Dynamic Purity Analysis for Java Programs

Haiying Xu   Christopher J. F. Pickett   Clark Verbrugge

School of Computer Science, McGill University
Montréal, Québec, Canada H3A 2A7
{hxu31,cpicke,clump}@sable.mcgill.ca

Abstract

The *pure* methods in a program are those that exhibit functional or side effect free behaviour, a useful property in many contexts. However, existing purity investigations present primarily static results. We perform a detailed examination of dynamic method purity in Java programs using a JVM-based analysis. We evaluate multiple purity definitions that range from strong to weak, consider purity forms specific to dynamic execution, and accomodate constraints imposed by an example consumer application, 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.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects; Procedures, functions, and subroutines; D.2.8 [*Software Engineering*]: Metrics—Complexity measures; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages; D.3.4 [*Programming Languages*]: Processors—Optimization; Run-time environments

General Terms Measurement, Experimentation, Languages

Keywords Purity, Side Effects, Memoization, Escape Analysis, Dynamic Analysis, Software Metrics, Java

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 when executed, or the extent of these side effects is limited in some way; furthermore, the extent to which it depends on previously available state may also be constrained. 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 [13], isolating and examining functional or *side effect free* fragments [30], and verification in model checking [8]. When optimizing, improved method purity information allows for less conservative assumptions, and has been used to drive compiler optimization [9, 19], novel hardware architectures [3], and caching or *memoization* of function calls [17].

PASTE'07 June 13–14, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-595-3/07/0006. . . $5.00

Although these applications demonstrate successful uses of various forms of purity and side effect data, the extent to which programs actually exhibit purity has not been fully investigated. Static analyses have shown the existence of large classes of pure methods [30, 31], 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 it clear 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 programs. 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 more varied when compared with previous data. Weaker forms of purity that allow for some impure invocations of a method expose slightly more purity, and we identify room for future work in this area. We explore this space and show the impact of dynamically changing purity properties, and also report the frequencies of different sources of impurity. Although we find that in some cases a large percentage of execution is pure, many methods are small or have other features which make efficient exploitation difficult.

A purity-specific consumer optimization for purity information provides constraints on the kind of purity data that is practically useful. To this end, we implement a prototype but non-trivial method memoization optimization. Memoization maps method arguments to return values, and allows for execution of reasonably pure methods to be safely bypassed; we extend the traditional definition by also considering heap dependences as inputs. This practical and direct application exposes additional constraints that a more abstract model of purity may not consider. It also serves as a useful functional test, and we examine the impact of subtle language and technical concerns on the use of purity information in a VM setting.

1.1 Contributions

We make the following specific contributions:

- The design and implementation of a dynamic purity analysis for Java bytecode that is available online and offline. Our analysis is scalable and handles SPECjvm98 at size 100 with acceptable overhead.
- Support for several different purity definitions that roughly correspond with previous work on static purity analysis, in order to facilitate comparison of disparate approaches. We also identify some forms of purity that are not observable statically.
- Three different metrics for evaluating the extent of dynamic purity: method purity, invocation purity, and bytecode purity. We apply these metrics to the results of a simple static analysis and multiple dynamic analyses corresponding to our purity definitions.

- A JVM implementation of the most traditional consumer of purity information, memoization. Although it does not achieve any speedup, it serves as a useful functional test module and a good basis for future investigations.

In the next section we discuss related work. In Section 3 we describe our static and dynamic analysis environments, present the different forms of purity we investigate, and illustrate the design of our memoization system. Section 4 provides experimental data, and in Section 5 we conclude and describe future work.

2. Related Work

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 [4]. Early work on side effect analysis concentrates on determining read and write sets in the context of functional or procedural languages [7, 10, 18]. Modern object-oriented languages introduce additional concerns through the use of virtual method dispatch and intensive use of dynamic memory, but these works bring out many of the core techniques, including interprocedural designs, and the need for reference analysis.

More recently side effect analysis has been investigated specifically in the context of Java programs. Razafimahefa presents an interprocedural side effect analysis for Java based on a *points-to* analysis [29]. His algorithm is derived from Steensgaard's points-to analysis for C [34], and is flow and context insensitive. Milanova *et al.* explore the use of context sensitive points-to information on side effect information [23]. 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 [19]. These works all focus on precisely identifying read and write sets, and not of course on identifying different notions of purity *per se*.

