Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors *

Michael D. Bond
Dept. of Computer Sciences
University of Texas at Austin
mikebond@cs.utexas.edu

Nicholas Nethercote
National ICT Australia
njn@csse.unimelb.edu.au

Stephen W. Kent
Dept. of Computer Sciences
University of Texas at Austin
stephenkent@mail.utexas.edu

Samuel Z. Guyer
Dept. of Computer Science
Tufts University
sguyer@cs.tufts.edu

Kathryn S. McKinley
Dept. of Computer Sciences
University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

Programs sometimes crash due to *unusable* values, for example, when Java and C# programs dereference null pointers and when C and C++ programs use undefined values to affect program behavior. A stack trace produced on such a crash identifies the effect of the unusable value, not its cause, and is often not much help to the programmer.

This paper presents efficient *origin tracking* of unusable values; it shows how to record where these values come into existence, correctly propagate them, and report them if they cause an error. The key idea is *value piggybacking*: when the original program stores an unusable value, value piggybacking instead stores origin information in the spare bits of the unusable value. Modest compiler support alters the program to propagate these modified values through operations such as assignments and comparisons. We evaluate two implementations: the first tracks null pointer origins in a JVM, and the second tracks undefined value origins in a memory-checking tool built with Valgrind. These implementations show that origin tracking via value piggybacking is fast and often useful, and in the Java case, has low enough overhead for use in a production environment.

* This work is supported by an Intel fellowship, NSF CCF-0429859, NSF CCR-0311829, NSF EIA-0303609, DARPA F33615-03-C-4106, Intel, IBM, and Microsoft. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
Copyright © 2007 ACM 978-1-59593-786-5/07/0010. . . $5.00

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids

General Terms Reliability, Performance, Experimentation

Keywords Debugging, Low-Overhead Run-Time Support, Null Pointer Exceptions, Undefined Values, Managed Languages, Java, Valgrind

1. Introduction

Finding the causes of bugs is hard, both during testing and after deployment. One reason is that a bug's effect is often far from its cause. Liblit et al. examined bug symptoms for various programs and found that inspecting the methods in a stack trace did not identify the method containing the error for 50% of the bugs [21].

This paper offers help for a class of bugs due to *unusable values* that either cause a failure directly or result in erroneous behavior. In managed languages such as Java and C#, the *null value* is unusable and causes a null pointer exception when dereferenced. In languages like C and C++, *undefined values*—those that are uninitialized, or derived from undefined values—are unusable, and their use can cause various problems such as silent data corruption, altered control flow, or a segmentation fault.

Failures due to unusable values are difficult to debug because (1) the origin of the unusable value may be far from the point of failure, having been propagated through assignments, operations, and parameter passing; and (2) unusable values themselves yield no useful debugging information. At best, the programmer sees a stack trace from the crash point, but this identifies the effect of the unused value, not its source. This problem is particularly bad for deployed software since the bug may be difficult to reproduce.

Null pointer exceptions are a well-known problem for Java programmers. Eric Allen writes the following in *Diagnosing Java* for the IBM developerWorks Java Zone [2]:

> *Of all the exceptions a Java programmer might encounter, the null-pointer exception is among the most dreaded, and for good reason: it is one of the least informative exceptions that a program can signal. Unlike, for example, a class-cast exception, a null-pointer exception says nothing about what was expected instead of the null pointer. Furthermore, it says nothing about where in the code the null pointer was actually assigned. In many null-pointer exceptions, the true bug occurs where the variable is actually assigned to null. To find the bug, we have to trace back through the flow of control to find out where the variable was assigned and determine whether doing so was incorrect. This process can be particularly frustrating when the assignment occurs in a package other than the one in which the error was signaled.*

Our goal is to provide this information automatically and at a very low cost.

Unused value errors are similarly difficult to debug for programs written in unmanaged languages such as C and C++. For example, Memcheck [31] is a memory checking tool built with the dynamic binary instrumentation framework Valgrind [26]. Memcheck can detect dangerous uses of undefined values, but prior to this paper gave no origin information about those values. Requests for such origin information from Memcheck users are common enough that the FAQ explains the reason for this shortcoming [26].

The key question is: why does this variable contain an unusable value? We answer this question and solve this problem by introducing *origin tracking*. Origin tracking records program locations where unusable values are assigned, so they can be reported at the time of failure. We leverage the property that unusable values are difficult to debug because they contain no useful information and store the origin information *in place of the unusable values themselves*, which is a form of *value piggybacking*. Value piggybacking requires no additional space, making origin tracking efficient. With some modifications to program execution, origin values flow freely through the program: they are copied, stored in the heap, or passed as parameters. They thus act like normal unusable values until the programs uses them inappropriately, whereupon we report the origin, which is often exactly what the programmer needs to diagnose the defect.

