object type and content. Mainly of course, garbage collection does not invalidate memoization tables. Secondly, however, a deep copy of an object will suffice when a different object actually has the same content. The output can be reused even the input object is stored in a different memory location, but the content of the input objects is the same. This will increase the possible re-usage, but does raise the issue of when can we safely consider objects identical. Certainly if addresses are not explicitly compared, then deep copies are equivalent for memoization. Code such as in Figure 6.7, however, compares object addresses using the `ACMP` bytecode. This is not safe to memoize. Note that we can still represent `null` object references, leaving the `IF(NON)NULL` bytecode safe for memoization. In Figure 6.7, `A.bar()` cannot be memoized, but `A.foo()` is safe for memoization.

6.3.2 Mapping Construction

A typical memoization scheme maintains a memo table mapping argument input to output. Our memoization design a hash structure for the memo table, shown on the left hand side of Figure 6.8 as *hash table*. Instead of saving all inputs directly into the table, we save the hash code as the key. The input data, due to its size is stored in a separate structure, the *mem_args* data structure shown on the right hand side of Figure 6.8. The offset of the content for each argument in *mem_args* is put in the *hash table* as the value. The detailed structure of each actual mapping in *mem_args* is shown at the bottom of the Figure 6.8. The first field is the size of this entry and the second field is a unique id for the method. This ensures different methods with identical inputs are not confused. The third field is the flattened content of the argument inputs. This content is followed by the output of the method, the return value to which the arguments map. *Flag* will be set when the *return value* is filled. The *counter*, is used to count the times this method is invoked with these particular arguments. Note that both the *hash table* and the *mem_args* can be expanded if necessary. By using offsets rather than addresses the hash table can easily accommodate a reallocated memory table.

6.3.3 Argument Lookup

Since our memo table is based on a hashing approach, the first step for argument lookup will be getting the hash code for the current method invocation. The hash code is computed from the content of argument tracking, content size, and method id, as shown at the bottom of Figure 6.8. If a method has been invoked previously with the same parameters we can find its offset in *mem_args*, where we save the content and return value for all previous invocations; this lookup itself is fast due to hashing, and has complexity *O(1)*. After locating a matching previous invocation content and return value by hashcode, however, we still need to make sure that the actual content before and now encountered are the same. Therefore, we make a simple comparison between the current and previous invocation content; that is, two invocations are considered invoked by the same parameter if the flattened content of these two invocations are the same. If they are exactly the same, we skip the current invocation by returning the stored value from previous invocation. Otherwise memoization fails, and this method will be executed as usual.

In the next chapter we give experimental results from our memoization system and purity analyses. This includes offline and online data, as well as a detailed investigation of the source of purity/impurity and how and when it is useful.

Chapter 7
Experiments

Experimental evaluation was conducted using the standard SPEC JVM98 benchmark suite at input size 100 [Sta98] on a 2 GHz Athlon `x86_64` machine running Linux. Our memoization system does not yet support multi-threading, and so we substitute the single-threaded *raytrace* benchmark for *mtrt*. We evaluate each form of purity described in the previous section using offline analysis, save for performance evaluation of the online analysis module. All averages are computed as geometric means.

The following section describes constraints for our experimental results. In Section 7.2, we introduce the metrics used to evaluate our purity evaluation. The experimental results for static analysis are given in Section 7.3, and in Section 7.4, we evaluate our different dynamic purity forms. Reasons why methods are considered Impure are examined in Section 7.5. Finally, based on our analysis results, we show the memoization experiments in Section 7.6.

7.1 Constraints

A few constraints are further imposed by our basic memoization implementations; several additional constraints that are unsafe for memoization are imposed in our dynamic purity analysis experiments. First of all, `ACMP_*` instructions are treated as unsafe. Our memoization is based on *once-impure* dynamic purity, in which pure methods are allowed to have object parameters. The content of all arguments is cached without considering the address of objects. The result of `ACMP_*` depends on a comparison of actual object addresses, and thus is not possible in our memoization. Figure 7.1 provides an example of the potential problem. If `ACMP_*` are considered safe, `Test.bar()` would be pure. Therefore, this method is memoizable. When `Test.bar()` is invoked for the first time at line 13, the mapping between arguments and result will be stored. The input content will be $< D,(5);D,(5) >$, and the return value in memo table will be *false*. Then, when it is invoked for the second time at line 15, however, after flattening the content of arguments, we get $< D,(5);D,(5) >$, a match for our entry in the *memo table*. Therefore, the method is skipped by replacing the return value with *false*. But this is wrong, because these two objects now actually point to the same object, and the return value should be *true*. In practice, we find that the constraint on `ACMP_*` has little bearing on overall results, with a notable exception being *jess* as we discuss in Section 7.5.

Another important consideration is that our memoization system does not yet support multi-threading; we assume a single thread execution. In this situation, `MONITOR` operations can actually be treated as safe instructions. In order to examine the upper bound of pure methods, we ignore synchronization operations that may cause method to be impure based on our purity criteria.

7.2 Metrics

In order to evaluate and compare the result from different levels of purity analysis. We introduce some purity metrics. *Static method purity* is calculated as the percentage of all methods in the call graph that are pure, as reported by prior work on purity and side effect analysis [Rou04, SR05]. Dynamic purity is evaluated by three *dynamic purity metrics*.

- *Method purity*
  This is calculated as the percentage of all reached methods at runtime that are pure. The formula can be described as:

  $pure\_methods/total\_methods * 100\%$.

- *Invocation purity*

```
class D {
  int x;
  D(int x) { this.x=x; }
}
public class Test {
  boolean bar(D o1, D o2){
    return o1 == o2; // ACMPEQ
  }
  void foo() {
    boolean flag;
    D o1 = new D(5);
    D o2 = new D(5);
    flag = bar(o1, o2); //false
    o2 = o1;
    flag = bar(o1, o2); //true
  }
}
```

Figure 7.1: *An example for ACMP_*.*

This is the percentage of all runtime method invocations that are pure. The formula can be described as:

$pure\_method\_invocations/total\_method\_invocations * 100\%$.

- *Bytecode purity*
  This is the percentage of the executed bytecode instruction stream that is contained within pure methods. It can be described in the formula:

  $pure\_method\_bytecodes/total\_bytecodes * 100\%$.