A study that bridges some of the gaps between purity and side effect analysis is one by Clausen [9]. 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 `<clinit>`, an empty inherited method in most classes. 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 [26].

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 [6, 20]. 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 [8].

Side effect free methods are identified by 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 [30]. 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. Sălcianu and Rinard [31] present a purity analysis based on a previous points-to and escape analysis [36]. 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.

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 [14].

Dallmeier *et al.* have examined dynamic purity analysis for Java programs concurrently with this work [11]. Their `jdynpur` tool uses the ASM bytecode manipulation framework [5] 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 [2]. Initially, reference parameters are classified as *unknown* with respect to mutability. A static analysis in Soot [35] 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.

We contribute one practical consumer of purity information with an implementation of method memoization. Our JVM-based design that interfaces with either an online or offline purity analysis is novel, but the idea of memoizing or caching function results is not. Similar techniques have been used for developing dynamic algorithms [15, 24, 32], and also for incremental computation [27]. Several works have looked at improving function memoization or caching efficiency [1, 17, 22] in the context of functional languages. Our memoization design is inspired by some of our earlier work on adapting memoization for use by Java-based return value prediction [25].

3. Design

Our investigation into purity is roughly divided into static and dynamic experimental designs, with a strong bias toward dynamic analysis as a means to complement existing static purity analysis work. We first introduce our static implementation, which is based on the Soot program analysis framework [35], and our dynamic evaluation of that static analysis. Then we describe our dynamic analysis implementation in the SableVM Java bytecode interpreter [16], and finally present the design of our memoization consumer application. In the first two subsections we also introduce the main purity definitions that we analyse experimentally.

3.1 Static Purity Analysis

Previous work has established that statically, a significant number of methods have fairly weak purity properties [30, 31]. Our static work considers the existence of *strong* purity in Java programs; a method is *strongly pure* iff it does not change or depend on any initial state beyond its primitive input values and returns the same result for the same input in any context. For Java, this means that a strongly pure method may not read from or write to heap or static data, perform synchronization, allocate objects, invoke a native method, throw explicit exceptions, or invoke any method not itself identified as strongly pure.

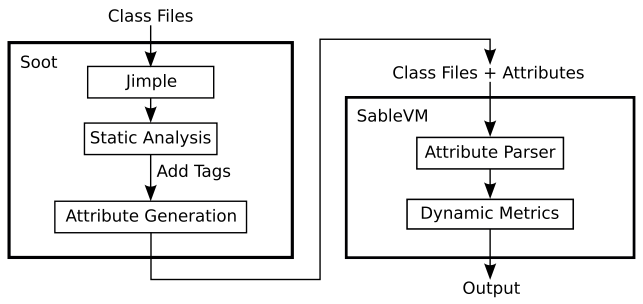

Figure 1. *Static purity analysis framework.*

The left half of Figure 1 shows the framework used to identify strong static purity. Application and library Java class files are used as input, and we perform a flow-insensitive analysis using Soot [35]. We follow an analysis strategy similar to that used by Clausen [9]. We first perform an intraprocedural scan of the instructions in each method. Instructions are classified as impure if listed in Table 1, otherwise as pure. In this simple analysis any method containing an impure instruction is treated as impure. We assume that exceptions do not propagate unchecked; more comprehensive safety analyses [28] would enable better identification of problematic code. Interprocedural analysis is then performed by propagating impurity up from the leaves of a Class Hierarchy Analysis (CHA-based) call graph [12] constructed by Soot, and computing a least fixed point for recursive invocations.

impurity	instructions
native code	native `INVOKE*`
heap access	`NEW`, `NEWARRAY`, `ANEWARRAY`, `MULTIANEWARRAY`, `GETFIELD`, `PUTFIELD`, `*ALOAD`, `*ASTORE`
static access	`GETSTATIC`, `PUTSTATIC`
synchronization	synchronized `INVOKE*`, synchronized `*RETURN`, `MONITORENTER`, `MONITOREXIT`
exceptions	`ATHROW`

Table 1. *Impure instructions for strong purity.*

A simple extension to this design allows for dynamic evaluation of our strong static purity analysis. We use Soot to write out purity information to Java class file attributes, and use the SableVM Java virtual machine [16] to read it in during class loading, as shown in the right half of Figure 1. The number of pure methods reached at runtime, the frequency of pure method invocations, and the percentage of bytecodes executed by pure methods all provide indications as to how well static results correlate with dynamic behaviour. We now consider performing the purity analysis itself dynamically.