We implement two separate origin tracking implementations: one tracks null pointer origins in Java, and the other tracks origins of undefined values in binaries of C, C++, Fortran, and other languages. For the null pointer implementation, we modify Jikes RVM [3, 16], a high-performance virtual machine. The undefined value implementation is built on top of the memory checking tool Memcheck [31]. Both origin tracking implementations are publicly available (Sections 2 and 5).

Our results show that origin tracking is effective at reporting the origins of Java null pointer exceptions and adds minimal overhead to overall execution time (4% on average), making it suitable for deployed programs. We collected a test suite of 12 null pointer exceptions from publicly available Java programs with documented, reproducible bugs. Based on examining the stack trace, origin, source code, and bug report, we determine that origin tracking correctly reports the origin for all of the 12 bugs in this test suite, provides information not available from the stack trace for 8 bugs, and is useful for debugging in 7 of those 8 cases.

Origin tracking in Memcheck is accurate for 32-bit undefined values but fails to identify origins for values too small to hold a program location. It finds 72% of undefined value errors involving 32-bit data in 20 C and C++ programs. Memcheck is a heavy-weight instrumentation tool, and it already slows programs down by a factor of 28 on average. Our additions to Memcheck add on average no measurable overhead.

The main contributions of this work are:

- *Value piggybacking* to record and propagate debugging information in place of unusable values.
- Two new applications of value piggybacking for *origin tracking*: (a) identifying the origins of null pointer exceptions in deployed Java programs cheaply and (b) identifying the origins of undefined values in C, C++, and Fortran programs at testing time.

These applications are notable because previous value piggybacking applications conveyed only one or two bits of information per value (e.g., [20]), whereas we show it can convey more useful information. Since our approach requires modest changes to the Java virtual machine and incurs very low overhead, commercial JVMs could rapidly deploy it to help programmers find and fix bugs.

2. Origin Tracking in Java

This section describes our implementation for tracking the origins of null references in Java programs. In each subsection, we first describe the general strategy used to piggyback information on null references, and then describe the details specific to our Jikes RVM implementation.

2.1 Supporting Nonzero Null References

Java virtual machines typically use the value zero to represent a null reference. This choice allows operations on null references, such as comparisons and detection of null pointer exceptions, to be simple and efficient. However, the Java VM specification does not require the use of a particular concrete value for encoding null [22]. We modify the VM to instead represent null using a range of reserved addresses. Objects may not be allocated to this range, allowing null and object references to be differentiated easily.

Our implementation reserves addresses in the lowest 32nd of the address space: `0x00000000`–`0x07ffffff`. That is, a reference is null if and only if its highest five bits are

	Java semantics	Standard VM	Origin tracking
(a) Assignment of null constant	`obj = null;`	`obj = 0;`	`obj = this_location;`

(b) Object allocation	`obj = new Object();`	... *allocate object* ...
		`foreach ref slot i`
		`  obj[i] = 0;`
		... *call constructor* ...
			... *allocate object* ...
			`foreach ref slot i`
			`  obj[i] = this_location;`
			... *call constructor* ...

(c) Null reference comparison	`b = (obj == null);`	`b = (obj == 0);`	`b = ((obj & 0xf8000000) == 0)`

(d) General reference comparison	`b = (obj1 == obj2);`	`b = (obj1 == obj2);`	`if (((obj1 & 0xf8000000) == 0))`
			`  b = ((obj2 & 0xf8000000) == 0);`
			`else`
			`  b = (obj1 == obj2);`

Table 1. How origin tracking handles uses of null in Java code. Column 2 shows example code involving null. Column 3 shows typical semantics for an unmodified VM. Column 4 shows semantics in a VM implementing origin tracking.

zero. The remaining 27 bits encode a program location as described in the next section.

As an alternative to a contiguous range of null values, null could be represented as any value with its lowest bit set. Object references and null could be differentiated easily since object references are word-aligned. VMs such as Jikes RVM that implement null pointer exceptions using hardware traps could instead use alignment traps. We did not implement this approach since unmodified Jikes RVM compiled for IA32 performs many unaligned accesses already (alignment checking is disabled by default on IA32).

2.2 Encoding Program Locations

The strategy described above provides 27 bits for encoding an origin in a null reference. We use one of these bits to distinguish between two cases: nulls that originate in a method body (the common case) and nulls that result from uninitialized static fields. In the latter case, the remaining 26 bits identify the particular field. In the former case, we encode the program location as a <method, line number> pair using one more bit to choose from among the following two layouts for the remaining 25 bits:

1. The default layout uses 13 bits for method ID and 12 bits for bytecode index, which is easily translated to a line number.
2. The alternate layout uses 8 bits for method ID and 17 bits for bytecode index, and is used only when the bytecode index does not fit in 12 bits. The few methods that fall into this category are assigned separate 8-bit identifiers.