For *bytecode purity*, there are two complications involved in calculating dynamic bytecode purity. First, only those instructions executed *locally* in a given method are counted towards the total number of impure or pure bytecodes. Second, for an impure method executed within a pure context such that under moderate or weak purity the execution is actually pure, the instructions are counted towards the total number of pure bytecodes. This requires propagating purity information on method invocation.

```
foo(){ //impure
  ...... //4 bytecodes
  putstatic; //impure bytecode.
  bar();
  ...... //4 bytecodes;
}

bar(){ //pure
  ...... //10 bytecodes.
}
```

```
foo(){ //pure
  ...... //4 bytecodes
  o = new Object(); //new
  bar(o);
  ...... //4 bytecodes
}
bar(Object o){ //impure
  o.f = <value>; //putstatic
  ...... //9 bytecodes
}
```

Figure 7.2: *Examples for bytecode counting.*

Two examples for bytecode counting are given in Figure 7.2. In the example on the left hand side the pure method, `bar()` is called inside an impure method `foo()`. The bytecode size of `foo()` is 20, including all bytecodes in `bar()`. The size for `bar()` is 10. The method size of a method is calculated by counting the really executed bytecodes after the method is invoked and before the method returns, and this will include all bytecodes in its callees. To avoid multiple counting (e.g., in this example, the total bytecodes of `foo()` and `bar()` are 30, but the total bytecodes really executed are 20), the bytecode purity is calculated through *local bytecodes*. In this example, the local bytecodes for `foo()` is 10, and `bar()` is 10 too. Therefore, the percentage of bytecodes in pure methods is $10/(10+10)*100\% = 50\%$. The right hand side of Figure 7.2 is an example of an impure method `bar()` invoked in the pure context of `foo()`. The local bytecodes for both `foo()` and `bar()` are 10. And `bar()` is identified as impure. In a *moderately* or *weakly* pure context `foo()` is pure. In this case, the bytecode purity for the program is **100%** — the total bytecodes executed are 20, and all of them are in the pure context, even the impure method.

It is important to determine whether dynamic purity is present in a non-trivial way, and in this respect we consider bytecode purity a better indicator than invocation purity, and invocation purity a better indicator than method purity. A large percentage of pure bytecodes, most directly shows the possible benefit from the pure methods. The invocation purity shows the invocation frequency from pure methods, a more easily gathered but less accurate measure, and the method purity gives the information of the number of pure methods, a measure suitable for static comparisons.

7.3 Static Purity Analysis

Our static analysis includes all methods in both class library and application code that are found in the call graph created by our conservatively-correct CHA-based whole program analysis. A more precise analysis would analyze fewer methods in exchange for computation time [LH06], but we did not investigate this in this thesis.

Table 7.1 shows the percentage of all methods in the call graph identified as statically pure at compile time. About 13% of methods are found to be strongly pure under all possible execution scenarios, with most pure methods coming from the standard Java libraries. On average, more that 40% of pure methods are `<init>` methods. The data from Table 7.2 is collected at runtime based on the purity information identified through static analysis. The first row shows the percentage of all methods reached at runtime that are statically pure; the *invocations* row shows the percentage of all dynamic method invocations that execute some statically pure method; the *bytecode* row shows the percentage of the bytecode instruction stream that is executed by some statically pure methods. Clearly not all methods from static analysis will be invoked at runtime. All possible target methods will be included in the call graph at the static time, but the methods invoked at runtime are determined by the actual received object type at runtime. This greatly reduce the total number; at runtime, we only find 5–6% of reached methods are statically identified as pure, with even less invocations and bytecodes.

Tables 7.3, 7.4, 7.5 list the distributions for *strongly* pure methods on method type

	comp	db	jack	javac	jess	mpeg	rt
pure methods	14%	13%	13%	12%	13%	13%	13%
`<init>`	41%	41%	42%	38%	45%	41%	41%
app	2%	2%	5%	2%	11%	5%	2%

Table 7.1: *Strong static method purity.*

metric	comp	db	jack	javac	jess	mpeg	rt
methods	6%	6%	6%	5%	5%	6%	5%
invocation	≈0%	12%	10%	10%	6%	16%	3%
bytecode	≈0%	2%	1%	≈0%	≈0%	2%	.5%

Table 7.2: *Strong dynamic purity identified at static.*

and method size in method purity, invocation purity, and bytecode purity. All data in these three tables are collected at runtime in the modified SableVM, and all pure methods are identified as pure during static analysis in Soot. Table 7.3 shows the distribution of method purity; more than 60% pure method invoked are from `<init>`. The invoked non-library pure methods are varied for different benchmarks. It could be as much as 50%, such as *javac*, and *jess*, or only a small percentage for *comp*, *db*, and *raytrace*. Most pure methods are also small, as the second section in Table 7.3 shows. The invocation purity distribution is showed in Table 7.4. The `<init>` or non-library pure method invocations, the percentage could be extremely different for different benchmarks. More than 50% method invocation are to `<init>` in *comp*, *jack*, *jess*, and *raytrace*, but only a small percentage of invocations are to `<init>` in *mpegaudio*. In this case it is because the method `spec/benchmarks/_222_mpegaudio/q.j(F)S` is invoked particularly frequently in *mpegaudio*. The second section of Table 7.4 shows that most invoked pure methods are those methods of small size, less than 15 bytecodes. Finally, Table 7.5 shows the distribution of bytecodes executed by pure methods. The bytecode percentage for calls to `<init>` is not dominant, as they were with method purity and invocation purity, and again vary for different benchmarks. The executed bytecodes that come from pure application methods could be extremely different for different benchmarks; nearly 0% in *comp* and up to 99% in *mpegaudio*.