3.2 Dynamic Purity Analysis

Statically, a method is conservatively determined to be pure for all possible executions, otherwise declared impure. However, for a given program run, a method declared impure statically may actually exhibit only pure control flow. We use a *dynamic purity analysis* to identify methods as pure or not based on their actual runtime behaviour, increasing the number of pure methods identified.

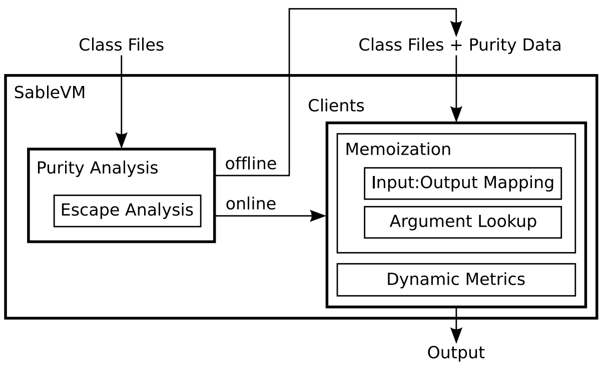

Figure 2. *Dynamic purity analysis framework.*

Figure 2 shows our framework for performing, using, and evaluating dynamic purity analyses. Initially, class files are read into SableVM, and method purity is determined by examining the executing instruction stream. The purity analysis module employs an online escape analysis sub-module that tracks writes to locally allocated objects. Purity information can be used immediately in an *online analysis*, or written out to a file for use by an *offline analysis* in a subsequent program run. Either kind of analysis can be used to drive client applications; the latter provides analysis results without analysis overhead, and is suitable for investigation of the upper bounds on exploitation of dynamic purity. The clients we provide include a memoization consumer that maps method inputs to outputs and looks up output based on current arguments, as discussed in Section 3.3, and the same dynamic metrics module used by the static purity analysis framework.

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. We now describe each kind of purity, along with the corresponding implementation and runtime component requirements.

Strong Dynamic Purity. Strong dynamic purity has the same criteria as strong static purity. However, we now consider only those instructions that are actually executed, as opposed to the entire static method body. 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. Impure designations are also propagated up the call stack to maintain the property that pure methods do not invoke impure methods. A method is marked as pure if it returns without encountering any impurity. 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.

Moderate Dynamic Purity. Some definitions of purity allow for arbitrary method behaviour provided that the input state is not altered [30]. Objects may be created and then altered in a pure method, provided such objects do not *escape* the method execution context. Pure methods may also call impure methods, provided the impurity is contained within the caller.

We use this to define *moderately* pure methods, with the additional constraint that they do 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. A moderately pure method may not: 1) invoke native methods; 2) read from or write to static or previously existing heap objects; 3) perform monitor operations; 4) throw exceptions; or 5) call a moderately impure method, unless the only source of impurity is that the callee method accesses and mutates objects local to the caller, or allows an object to escape to the caller. We make special exceptions for the native `java.lang.System.arraycopy()` and `java.lang.Object.clone()` methods, treating them as heap access and allocation instructions respectively, as these methods otherwise induce large amounts of impurity.

Our analysis must now examine the `*NEW*`, `GETFIELD`, `PUTFIELD`, `*ALOAD` and `*ASTORE` instructions more closely than was necessary for strong dynamic purity. The `*NEW*` instructions are used to determine 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.

An online escape analysis module is used to monitor object locality. Each frame in the call stack has an *object table* that stores all currently local objects. Newly allocated objects are stored in the table for the current frame, and any object allocated and returned by a callee is merged into it. Furthermore, callee `PUTFIELD` instructions with a reference argument can allow objects local to the callee to escape to the caller, requiring an update to its object table. 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 method and all intermediate methods as impure; otherwise, the instruction is considered pure.