We find these layouts handle all the programs in our test suite for origin tracking. Alternatively, one could assign a unique 27-bit value to each program location that assigns null via a lookup table. This approach would use space proportional to the size of the program to store the mapping. Our approach adds no space since per-method IDs already exist in the VM.

2.3 Redefining Program Operations.

Our implementation redefines Java operations to accommodate representing null using a range of values.

Null Assignment At null assignments, our modified VM assigns the 27-bit value corresponding to the current program location (method and line number) instead of zero. The dynamic compiler computes this value at compile time. Table 1, row (a) shows the null assignment in Java and its corresponding semantics for an unmodified VM and for the origin a VM implementing origin tracking.

Object Allocation When a program allocates a new object, whether scalar or array, its reference slots are initialized to null by default. VMs implement this efficiently by allocating objects into mass-zeroed memory. Since origin tracking uses nonzero values to represent null, our modified VM adds code at object allocations that initializes each reference slot to the program location, as shown in Table 1, row (b). These values identify the allocation site as the origin of the null.

Since reference slot locations are known at allocation time, we can modify the compiler to optimize the code inserted at allocation sites. For hot sites (determined from profiles, which are collected by Jikes RVM and other VMs), the compiler inlines the code shown in the last column of Table 1, row (b). If the number of slots is a small, known constant (true for all small scalars, as well as small arrays where the size is known at compile time), the compiler flattens the loop.

Static fields are also initialized to null, but during class initialization. We modify the VM's class initialization to fill each static reference field with a value representing the static field (the VM's internal ID for the field). Of the 12 bugs we evaluate in Section 3, one manifests as a null assigned at class initialization time.

Null Comparison To implement checking whether a reference is null, VMs compare the reference to zero. Origin tracking requires a more complex comparison since null may have any value from a range. Since our implementation uses the range 0x00000000–0x07ffffff for null, it implements the null test using a bitwise AND with 0xf8000000, as shown in Table 1, row (c).

General Reference Comparison A more complex case is when a program compares two references. With origin tracking, two references may have different underlying values even though both represent null. To handle this case, the origin tracking implementation uses the following test: two references are the same if and only if (1) they are both nonzero null values or (2) their values are the same. Table 1, row (d) shows the modified VM implementation.

We optimize this test for the common case: non-null references. The instrumentation first tests if the first reference is null; if so, it jumps to an out-of-line basic block that checks the second reference. Otherwise, the common case performs a simple check for reference equality, which is sufficient since both references are now known to be non-null.

2.4 Implementation in Jikes RVM

We implement origin tracking in Jikes RVM, a high-performance Java-in-Java virtual machine [5, 16]. Our implementation is publicly available on the Jikes RVM Research Archive [17].

Jikes RVM uses two dynamic compilers. When a method first executes, Jikes RVM compiles it with a non-optimizing, baseline compiler. As a method becomes hotter, Jikes RVM recompiles it with an optimizing compiler at successively higher levels of optimization. We modify both compilers to redefine Java operations to support piggybacking of origins on null values.

Our implementation stores program locations instead of zero (Table 1, rows (a) and (b)) only in application code, not in VM code or in the Java standard libraries. This choice reflects developers' overriding interest in source locations in their application code. Since the VM is not implemented entirely in pure Java—it needs C-style memory access for low-level runtime features and garbage collection—generating nonzero values for null in the VM and libraries would confuse parts of the VM that assume null is zero. Since null references generated by the application sometimes make their way into the VM and libraries, our implementation modifies all reference comparisons to handle nonzero null references (Table 1, rows (c) and (d)) in the application, libraries, and VM.

Some VMs including Jikes RVM catch null pointer exceptions using a hardware trap handler: since low memory is protected, dereferencing a null pointer generates the signal SIGSEGV. The VM's custom hardware trap handler detects this signal and returns control to the VM, which throws a null pointer exception. The origin tracking implementation protects the address range 0x00000000–0x07ffffff so null dereferences will result in a trap. For origin tracking, we modify the trap handler to identify and record the culprit base address, which is the value of the null reference. When control is returned to the VM, it decodes the value into a program location and reports it together with the null pointer exception. VMs that use explicit null checks to detect null pointer exceptions could simply use modified null checks as described in Section 2.3.