Static strong purity is a strong and conservative purity criteria. Only a small percentage of methods are *strongly pure*, and most these methods are `<init>`. Almost all *strongly pure* methods are small size method with less than 10 instructions executed in the method. This is not difficult to understand. Java is an object-oriented language; most methods will read from or write to heap. The strong purity criteria that no heap related operation occur at all will limit pure methods to only very simple operations.

	comp	db	jack	javac	jess	mpeg	rt
`<init>`	61%	60%	63%	35%	76%	61%	58%
app	2.4%	0%	29%	45%	56%	22%	2.3%
`[ 1, 4)`	56%	53%	45%	68%	74%	57%	53%
`[ 4, 8)`	37%	38%	50%	27%	19%	35%	35%
`[ 8,16)`	7%	10%	5.4%	5%	6%	8%	9%
`[16,32)`	0%	0%	0%	0%	0%	0%	2%
`[32, )`	0%	0%	0%	0%	1%	2%	0%

Table 7.3: *Method purity distribution for strongly pure method identified at static*. The top row shows the proportion of `<init>` methods included in the results. The next row is the amount of application code. The bottom 5 rows divide pure methods according to size in bytecodes.

7.4 Dynamic Purity Analysis

In this section, we evaluate the different levels of dynamic purity using our dynamic metrics, *method purity*, *invocation purity* and *bytecode purity*. In order to allow a good comparison between different levels of purity, the experimental results for different purity forms are aggregated in the result tables, Table 7.6, Table 7.7, and Table 7.8. We start by showing the results for *strong purity*, followed by *moderate purity*, which removes the read and write limitation on local objects, we then consider our weaker forms still, and *once-impure purity*, which removes the purity requirement for the first invocation.

	comp	db	jack	javac	jess	mpeg	rt
`<init>`	59%	12%	53%	44%	88%	≈0%	71%
app	≈0%	0%	3%	26%	≈0%	99.5%	29%
`[ 1, 4)`	61%	12%	66%	64%	69%	.15%	99%
`[ 4, 8)`	39 %	88%	34%	36%	31%	.44%	.2%
`[ 8,16)`	.1%	≈0%	≈0%	≈0%	≈0%	99.4%	≈0%
`[16,32)`	0%	0 %	0%	0%	0%	0%	.6%
`[32, )`	0%	≈0%	0%	0%	≈0%	≈0%	0%

Table 7.4: *Invocation purity distribution for strongly pure methods identified at static.*

	comp	db	jack	javac	jess	mpeg	rt
`<init>`	19.5%	1.9%	19.8%	15.4%	59%	≈0%	50%
app	≈0%	0%	2.2%	14.8%	.1%	99.8%	40.6%
`[1,4)`	19%	1.8%	24%	24.5%	33%	≈0%	90.3%
`[4,8)`	81%	98%	76%	75.5%	67%	.2%	≈0%
`[8,16)`	.2%	≈0%	≈0%	≈0%	≈0%	99.8%	≈0%
`[16,32)`	0%	0%	0%	0%	0%	0%	9%
`[32, )`	0%	0%	0%	0%	0%	0%	0%

Table 7.5: *Bytecode purity distribution for strongly pure methods identified at static.*

purity	comp	db	jack	javac	jess	mpep	rt
strong	7%	7%	6%	6%	9%	8%	6%
moderate	10%	9%	8%	8%	9%	8%	6%
weak	18%	18%	15%	19%	23%	18%	22%
once-impure	19%	19%	16%	21%	24%	19%	23%

Table 7.6: *Dynamic method purity.* Percentage of all reached methods reached that are pure for different dynamic purity definitions.

purity	comp	db	jack	javac	jess	mpeg	rt
strong	$\approx 0\%$	15%	13%	11%	10%	16%	8%
moderate	$\approx 0\%$	15%	19%	17%	17%	16%	8%
weak	33%	87%	35%	27%	43%	31%	90%
once-impure	33%	87%	39%	29%	46%	31%	91%

Table 7.7: *Dynamic invocation purity.* Percentage of all method invocations that are pure for different dynamic purity definitions.

purity	comp	db	jack	javac	jess	mpeg	rt
strong	$\approx$0%	3%	1%	1%	1%	2%	1%
moderate	$\approx$0%	3%	1%	1%	1%	2%	1%
weak	5%	62%	17%	24%	13%	3%	53%
once-impure	6%	62%	20%	26%	16%	3%	56%

Table 7.8: *Dynamic bytecode purity.* Percentage of total bytecode instruction stream that is contained in a pure method for different dynamic purity definitions.

7.4.1 Strong Dynamic Purity

Strong dynamic purity is a weaker form of purity than its static equivalent, and the results in the first row of Tables 7.6, 7.7, and 7.8 improve on the runtime use of strong static purity in Table 7.2. In Table 7.6, up to 4% more pure methods are reached using strong dynamic purity. Around 1-4% more invocations are also dynamically strongly pure. Nevertheless, the overall impact remains small, with Table 7.8 showing no more than 3% of all bytecode instructions being executed in a pure context, and a maximum gain over strong static purity of just 1%.

During dynamic analysis, we only examine the executed bytecodes, rather than examining the whole method body as in static analysis. Thus there should be some methods strongly pure at runtime identified as impure statically. For example, the method `java.lang.String.valueOf(Object obj)` from *db* is identified as a dynamic strongly pure, but static strongly impure method. The program for this method is shown in Figure 7.3. There are two control paths in method `java.lang.String.valueOf-(Object obj)`: one path returns a constant string, another path will call an impure method `java.lang.Object.toString()`, which is impure because of its impure callee `java.lang.String.hashCode()`. However, this impure path never be invoked at runtime, when consider benchmark *db*. Therefore, `java.lang.String.-valueOf(Object obj)` is identified as a strongly pure method with dynamic analysis.

```
public static String valueOf(Object obj){
  return obj == null ? "null" : obj.toString();
}
public String toString(){
  //hashCode is an impure method, which includes write to the heap.
  return getClass().getName + '@' + Integer.toHexString(hashCode());
}
```

Figure 7.3: *Dynamic strongly pure method identified as strongly impure at static.*

Only a small improvement is gained through dynamic analysis compared with static analysis based on the strong purity form. Dynamic purity analysis will improve those methods with both impure and pure control flows, but which have only pure control flows reachable at runtime. From the experimental results, we know that this case does not happens very frequently in practice. However, in dynamic analysis, the method purity can also be described based on per-input. The method purity can keep changing as the program is run under different inputs. But if it is pure with some particular arguments, it will always be pure with these arguments.