Given that external heap reads are disallowed, a moderately pure method often does not have object parameters; if it does, it is unable to make any use of them outside of object reference comparisons. 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.

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. For *weak* dynamic purity, a `GETFIELD` operation is always safe, once the associated class is loaded, although `PUTFIELD` must still be considered in the context of our online escape analysis. This purity definition corresponds fairly closely with Rountev's [30].

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

3.3 Memoization

The forms of purity we define all ensure that pure methods have a functional property: there is a unique result for any given input. Even methods identified as weakly pure are thus candidates for *memoization*, an optimization that caches argument to return value mappings, jumping past actual method execution for repeated invocations with the same arguments. 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. One limitation of our current design is that after an initial impure invocation, any future impurity will disable memoization altogether. This could be remedied by an extension to the online analysis that tracks purity on a per-input basis.

As a consumer application for purity information, memoization imposes some additional constraints on its usage. The benefit obtained from jumping past method execution must exceed the cost of looking up the return value. In our case we use a few heuristic rules: the method must execute for long enough to be worth optimizing away, there must be a good hit rate after an initial warm up period, and the amount of input data to be processed cannot be too large.

Memoization is performed by associating hash tables with once-impure methods. Each time a memoizable method is executed its arguments are stored along with the return value. Primitive arguments can be stored directly, whereas reference arguments are "flattened", recursively gathering object type and primitive field values for all reachable types; circular data structures cannot presently be memoized. The advantages of storing only the object type are that garbage collection does not invalidate memoization tables and that a deep copy of an object will suffice when a different object was the original argument; the disadvantage is that direct object reference comparisons cannot be safely memoized, and so we must add the `ACMP_*` bytecodes to our impure operation list. However, we can still represent `null` object references, leaving `IF(NON)NULL` safe for memoization.

For subsequent invocations, the current arguments are hashed together, and a stored return value is simply substituted for the invocation if a match is found. This process is driven by either online or offline dynamic purity analysis information, as shown on the right hand side of Figure 2. The offline analysis, while unsound, eliminates the overhead of the purity analysis and allows for better evaluation of the memoization client in isolation. Upper bounds on memory consumption limit the number of methods that can be effectively memoized; a more efficient but also more complicated implementation would track the offsets of individual fields accessed during the initial purity determination, as opposed to entire objects.

4. Experimental Analysis

Experimental evaluation was conducted using the standard SPEC JVM98 benchmark suite at input size 100 [33] on a 2 GHz Athlon `x86_64` machine running Linux. Our memoization system does not yet support multithreading, and so we substitute the single-threaded *raytrace* benchmark for *mtrt*. Averages are computed as geometric means. We evaluate each form of purity described in the previous section using offline analyses, save for performance evaluation of the online analysis module. As we developed our analysis with the goal of identifying memoizable methods, we impose the additional constraint that `ACMP_*` are unsafe instructions in all dynamic purity analysis experiments. We find that this has little bearing on overall results, with a notable exception being *jess*.

4.1 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 [30, 31].

We introduce three new dynamic purity metrics. *Dynamic method purity* is calculated as the percentage of all methods reached at runtime that are pure, *dynamic invocation purity* as the percentage of all method invocations that are pure, and *dynamic bytecode purity* as the percentage of the executed bytecode instruction stream that is contained within a pure method. 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. In these experiments, our dynamic invocation purity metric does not account for impure methods called within a pure context, for better comparison with dynamic method purity.

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.

4.2 Static Purity Analysis

We evaluate our strong static purity analysis using the standard static method purity metric and our three dynamic purity metrics; results are shown in Table 2. The 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 analyse fewer methods in exchange for computation time [21]. On average, about 13% of methods are found to be strongly pure under all possible execution scenarios, a value in line with the progression of results from studies of weaker static purity forms. However, it is clear that not all of these methods will be invoked at runtime, and dynamically we find that only 5–6% of reached methods are statically identified as pure.

metric	comp	db	jack	javac	jess	mpeg	rt
static methods	14%	13%	13%	12%	13%	13%	13%
dynamic methods	6%	6%	6%	5%	5%	6%	5%
invocations	≈0%	2%	10%	10%	6%	16%	3%
bytecode	≈0%	2%	1%	≈0%	≈0%	2%	≈0%

Table 2. *Strong static purity.* The *static methods* row shows the percentage of all methods in the call graph identified as statically pure at compile time; the *dynamic methods* 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 method.