The Java Native Interface (JNI) communicates with unmanaged languages such as C and C++, allowing unmanaged code to access Java objects. The unmanaged code assumes that null is zero. We therefore modify the VM to identify null parameters passed to JNI methods, and to replace them with zero. This approach loses origin information for these parameters but ensures correct execution.

3. Finding and Fixing Bugs in Java Programs

This section describes a case study using origin tracking to identify the causes of 12 failures in eight programs. These results are summarized in Table 2, which contains the lines of code measured with the Unix wc command; whether the origin was identified; whether the origin was identifiable trivially; and how useful we found the origin report (these criteria are explained in detail later in this section). We describe three of the most interesting cases (Cases 1, 2, and 3) in detail below and the other nine in the appendix. In summary, our experience showed:

- The usefulness of origin information depends heavily on the complexity of the underlying defect. In some cases, it is critical for diagnosing a bug. Given the extremely low cost of origin tracking (see Section 4), there is little reason not to provide this extra information, which speeds debugging even when a defect is relatively trivial.
- Bug reports often do not contain sufficient information for developers to diagnose or reproduce a bug. Origin tracking provides extra information for users to put in bug reports in addition to a stack trace.
- It is not always clear whether the defect lies in the code producing the null value, or in the code dereferencing it (e.g., the dereferencing code should add a null check). A stack trace alone only provides information about the dereferencing code. Origin tracking allows programmers to consider both options when formulating a bug fix.
- Null pointer exceptions often involve a null value flowing between different software components, such as application code and library code. Therefore, even when the origin and dereference occur close together it can be difficult to evaluate the failure without a full understanding of both components. For example, a programmer might trigger a null pointer exception in a library method by passing it an object with a field that is unexpectedly null. Origin tracking indicates which null store in the application code is responsible, without requiring extra knowledge or source code for the library.

Case	Program	Lines	Exception description	Origin?	Trivial?	Useful?
1	Mckoi SQL DB	94,681	Access closed connection	Yes	Nontrivial	Definitely useful
2	FreeMarker	64,442	JUnit test crashes unexpectedly	Yes	Nontrivial	Definitely useful
3	JFreeChart	223,869	Plot without x-axis	Yes	Nontrivial	Definitely useful
4	JRefactory	231,338	Invalid class name	Yes	Nontrivial	Definitely useful
5	Eclipse	2,425,709	Malformed XML document	Yes	Nontrivial	Most likely useful
6	Checkstyle	47,871	Empty default case	Yes	Nontrivial	Most likely useful
7	JODE	44,937	Exception decompiling class	Yes	Nontrivial	Most likely useful
8	Jython	144,739	Use built-in class as variable	Yes	Nontrivial	Potentially useful
9	JFreeChart	223,869	Stacked XY plot with lines	Yes	Somewhat nontrivial	Marginally useful
10	Jython	144,739	Problem accessing __*doc*__ attribute	Yes	Somewhat nontrivial	Marginally useful
11	JRefactory	231,338	Package and import on same line	Yes	Trivial	Not useful
12	Eclipse	2,425,709	Close Eclipse while deleting project	Yes	Trivial	Not useful

Table 2. The diagnostic utility of origins returned by origin tracking in Java. Cases 1, 2, and 3 are described in detail in Section 3; the rest are described in the appendix. Bug repositories are on SourceForge [32] except for Eclipse [10] and Mckoi SQL Database [24].

3.1 Evaluation Criteria

For each error, we evaluate how well origin tracking performs using three criteria:

Origin identification. Does origin tracking correctly return the method and line number that assigned the null responsible for the exception?

Triviality. Is the stack trace alone, along with the source code, sufficient to identify the origin? In 8 of 12 null pointer exceptions, the origin is not easy to find via inspection of the source location identified by the stack trace.

Usefulness. Does knowing the origin help with understanding and fixing the defect? Although we are not the developers of these programs, we examined the source code and also looked at bug fixes when available. We believe that the origin report is not useful or marginally useful for one-third of the cases; potentially or most likely useful for another third; and most likely or definitely useful for the remaining third of the cases.

3.2 Origin Tracking Case Studies

We now describe three bugs that highlight origin tracking's role in discovering the program defect.

Case 1: Mckoi SQL Database: Access Closed Connection The first case highlights an important benefit of origin tracking: it identifies a null store far away from the point of failure (possibly in another thread). The location of the store identifies

The bug report comes from a user's message on the mailing list for Mckoi SQL Database version 0.93, a database management system for Java (Message 02079). The user reports that the database throws a null pointer exception when the user's code attempts to execute a query. The bug report contains only the statement `dbStatement.executeUpdate(dbQuery);` and a stack trace, so we use information from the developer's responses to construct a test case. Our code artificially induces the failure but captures the essence of the problem.