7.4.2 Moderate Dynamic Purity

It is not difficult to understand that we not get many *strongly* pure methods, because the strong purity form is too conservative. Java is an object-oriented language, disallowing all operations related to the heap represents a very big limitation. We consider relaxing the purity definition by allowing some forms of heap operation. In moderate dynamic purity heap read and write are allowed at least on local objects. This allows some heap interaction, while ensuring heap access is easily bounded. Before collecting our experiments, we expected a big gain to be achieved compared with *strong purity*: heap reading and writing are the main reason for *strong* impurity. Allowing access to local heap accesses may relax many *strongly* impure methods, and allow then to become *moderately* pure. This concept of locality in heap interaction can be extended through methods calls, and thus potentially scope methods. For example, in Figure 7.4, although both `bar()` and `baz()` are *moderately impure*, caller `foo()` actually is pure, because *obj* is created locally and does not escape from `foo()` despite its callees. Therefore, we expect to find more pure methods with *moderate purity*.

```
void foo() { //pure
  Object obj = bar();
  baz(obj);
}
Object bar() { //impure
  Object obj = new Object();
  return obj; //escaped to caller
}
void baz(Object obj) { //impure
  x = obj.f; //read from external object.
}
```

Figure 7.4: *Examples for moderate purity.*

Unfortunately, we observe marginal improvements to all runtime measurements, but overall do not find any large gains compared with *dynamic strong purity*. Recall that under moderate dynamic purity, methods are not allowed to read or write heap data from objects pre-existing the method call, preventing actual use of object parameters; this constraint ensures a simple bounding of the input state. Table 7.9 presents dynamic metrics for all methods that accept or return references. All benchmarks have at least 53% of reached methods executed in this context that is likely to be impure, and with the exception of *compress*, at least 57% of bytecode execution as well. However, even though compress exhibits a maximum of 4% of bytecode execution being impure due to reference parameters and return values, in fact *compress* is highly impure for other reasons, namely large amounts of execution within large and hot methods that contain `PUTFIELD` bytecodes.

metric	comp	db	jack	javac	jess	mpeg	rt
methods	62%	62%	53%	68%	54%	60%	62%
invocations	1%	51%	49%	46%	76%	33%	37%
bytecode	4%	60%	63%	57%	93%	92%	71%

Table 7.9: *All methods with reference parameters or return reference.*

7.4.3 Weak Dynamic Purity

Weak dynamic purity eliminates the restriction on moderate dynamic purity that a method not inspect the reachable heap. This allows significantly more purity to be identified, roughly doubling the number of pure methods, and resulting in even larger gains with respect to dynamic invocation purity and dynamic bytecode purity, as shown in Tables 7.7 and 7.8. In particular, *db* and *raytrace* execute a high percentage of the bytecode instruction stream within a pure context.

Data similar to that shown in Tables 7.3, 7.4, 7.5 is presented for weak dynamic purity in Table 7.10, 7.11, 7.12. Tables 7.10 shows the percentage of pure methods that are from application and `<init>`, and the distribution on method size. From Tables 7.10, we can find that only about 20% pure methods are `<init>`. This is much less than the *method purity* for *moderate purity*, which is more than 60%. Relative amount of pure methods from pure applications still can be very different for different benchmarks; this property is the same as *moderate purity*. However, in *weak purity*, we do find some pure methods with large method sizes. On average, about 5% of pure methods have more than 256 bytecodes. Disappointingly, however, these big pure methods are not invoked frequently, as can be seen in Table 7.11. More than 90% of *invocation purity* is from quite small pure methods, less than 16 bytecodes. The library and application split and use of `<init>` continues to vary. Almost all pure invocations come from the library for *db*, but for *comp*, *mpeg*, and *raytrace* almost all pure invocations come from the application. The invocation of pure `<init>` can be close to 0%, such as *comp* and *mpeg*, and ranges up to 17%. When considering bytecode purity, the percentage of bytecodes from `<init>` is small, which is shown in Table 7.12. For all benchmarks in *spec*, no more than 6% of bytecodes executed are from pure `<init>` methods. As we mentioned before, we do find some pure methods with big method size. Therefore, in *bytecode purity*, we are glad to find that in some benchmarks, we have more than 50% of bytecodes come from pure methods with big size, for example, about 65% of bytecode in *db* has more than 65 bytecode. And about 44% of bytecodes in *javac* have more than 256 bytecodes.

pure	comp	db	jack	javac	jess	mpeg	rt
`<init>`	20%	19%	25%	10%	28%	21%	13%
app	5%	3%	17%	50%	50%	23%	33%
`[ 1, 16)`	78%	78%	80%	81%	89%	82%	82%
`[ 16, 32)`	5%	5%	6%	6%	2%	5%	4%
`[ 32, 64)`	7%	7%	6%	6%	3%	6%	7%
`[ 64,128)`	1%	2%	.7%	1%	2%	.7%	3%
`[128,256)`	3%	4%	2%	1%	.4%	0%	2%
`[256, -)`	6%	5%	6%	4%	3%	6%	3%

Table 7.10: *Distribution on method purity for weak purity.*

Although we can find some big pure methods through *weak purity*. as we will show in our memoization experiments, many of these methods are not necessarily suitable for profitable exploitation, and for our benchmarks weak purity does not result in such large practical increases. We hypothesized that this might be due to initialization requirements, leading to aggressive rejection of methods as impure based on special operations performed only on the first invocation. We now consider the impact of initialization through once-impure data.

pure	comp	db	jack	javac	jess	mpeg	rt
`<init>`	≈0%	2%	16%	17%	12%	≈0%	2%
app	≈100%	≈0%	4%	52%	77%	≈100%	98%
`[ 1, 16)`	≈100%	85%	98%	87%	≈100%	≈100%	96%
`[ 16, 32)`	≈0%	≈0%	1%	2%	≈0%	.1%	≈0%
`[ 32, 64)`	≈0%	≈0%	≈0%	10%	≈0%	≈0%	2%
`[ 64,128)`	≈0%	15%	≈0%	≈0%	≈0%	≈0%	≈0%
`[128,256)`	≈0%	≈0%	.1%	≈0%	≈0%	0%	2%
`[256, -)`	≈0%	≈0%	≈0%	.5%	≈0%	≈0%	≈0%