Further examination of these methods reveals that all are small, consisting of less than 20 bytecode instructions. We find that many strongly pure methods are executed infrequently; others are executed frequently but also trivially empty, examples being `<clinit>` and `<init>`. This qualitative analysis of post-execution purity data is borne out by our dynamic bytecode purity metric, which finds that a very small percentage of bytecode execution is pure, even when dynamic invocation purity is considerably higher than dynamic method purity, as exhibited by *jack*, *javac*, and *mpegaudio*.

4.3 Dynamic Purity Analysis

Strong dynamic purity is a weaker form of purity than its static equivalent, and the results in the first row of Tables 3, 4, and 5 improve on the runtime use of strong static purity in rows 2–4 of Table 2. In Table 3, up to 4% more pure methods are reached using strong dynamic purity. Some of these methods are also invoked with significant frequency: 13% more pure invocations for *db* are shown in Table 4. Nevertheless, the overall impact remains small, with Table 5 showing no more than 3% of all bytecode instructions being executed in a pure context, and a maximum gain over strong static purity of 1%.

Our moderate dynamic purity definition further relaxes purity constraints. We observe marginal improvements to all runtime measurements, but overall do not find any large gains. Recall that under moderate dynamic purity, methods are not allowed to read 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 6 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.

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 4 and 5. In particular, *db* and *raytrace* execute a high percentage of the bytecode instruction stream within a pure context. However, as we will show in our memoization experiments, many of these methods are actually unsuitable for profitable exploitation. For other benchmarks, weak purity does not result in such large 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.

Once-impure results are shown in the last row of Tables 3, 4, and 5. We observe small gains in dynamic method purity for all benchmarks, and slightly larger gains for some benchmarks in the context of dynamic invocation and bytecode purity, but negligible gains for others. Although the changes are small, we still observe a general trend in our dynamic metrics: dynamic method purity is fairly similar between benchmarks, differences in dynamic invocation purity are greater, separating programs into two groups, and differences in dynamic bytecode purity are greater still, separating programs into three distinct groups. This tendency of our metrics to polarize benchmarks is a useful property.

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. Table 7 provides a detailed breakdown according to the number and kinds of methods captured and ignored by our purity analysis. 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 analysed the extent of missed opportunities in the second sec-

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 3. *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	≈ 0%	15%	13%	11%	10%	16%	8%
moderate	≈ 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 4. *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	≈0%	3%	1%	1%	1%	2%	1%
moderate	≈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 5. *Dynamic bytecode purity.* Percentage of total bytecode instruction stream that is contained in a pure method for different dynamic purity definitions.

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 6. *All methods with reference parameters or return values.*

tion using our dynamic metrics, and found a surprising amount of unaccounted for execution, particularly for *jack*, *javac*, and *jess*.

Methods themselves may be impure for multiple reasons or only for a single reason. Tables 8, 9, and 10 give details as to which bytecodes actually cause impurity under once-impure dynamic purity. In Table 8, 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 9, 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 10 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 analysing and handling methods marked impure due to execution of multiple kinds of impure bytecodes.

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. *Analysed and unanalysed methods.* The first section shows reached method counts accounted for by our analyses: 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 for the methods identified in the second section.

impurity	comp	db	jack	javac	jess	mpeg	rt
`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%
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 8. *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*.

impurity	comp	db	jack	javac	jess	mpeg	rt
`ACMP_*`	≈0%	≈0%	12%	6%	54%	≈0%	≈0%
`PUTFIELD`	81%	82%	45%	25%	24%	40%	71%
`*STATIC`	≈0%	≈0%	2%	≈0%	3%	≈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 9. *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%
`PUTFIELD+`	79%	13%	48%	66%	45%	89%	66%
others	≈0%	≈0%	3%	2%	1%	≈0%	1%

Table 10. *Reasons for dynamic bytecode impurity.*

4.4 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 3.3, we cannot compare object addresses directly using `ACMP_*`, and so the usable purity information is a strict subset of what once-impure purity can actually identify.