The stack trace is shown in Figure 1(a). This information presents two problems for the application developer. First, the failure is in the library code, so it cannot be easily debugged. Second, it indicates simply that the query failed, with no error message or exception indicating why.

Our origin information, shown in Figure 1(b), reveals the reason for the failure: the null store occurred in `AbstractJDBCDatabaseInterface.internalDispose()` at line 298. This method is part of closing a connection; line 298 assigns null to the connection object reference. The cause of the failure is that the query attempts to use a connection that has already been closed.

The origin information may be useful to both the application user and the database library developers. Users can probably guess from the name of the method `Abstract-`

```
java.lang.NullPointerException:
 at com.mckoi.database.jdbcserver.JDBCDatabaseInterface.
        execQuery():213
 at com.mckoi.database.jdbc.MConnection.
        executeQuery():348
 at com.mckoi.database.jdbc.MStatement.
        executeQuery():110
 at com.mckoi.database.jdbc.MStatement.
        executeQuery():127
 at Test.main():48
```

(a)

```
Origin:
 com.mckoi.database.jdbcserver.
     AbstractJDBCDatabaseInterface.internalDispose():298
```

(b)

Figure 1. Case 1: VM output for Mckoi SQL Database bug. (a) The stack trace shows the query failed inside the library. (b) Origin tracking suggests that the failure is due to a closed connection.

```
java.lang.NullPointerException:
  at freemarker.template.WrappingTemplateModel.wrap():131
  at freemarker.template.SimpleHash.get():197
  at freemarker.core.Environment.getVariable():959
  at freemarker.core.Identifier._getAsTemplateModel():70
  at freemarker.core.Expression.getAsTemplateModel():89
  ...
  at junit.textui.TestRunner.main():138
```

(a)

```
Origin: freemarker.template.DefaultObjectWrapper.instance
```

(b)

Figure 2. Case 2: VM output for FreeMarker bug. (a) The stack trace shows the fault location. (b) The null's origin is an uninitialized static field.

```
java.lang.NullPointerException:
  at org.jfree.chart.plot.FastScatterPlot.draw():447
  at Bug2.test():16
  at Bug2.main():9
```

(a)

```
Origin: Bug2.test():13
```

(b)

Figure 3. Case 3: VM output for JFreeChart bug. (a) The stack trace shows a failure inside the library. (b) Origin tracking identifies the error is caused by a null from user code.

`JDBCDatabaseInterface.internalDispose()` that the problem is a closed connection, and can plan for this possibility in their application logic. The developers can also modify the `execQuery()` method to check for a closed connection and to throw a useful `SQLException` that reports the reason, as noted in an existing response on the mailing list.

Case 2: FreeMarker: JUnit Test Crashes Unexpectedly The second case illustrates how origin tracking helps diagnose errors when the null reference passes from variable to variable by assignment. The case is also interesting because at first glance the initial assignment appears to be non-null, but it is in fact null because of a static initialization ordering issue.

FreeMarker 2.3.4 is a Java library that generates output such as HTML and source code using user-defined templates. We reproduced an exception in the library using test code posted by a user (Bug 1354173). Figure 2 shows the exception stack trace.

The exception occurs at line 131 of `wrap()`, which tries to dereference `defaultObjectWrapper`, which is null. Previously, `defaultObjectWrapper` was assigned the value of the static, final field `DefaultObjectWrapper.instance`. At first glance, it appears that `DefaultObjectWrapper-.instance` is properly initialized:

```
static final DefaultObjectWrapper instance =
  new DefaultObjectWrapper();
```

However, due to a circular initialization dependency between `WrappingTemplateModel` and `DefaultObject-Wrapper`, `instance` is in fact initialized to null. Origin tracking helps diagnose this error by reporting the uninitialized static field `instance` as the origin of the offending null. The origin should be quite useful for diagnosing the bug since (1) the null passes through a variable, and (2) it is not intuitive that the original assignment assigns null. A responder to the bug report also came to the conclusion that the exception is a result of static class initialization ordering, but to our knowledge it has not been fixed in any version of FreeMarker.

Case 3: JFreeChart: Plot Without X-Axis This case involves a small test program provided by a bug reporter that causes a null pointer exception inside JFreeChart 1.0.2, a graphing library (Bug 1593150). This case, like the first case in this section, represents an important class of failures for which origin tracking is useful: the failure is induced by the application, but since it occurs inside the library the programmer has no easy way to interpret the stack trace or to debug the library code.