Table 7.11: *Distribution on invocation purity for weak purity.*

pure	comp	db	jack	javac	jess	mpeg	rt
`<init>`	≈0%	.1%	3%	6%	4%	≈0%	.5%
app	≈100%	≈0%	6%	23%	78%	99%	99%
`[ 1, 16)`	≈100%	35%	77%	23%	98%	99%	70%
`[ 16, 32)`	≈0%	≈0%	4%	6%	≈0%	.1%	.1%
`[ 32, 64)`	≈0%	≈0%	.1%	16%	.4%	.1%	8%
`[ 64,128)`	≈0%	65%	≈0%	10%	≈0%	≈0%	.2%
`[128,256)`	≈0%	≈0%	18%	2%	≈0%	≈0%	21%
`[256, -)`	.1%	≈0%	.5%	44%	1%	.4%	1%

Table 7.12: *Distribution on bytecode purity for weak purity.*

7.4.4 Once-impure Dynamic Purity

Once-impure results are shown in the last row of Tables 7.6, 7.7, and 7.8. We observe small gains in dynamic method purity for all benchmarks, only about 1-2% improvement compared with *weak purity*, and slightly larger gains for some benchmarks in the context of dynamic invocation and bytecode purity, about 3–4% more invocation purity for *jack* and *jess*, and around 3% more bytecode purity for *jack*, *jess*, and *raytrace*. This means for some benchmarks, the new pure methods found by *once-impure purity* could be some methods with big size.

Of course, once-impure dynamic purity can be generalized: it is possible that the purity of a given method is dynamically manifest only after $n > 1$ impure executions, that a pure method actually becomes impure after some number of executions, or that more complex pure$\longleftrightarrow$impure transitions occur. Detailed information on more general forms of *once-impure* will be described in next section.

7.4.5 Remaining Dynamic Purity

Table 7.13 provides a detailed breakdown according to the number and kinds of methods captured and ignored by our purity analysis discussed from Section 7.4.1 to Section 7.4.4. The first section in Table 7.13 shows reached method counts accounted for by our analysis: those that are either always pure, always impure, or once-impure. The second section shows reached methods that are not accounted for: those that are impure twice or more before becoming always pure, those that are pure once or more before becoming always impure, and those that change state more than once in *remainder*. The final section provides dynamic purity metrics, method, invocation, bytecode for the methods identified in the data above.

For all benchmarks, less than 10% of methods change their purity status over the course of execution, with the vast majority being always pure or always impure. Once-impure does indeed capture the bulk of methods that change state from impure to pure, with no more than ≈2% of ultimately pure methods remaining impure for more than one execution. Interestingly, there are no methods that are initially always pure that later permanently change to being always impure, as seen in the `P+I+` row. There are however fairly large numbers of methods that change state more than once, as seen in the *remainder* row. We analyzed the extent of missed opportunities in the second section using our dynamic metrics, and found a surprising amount unaccounted for execution, particularly for *jack*, *javac*, and *jess*.

7.4.6 Summary for Dynamic Purity

In Section 7.4, we report the experimental results for four different levels of purity by our dynamic metrics. The changes between *strong purity* and *moderate purity* are small, the

state regexp	comp	db	jack	javac	jess	mpeg	rt
`P+`	130	135	152	299	281	158	198
`I+`	559	602	795	1120	873	706	680
`IP+`	10	10	11	33	11	12	12
`II+P+`	0	0	0	3	6	1	1
`P+I+`	0	0	0	0	0	0	0
remainder	22	23	36	109	42	24	22
method	3%	3%	4%	7%	4%	3%	3%
invocation	21%	3%	16%	17%	13%	6%	1%
bytecode	12%	3%	25%	26%	30%	5%	6%

Table 7.13: *Analyzed and unanalyzed methods.*

difference between *weak purity* and *once-impure* is also small. However, compared with *moderate purity*, we achieve big gains from *weak purity*, which only allows local heap reads.

We also observe a general trend in our dynamic metrics. First of all, dynamic method purity is fairly similar between benchmarks. There is no large difference between static *strong purity*, dynamic *strong purity*, and dynamic *moderate purity* for *method purity*. Dynamic *weak purity*, however, doubles the method purity compared with dynamic *moderate purity*. Unfortunately, this pattern does not continue: only small gains are achieved from *once-impure purity.* Secondly, differences in dynamic *invocation purity* are greater, separating programs into two groups. For some benchmarks, such as *db* and *rt*, the *invocation purity* can go up to 90%; for other benchmarks, however, there are only about 30–40% invocations that come from pure methods. The big gain between different purity forms still happens between *moderate purity* and *weak purity*. Thirdly, differences in dynamic *bytecode purity* are greater still, separating programs into three distinct groups. The highest percentage groups still come from *db* and *rt*, which have around 50–60% executed bytecodes based in pure methods, Another group is around 20%, and includes *jack*, *javac*, and *jess*. The group with the smallest improvement includes *comp* and *mpeg*, which have only about 5% of executed bytecodes in pure methods. Once again the difference between *mod-*

erate purity and *weak purity* tends to represent the largest gain: for all benchmarks, the percentage of bytecode for *moderate purity* is less than 5%. Generally, both *invocation* and *bytecode purity* is unpredictable from *method purity*, and may vary greatly between benchmarks. This tendency of our metrics to polarize benchmarks is a useful property. We consider *bytecode purity* a better indicator for purity evaluation than *invocation purity*, and *invocation purity* a better indicator than *method purity*, and intend to explore the relation between purity styles/results and benchmark behaviour as future work.

7.5 Impure Reasons Analysis

Methods themselves may be impure for multiple reasons or only for a single reason. Tables 7.14, 7.15, and 7.16 give details as to which bytecodes actually cause impurity under once-impure dynamic purity.

Table 7.14 shows the reasons for dynamic method impurity. In the top section each row shows methods rejected solely for encountering that bytecode in our once-impure analysis, or a native `INVOKE*` in the case of *native*. Methods rejected for encountering `PUTFIELD` that also encountered another impure bytecode are shown in `PUTFIELD+`. Individually negligible sources of impurity not accounted for by the other rows are summed in *others*. In this table, between 20% and 30% of reached methods are impure entirely due to the use of `PUTFIELD` on escaping objects, and well over 50% of methods are marked impure after encountering other disallowed bytecodes in addition to `PUTFIELD`. In the case of dynamic invocation impurity, shown in Table 7.15, this balance tips more in the other direction: *compress*, *db*, and *raytrace* find `PUTFIELD` alone a much more significant contributor than multiple impurity reasons. *jess* is marked by the dominance of `ACMP_*` bytecodes in impurity decisions; these bytecodes are used extensively for the implementation of `equals()` methods in the different application classes of *jess*.