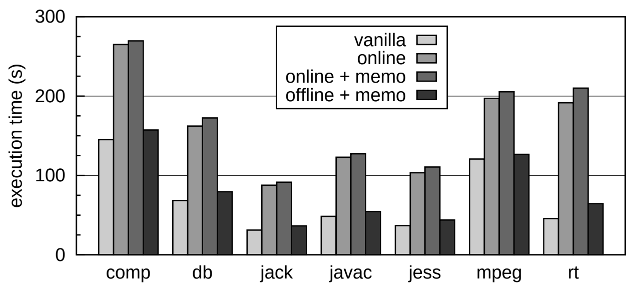

Figure 3. *Execution times.* Shown are execution times for vanilla SableVM, online purity analysis, online purity analysis with memoization, and offline purity analysis with memoization. The minimum method size for memoization is 50 bytecode instructions, and all other parameters remain unchanged.

size	comp	db	jack	javac	jess	mpeg	rt
20+	15/42	15/45	18/49	33/106	18/54	27/60	24/62
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 11. *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.

The impact of these constraints on memoization is significant. Table 11 shows low absolute numbers of methods memoized under our present cost constraints, and Table 12 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 5. 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.

We examine the costs of both purity analysis and our memoization optimization in Figure 3. 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 analyses [31] that do not scale well [2]. 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 their individual fields. Accordingly, part of our future work involves improving the efficiency of our purity analysis and the accuracy and breadth of our memoization design.

size	comp	db	jack	javac	jess	mpeg	rt
20+	≈0%	≈0%	9%	5%	≈0%	≈0%	≈0%
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 12. *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.

5. Conclusions & Future Work

Our dynamic purity analyses identify considerable amounts of purity, and evaluation 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. This 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. However, 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.

Our dynamic purity metrics are not exhaustive. Java bytecodes do not correlate well 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 faciliate 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 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 [31]. However, in our experiments, `ARETURN` is responsible for only 1% of dynamic method impurity, and it will 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 [3, 26] 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.

References

[1] U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *POPL'03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–25, Jan. 2003.

[2] S. Artzi, A. Kieżun, D. Glasser, and M. 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, Mar. 2007.

[3] S. Balakrishnan and G. 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*, pages 302–313, June 2006.

[4] J. 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*, pages 29–41, Jan. 1979.

[5] E. Bruneton, R. Lenglet, and T. 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*, Nov. 2002. `http://asm.objectweb.org/`.

[6] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT: International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[7] M. 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.

[8] N. Cataño and M. 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*, volume 2575 of *LNCS: Lecture Notes in Computer Science*, pages 26–40, Jan. 2003.

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

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

[11] V. Dallmeier, C. Lindig, and A. Zeller. Dynamic purity analysis for Java programs, Feb. 2007. `http://www.st.cs.uni-sb.de/models/jdynpur/`.

[12] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95: Proceedings of the 9th European Conference on Object-Oriented Programming*, volume 952 of *LNCS: Lecture Notes in Computer Science*, pages 77–101, Aug. 1995.

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

[14] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *OOPSLA'03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 149–168, Oct. 2003.

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

[16] E. M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, Dec. 2002.

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

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

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

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

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

[22] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, Feb. 1995.

[23] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, Jan. 2005.

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

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

[26] C. J. F. Pickett and C. 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*, volume 4339 of *LNCS: Lecture Notes in Computer Science*, pages 304–318, Oct. 2005.

[27] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *POPL'89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 315–328, Jan. 1989.

[28] F. Qian, L. J. Hendren, and C. Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *CC'02: Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *LNCS: Lecture Notes in Computer Science*, pages 325–342, Apr. 2002.

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

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

[31] A. Sălcianu and M. 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*, volume 3385 of *LNCS: Lecture Notes in Computer Science*, pages 199–215, Jan. 2005. `http://jppa.sourceforge.net`.

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

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

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

[35] R. Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, July 2000. `http://www.sable.mcgill.ca/soot/`.

[36] J. Whaley and M. 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*, pages 187–206, Nov. 1999.