The following is code provided by the user, annotated with line numbers:

```
12:   float[][] data = {{1.0f,2.0f},{3.0f,4.0f}};
13:   FastScatterPlot plot =
        new FastScatterPlot(data, null, null);
14:   Button aButton = new Button();
15:   Graphics2D graphics =
        (Graphics2D)(aButton.getGraphics());
16:   plot.draw(graphics, new Rectangle2D.Float(),
                new Point2D.Float(), null, null);
```

Figure 3(a) shows the exception stack trace. The method `FastScatterPlot.draw()`, called from line 16 of the user code, throws a null pointer exception. This stack trace is not very helpful to the library user, who may not have access to or be familiar with the JFreeChart source code.

On the other hand, origin tracking provides information that is directly useful to the user: the origin is line 13 of the user's `test()` (Figure 3). The user can quickly understand that the exception occurs because the code passes null as the x-axis parameter to the `FastScatterPlot` constructor.

While the origin allows a frustrated user to modify his or her code immediately, it also suggests a better long-term fix: for JFreeChart to return a helpful error message. The developers diagnosed this bug separate from us, and their solution, implemented in version 1.0.3, causes the constructor to fail with an error message if the x-axis parameter is null.

Remaining Cases The appendix contains details of the remaining nine bugs.

4. Java Run-Time Performance

This section evaluates the performance impact of piggybacking origins on null values in Java programs.

4.1 Methodology

Execution For our Java experiments we run Jikes RVM in two different just-in-time compilation modes: *adaptive* mode and *replay* mode. The adaptive mode results show the overall effect of origin tracking in a deployed setting, while the replay mode results allow us to measure the individual performance factors.

In adaptive compilation mode, which is the default, Jikes RVM dynamically identifies frequently executed methods and recompiles them at higher optimization levels. Many JVMs use this strategy, and thus the adaptive mode results represent the expected bottom-line cost of using origin tracking in deployed software.

The way adaptive mode is implemented, however, makes it difficult to break out the components of performance overhead: how much is due to the extra cost of *compiling* the origin tracking code, and how much is due to *executing* the origin tracking code. The adaptive compiler uses timer-based sampling to make compilation decisions, and therefore the additional cost of value piggybacking alters these decisions, making it difficult to compare runs with and without origin tracking.

To address this problem, we use *replay compilation* methodology, which is deterministic. Replay compilation forces Jikes RVM to compile the same methods in the same order at the same point in execution on different executions and thus avoids variability due to the compiler.

Replay compilation uses *advice files* produced by a previous well-performing adaptive run (best of five). The advice files specify (1) the optimization level for compiling each method, (2) the dynamic call graph profile, and (3) the edge profile. Fixing these inputs, we execute two consecutive iterations of the application. During the first iteration, Jikes RVM optimizes code using the advice files. The second iteration executes only the application with a realistic mix of optimized code.

We execute each benchmark in a high-performance configuration: we use a heap size fixed at three times the minimum necessary for that benchmark to run using a generational mark-sweep garbage collector. For replay compilation, we report the minimum run-time of five trials since it represents the run least perturbed by external effects. For adaptive runs, we perform 25 trials because of high variability and report the median to discount outliers.

Benchmarks We evaluate the performance of origin tracking using the DaCapo benchmarks (version 2006-10), SPEC JVM98, and a fixed-workload version of SPEC JBB2000 called `pseudojbb` [8, 33, 34]. We omit `xalan` from replay experiments because we could not get it to run correctly with replay compilation, with or without origin tracking.

We perturb the methodology of several replay experiments that otherwise execute incorrectly; the determinism of replay means exposed VM bugs often occur consistently. We execute `pmd` with 3.1 times the minimum heap instead of 3.0 because 3.0 exposes a VM bug. For the same reason, we execute `chart` with a generational copying collector and `pmd` with a full-heap mark-sweep collector (these benchmarks would not execute correctly in any heap size with a generational mark-sweep collector).

Platform We perform our experiments on a 3.6 GHz Pentium 4 with a 64-byte L1 and L2 cache line size, a 16KB 8-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, a 2MB unified 8-way set associative L2 on-chip cache, and 2GB main memory, running Linux 2.6.12.

4.2 Space Overhead

Except for one ID per application method, origin tracking adds no space overhead because it uses value piggybacking to store program locations *in place* of null references.

4.3 Execution Time Overhead

Figure 4 shows the application execution overhead of origin tracking. We report the second run of replay methodology, during which only the application executes. We present several configurations that represent various origin tracking functionality levels (presented in order of monotonically increasing functionality):

- *Base* is application execution time. All bars are normalized to *Base*.
- *Simple checks* includes all origin tracking functionality except for two key but relatively costly components: (1) instrumentation to support general reference comparisons and (2) initialization of nulls to program locations at object allocation time. *Simple checks* adds 1% overhead on average.
- *Ref compare* adds support for general reference comparisons to *Simple checks*. *Ref compare* adds 1% over *Simple checks*, for a total of about 2% on average.
- *Origin tracking* adds initialization of nulls at object allocation time to *Ref compare*, and this configuration contains all functionality needed to track and report origins for null pointer exceptions. *Origin tracking* adds 1% over *Ref compare*, for a total of 3% on average.