Bytecode execution data in Table 7.16 show the importance of considering other method execution properties in evaluating purity. Although `ACMP_*` is a dominant factor for *jess*, in practice these bytecodes are contained in small methods, and the executed bytecode contribution to impurity is somewhat reduced when compared with dynamic invocation impurity. `PUTFIELD` as a lone contributor is also less important in terms of bytecode execution; only *db* continues to show PUTFIELD as a significant single source of impurity, although PUTFIELD does maintain a large presence when there are multiple impurity reasons. Clearly, further weakening of purity to allow more pure PUTFIELD operations will be of value, however measured. Nevertheless, the largest potential source of further, weaker purity apparently lies in analyzing and handling methods marked impure due to execution of multiple kinds of impure bytecodes.

impurity	comp	db	jack	javac	jess	mpeg	rt
ATHROW	0%	0%	0%	0%	0%	0%	0%
ACMP_⋆	1%	1%	1%	1%	1%	1%	1%
PUTFIELD	27%	29%	21%	21%	23%	24%	28%
⋆STATIC	6%	6%	4%	3%	4%	5%	5%
ARETURN	1%	1%	1%	1%	1%	1%	1%
ASTORE	0%	0%	0%	0%	0%	0%	0%
native	8%	8%	6%	5%	6%	7%	9%
PUTFIELD+	52%	52%	58%	66%	61%	60%	53%
others	5%	3%	9%	3%	4%	2%	3%

Table 7.14: *Reasons for dynamic method impurity*

7.6 Memoization

Our evaluation of memoization depends on a once-impure dynamic purity analysis. For efficiency, memoization is only applied to methods for which it cost effective to do so. We investigated different limits on method size, and for each method used an input size limit of 100 KB, a warm up period of 1000 cold start misses, after that a minimum hit ratio of 10%, and a global size limit on memoization data of 1 GB. As discussed in Section 6, we cannot compare object addresses directly using ACMP_⋆, and so the usable purity information is a strict subset of the once-impure purity we can actually identify.

The impact of these constraints on memoization is significant. Table 7.17 shows low absolute numbers of methods memoized under our present cost constraints, and Table 7.18

impurity	comp	db	jack	javac	jess	mpeg	rt
ATHROW	0%	0%	0%	0%	0%	0%	0%
ACMP_*	≈0%	≈0%	12%	6%	54%	≈0%	≈0%
PUTFIELD	81%	82%	45%	25%	24%	40%	71%
*STATIC	≈0%	≈0%	2%	≈0%	3%	≈0%	≈0%
ARETURN	0%	0%	0%	0%	0%	0%	0%
ASTORE	0%	0%	0%	0%	0%	0%	0%
native	≈0%	1%	3%	10%	≈0%	≈0%	1%
PUTFIELD+	19%	17%	37%	58%	19%	60%	28%
others	0%	≈0%	1%	1%	0%	≈0%	≈0%

Table 7.15: *Reasons for dynamic invocation impurity.*

impurity	comp	db	jack	javac	jess	mpeg	rt
ACMP_*	≈0%	2%	11%	7%	46%	≈0%	≈0%
PUTFIELD	21%	85%	38%	25%	8%	11%	33%
*STATIC	0%	0%	0%	0%	1%	0%	0%
native	0%	0%	0%	1%	0%	0%	0%
ARETURN	0%	0%	1%	0%	0%	0%	0%
PUTFIELD+	79%	13%	48%	66%	45%	89%	66%
others	≈0%	≈0%	3%	2%	1%	≈0%	1%

Table 7.16: *Reasons for dynamic bytecode impurity.*

shows the percentage of normal execution that is successfully skipped. Even with *db* and *raytrace* containing a large amount of pure execution, memoization cannot be effectively applied. In the case of *db*, this is due to the fact that pure methods are simply not executed frequently enough with the same arguments. In fact, only *jack* and *javac* exhibit non-negligible memoizability, despite that they also exhibit fairly low amounts of pure bytecode execution in Table 7.8. For all benchmarks, the success of memoization is inversely related to minimum method size: although large memoizable methods provide significant benefits, they are much less common than smaller methods. Ensuring that a memoization system has low overhead is thus critical if numerous smaller methods are to be efficiently memoized.

size	comp	db	jack	javac	jess	mpeg	rt
10+	21/42	21/45	20/50	50/107	25/54	35/61	19/62
20+	15/42	15/45	18/49	33/106	18/54	27/60	24/62
30+	14/42	14/45	17/49	32/108	17/54	25/59	23/62
40+	14/42	13/45	17/49	30/107	17/54	25/59	23/62
50+	13/42	12/45	16/49	30/106	16/54	24/59	22/62
100+	12/42	12/45	15/49	26/106	16/54	23/58	21/62
200+	11/42	12/45	14/49	22/106	14/54	22/58	18/61
400+	8/41	9/41	11/48	19/104	11/53	19/57	15/60

Table 7.17: *Memoized and memoizable methods.* Minimum method size is given in the *size* column, and each benchmark column shows successfully memoized methods over total memoizable methods.

size	comp	db	jack	javac	jess	mpeg	rt
10+	≈0%	≈0%	11%	5%	1%	≈0%	≈0%
20+	≈0%	≈0%	9%	5%	≈0%	≈0%	≈0%
30+	≈0%	≈0%	5.56%	4.73%	.14%	.01%	.22%
40+	≈0%	≈0%	5.56%	4.25%	.14%	.01%	.21%
50+	≈0%	≈0%	6%	4%	≈0%	≈0%	≈0%
100+	≈0%	≈0%	6%	4%	≈0%	≈0%	≈0%
200+	≈0%	≈0%	6%	4%	≈0%	≈0%	≈0%
400+	≈0%	≈0%	≈0%	4%	≈0%	≈0%	≈0%

Table 7.18: *Memoized bytecode execution.* Minimum method size is given in the *size* column, and each benchmark column shows the percentage of normal execution that was successfully memoized.

We examine the costs of both purity analysis and our memoization optimization in Figure 7.5. We show the execution time for our benchmarks under four scenarios: a base run with both purity and memoization disabled, an online purity analysis run, an online purity analysis with memoization run, and an offline purity analysis with memoization run. Memoization overhead is low, but this is indeed affected by our choice of minimum method size. For example, when executed with a minimum size of 5 bytecodes, some benchmarks required on the order of hours to complete. Purity analysis overhead itself is significant, but we actually consider it fairly tolerable for non-optimization purposes, especially when compared with heavyweight static analysis [SR05] that do not scale well [AKGE07]. Identifying weaker forms of purity involves an online escape analysis as well as inspection of potentially impure instructions; these are expensive operations, and they further add to the burden of providing cost-effective memoization. However, our implementation is not fully optimized, especially given that our prototype memoization consumer tracks entire data structures and not just the individual fields used by the memoized code. Accordingly, part of our future work involves improving the efficiency of our purity analysis and the accuracy and breadth of our memoization design.

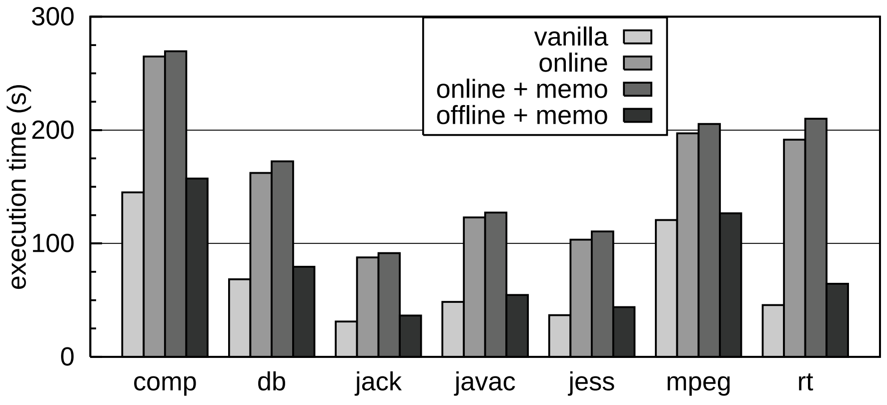


Figure 7.5: *Execution times.* Shown are execution times for vanilla SableVM, online purity analysis, online purity analysis with memoization, and offline purity analysis with memoization. The minimum method size for memoization is 50 bytecode instructions, and all other parameters remain unchanged.

Chapter 8
Conclusions and Future Work

8.1 Conclusions

We present several different purity definitions, which we term *strong*, *moderate*, *weak*, and *once-impure* in the order of conservativeness. *Strong purity* is the most conservative forms of purity in which no heap access is allowed, and this must trivially provide functional properties. *Weak purity* is a common used purity form, and *once-impure* purity can identify some forms of purity that cannot be observed statically.

Our dynamic purity analyses identify considerable amounts of purity for *weak purity* and *once-impure purity*, but relatively little purity for *strong purity* and *moderate purity*. Although *moderate purity* looks much more flexible than *strong purity*, not much gain can be achieved from *moderately* pure methods, because a *moderately* pure method is determined exclusively by its primitive input arguments. In a Java context, this can greatly reduce the possible purity, even given the ability to access and mutate locally allocated objects; reading input heap data, and object parameters is common in Java programs. More useful pure methods can be found through *weak purity*, which allows arbitrary heap reading at a cost of more complex input tracking. A further useful observation from our experiments shows that actual program behaviour is not predictable based on purely static observations. Statically pure methods are not always well-exercised dynamically, and opportunities for the execution of pure code are correspondingly diminished.

We proposed three different metrics for evaluating dynamic purity, and showed that while there was little variation in dynamic method purity over our benchmark suite, examination of dynamic invocation purity and dynamic bytecode purity revealed significant differences. The difference in dynamic invocation purity is divided into two groups, that is, over 80% invocations in *db* and *raytrace* are pure, with only about 30% pure invocation for other benchmarks. The difference in dynamic bytecode purity can be separated into three groups: again *db* and *raytrace* show high purity values, more than 50% bytecodes executed are from pure context, but then around 15–20% of bytecode are pure for *jack*, *javac*, and *jess*, but only about 3–5% for *comp* and *mpegaudio*. The distribution of these three metrics is despite the presence of many impure constructs: impurity in general is often bounded in dynamic scope, and potentially open to exploitation through appropriate dynamic purity tests. In all benchmarks, the main reason for impurity is the *PUTFIELD* instruction, either alone or combined with other reasons.

We show the cost of both purity analysis and our memoization optimization. Purity analysis overhead itself is significant, because online escape analysis will inspect all instruction executed; this is an expensive operation. However, we consider it fairly tolerable for non-optimization purposes, especially compared with heavyweight static analysis. Memoization overhead is low, but it is strongly affected by our choice of minimum memoizable method size. We also showed that consumer applications can impose strong constraints on usable purity information. In our memoization experiments, only a minimal amount of purity was exploited, and it may be the case that memoization is of limited use for non-functional languages. Nevertheless, our memoization client is a prototype design that can be optimized in several ways, most importantly by tracking individual fields instead of entire objects; we still hope to demonstrate that automatic memoization can be an effective optimization for Java programs.

8.2 Future Work

Our dynamic purity metrics are not exhaustive. Java bytecodes do not correlate perfectly with machine instructions or CPU cycles, and measuring these as well may provide more insight as to the true extent of dynamic purity. It might also be interesting to consider purity at finer granularities, such as loops, basic blocks, and individual instructions. If a method spends most of its time executing pure bytecodes inside a loop, and then executes an impure bytecode after completion of the loop, our analysis presently counts the entire execution as impure. As far as static metrics are concerned, it may be useful to examine *static invocation purity*, the percentage of call graph edges that have a pure method as a target, and *static bytecode purity*, the percentage of all bytecode instructions in the call graph contained in some pure method.