4.4 Compilation Overhead

Origin tracking increases compilation time because it adds instrumentation to the application that redefines reference comparison and sets null references at object allocation. Figure 5 shows the overhead origin tracking adds to compilation for the same configurations as in Figure 4. We determine compilation time by measuring time spent in the compiler during the first run of replay compilation, which compiles

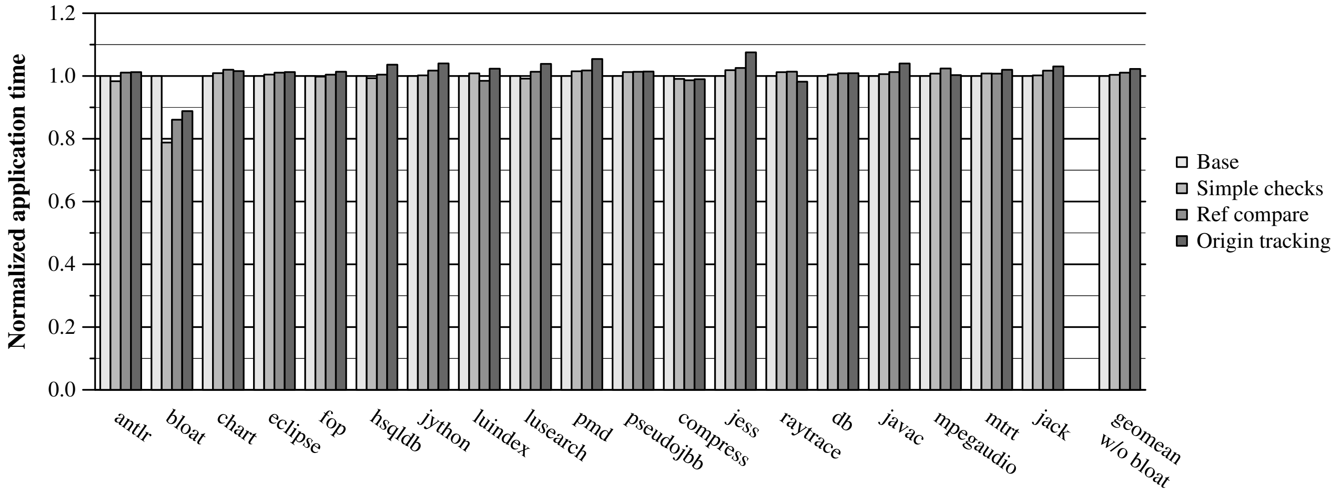

Figure 4. Application execution time overhead of origin tracking.

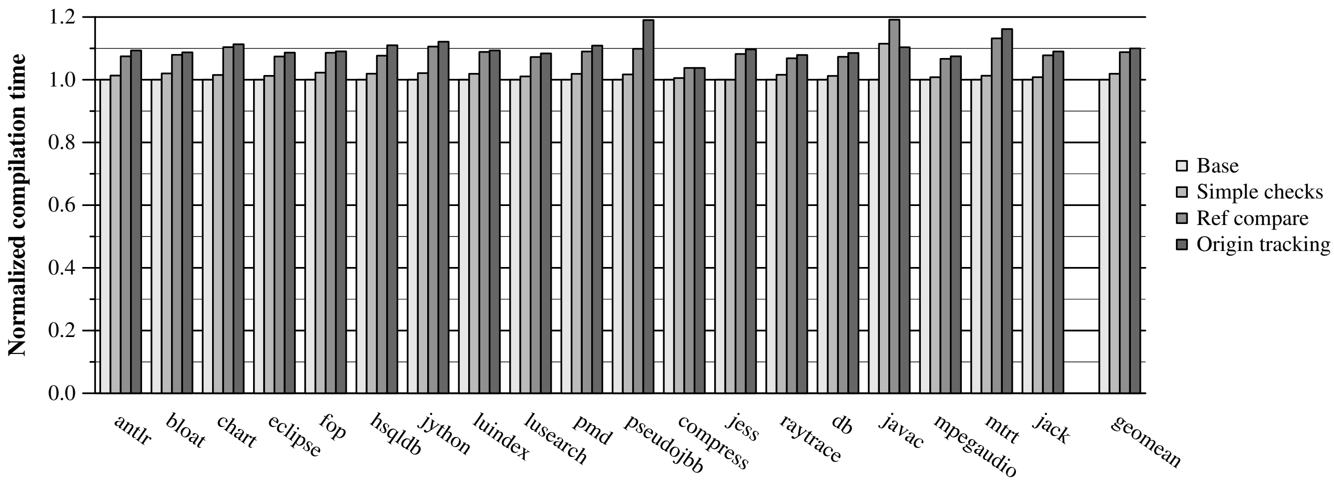

Figure 5. Compilation time overhead of origin tracking.