Our experimental framework is suitable for examining various forms of purity, and we aim to continue exploring purity notions. A fully parameterized analysis framework would facilitate detailed comparative evaluations of different purity definitions, and could be extended to analyse or even visualize the *evolution* of purity within a program. Additional manual analysis of native code beyond `clone()` and `arraycopy()` might identify more strongly pure methods; however, our analysis of impurity reasons showed that only a small percentage of execution is impure due to native methods alone. Our analyses were designed for memoization, and thus do not allow pure methods to return new objects, in contrast with Sălcianu's static analysis [SR05]. However, in our experiments, `ARETURN` is responsible for only 1% of dynamic method impurity, and it would be interesting to fully evaluate whether allowing new objects to escape from a pure method provides any real benefit. Other escape analyses that handle local impurities due to synchronization and exceptions might also be useful in certain contexts.

It will be interesting to consider purity on a per-input basis, as our analysis identified a fairly significant amount of unaccounted for execution in methods that change purity state more than once. Even weaker purity forms could be applied to *speculative* optimization [BS06, PV05] as a means to identify semi-pure code that has reduced potential to violate dependences and result in the roll-back of speculative computations. In general, we are optimistic about future opportunities for identifying and exploiting dynamic purity.

Bibliography

[ABH03] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *POPL'03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2003, pages 14–25.

[AKGE07] Shay Artzi, Adam Kieżun, David Glasser, and Michael D. Ernst. Combined static and dynamic mutability analysis. Technical Report MIT-CSAIL-TR-2007-020, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, March 2007.

[ALL96] Martin Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *ICFP'96: Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*, May 1996, pages 83–91.

[Ban79] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL'79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1979, pages 29–41.

[BCC$^+$05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT: International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *JC'02: Proceedings of the ACM SIGOPS France Journées Composants 2002: Systèmes à Composants Adaptables et Extensibles*, November 2002. `http://asm.objectweb.org/`.

[BS06] Saisanthosh Balakrishnan and Gurindar S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *ISCA'06: Proceedings of the 33rd International Symposium on Computer Architecture*, June 2006, pages 302–313.

[Bur90] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[CH03] Néstor Cataño and Marieke Huisman. Chase: A static checker for JML's assignable clause. In *VMCAI'03: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, January 2003, volume 2575 of *LNCS: Lecture Notes in Computer Science*, pages 26–40.

[CK88] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI'88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, June 1988, pages 57–66.

[Cla97] Lars Ræder Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, December 1997.

[DDHV03] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *OOPSLA'03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2003, pages 149–168.

[DL04] Yonghua Ding and Zhiyuan Li. A compiler scheme for reusing intermediate computation results. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Palo Alto, California, March 2004, page 279. IEEE Computer Society.

[DLZ07] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Dynamic purity analysis for Java programs, February 2007. `http://www.st.cs.uni-sb.de/models/jdynpur/`.

[DR02] Brian Demsky and Martin Rinard. Role-based exploration of object-oriented programs. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, May 2002, pages 313–324.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002, pages 234–245. ACM Press, New York, NY, USA.

[Fre97] Greg N. Frederickson. Ambivalent data structures for dynamic 2-edge–connectivity and k smallest spanning trees. *SIAM Journal on Computing*, 26(2):484–538, April 1997.

[Gag02] Etienne M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, December 2002. `http://sablevm.org`.

[HLY00] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *PLDI'00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000, pages 311–320.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international sympo-*

sium on Software testing and analysis, Portland, Oregon, United States, 2000, pages 113–123. ACM Press, New York, NY, USA.

[JG91] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *POPL'91: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1991, pages 303–310.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, May 2006.

[LH06] Ondřej Lhoták and Laurie J. Hendren. Context-sensitive points-to analysis: Is it worth it? In *CC'06: Proceedings of the 15th International Conference on Compiler Construction*, March 2006, volume 3923 of *LNCS: Lecture Notes in Computer Science*, pages 47–64.

[LLH05] Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using inter-procedural side-effect information in JIT optimizations. In *CC'05: Proceedings of the 14th International Conference on Compiler Construction*, April 2005, volume 3443 of *LNCS: Lecture Notes in Computer Science*, pages 287–304.

[LT95] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, February 1995.

[MH98] Paul McNamee and Marty Hall. Developing a tool for memoizing functions in C++. *SIGPLAN Not.*, 33(8):17–22, 1998.

[Mic68] Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, 2002, pages 1–11.

[MRR05] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, January 2005.

[Mul91] Ketan Mulmuley. Randomized multidimensional search trees (extended abstract): Dynamic sampling. In *SCG'91: Proceedings of the 7th Annual Symposium on Computational Geometry*, June 1991, pages 121–131.

[PQVR$^+$01] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In Reinhard Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction (CC '01)*, April 2001, volume 2027 of *LNCS: Lecture Notes in Computer Science*, pages 334–354.

[PT89] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *POPL'89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1989, pages 315–328.

[Pug88] William Pugh. An improved replacement strategy for function caching. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, Snowbird, Utah, United States, 1988, pages 269–276. ACM Press, New York, NY, USA.

[PV04] Christopher J. F. Pickett and Clark Verbrugge. Return value prediction in a Java virtual machine. In *VPW2: Proceedings of the 2nd Value-Prediction and Value-Based Optimization Workshop*, October 2004, pages 40–47.

[PV05] Christopher J. F. Pickett and Clark Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *LCPC'05: Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005, volume 4339 of *LNCS: Lecture Notes in Computer Science*, pages 304–318.

[Raz99] Chrislain Razafimahefa. A study of side-effect analyses for Java. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, December 1999.

[RLS$^+$01] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186, 2001.

[Rou04] Atanas Rountev. Precise identification of side-effect-free methods in Java. In *ICSM'04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, September 2004, pages 82–91.

[RR01] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. *Lecture Notes in Computer Science*, 2027:20+, 2001.

[SR05] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *VMCAI'05: Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, January 2005, volume 3385 of *LNCS: Lecture Notes in Computer Science*, pages 199–215. http://jppa.sourceforge.net.

[ST81] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *STOC'81: Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, May 1981, pages 114–122.

[Sta98] Standard Performance Evaluation Corporation. SPEC JVM Client98 benchmark suite, June 1998. http://www.spec.org/jvm98/.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL'96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996, pages 32–41.

[VR00] Raja Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, July 2000.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA'99: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, November 1999, pages 187–206.