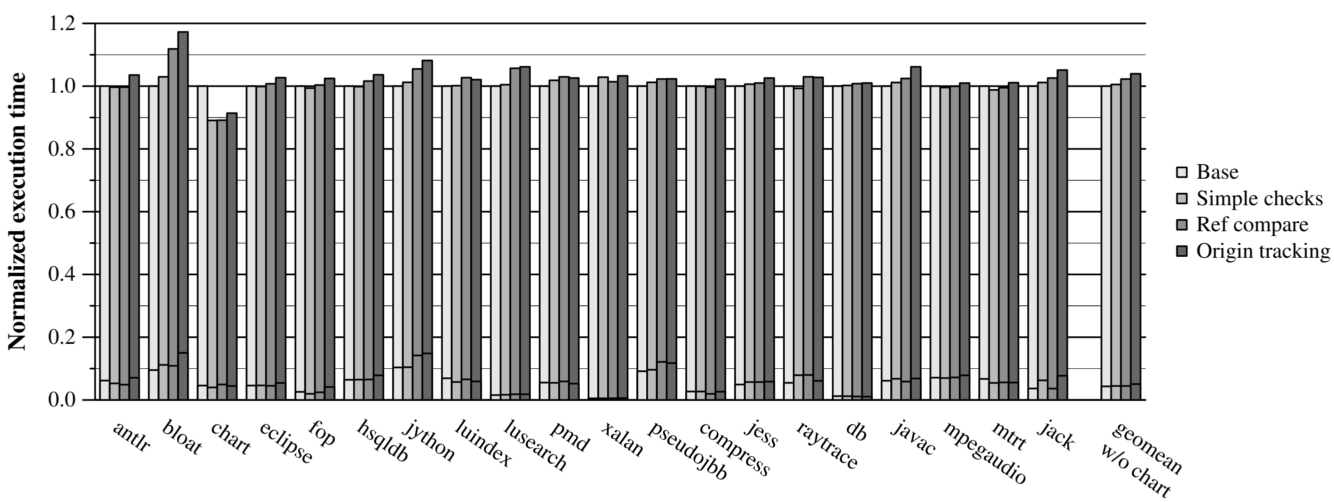

Figure 6. Adaptive methodology performance of origin tracking. The tall bars are overall execution time, while the sub-bars are the fraction of time spent in compilation.

and executes the application. Origin tracking adds 10% compilation overhead on average and 19% at most (`pseudojbb`). Compilation time, however, is a very small fraction of overall execution time (see below).

4.5 Overall Run-Time Overhead

Finally, we use adaptive methodology to measure the overall performance effect of origin tracking. Figure 6 shows combined application and compilation overhead for the same origin tracking configurations as in Figures 4 and 5. The bars are overall execution time, and the sub-bars are the fraction of time spent compiling. On average, origin tracking adds 4% to overall execution time. Compilation time is a small fraction of overall time: it increases from 0.043 in *Base* to 0.051 in *Origin tracking*. Compilation time varies considerably for single benchmarks, presumably because of the interplay between compilation and application execution.

5. Undefined Variables in C & C++ Programs

In this section, we present another application of value piggybacking: tracking the origins of undefined values in C, C++, and Fortran programs. We first describe *Memcheck* [31], a tool built with Valgrind [26] that validates memory usage in programs. We then describe how we modify Memcheck to piggyback origins on undefined values.

Our results show that origin tracking adds only negligible overhead to Memcheck, although Memcheck alone slows down execution on average by a factor of 28. Memcheck reports 147 undefined value warnings on our test suite of 20 C, and C++, and Fortran programs. Of these warnings, 47 are for 32-bit values (in which it is possible to fit a program location), and our enhanced MemCheck reports the origin for 34 (72%) of these warnings. Memcheck cannot report origins for the remaining warnings since they involve values smaller than 32 bits, so they cannot fit a program location.

Our version of Memcheck is publicly available as a branch in the Valgrind source repository:

```
svn://svn.valgrind.org/valgrind/branches/ORIGIN_TRACKING
```

5.1 Memcheck

Memcheck [31] is implemented in Valgrind [26], a framework for heavyweight dynamic binary instrumentation. Memcheck works on programs written in any language, but it is most useful for those written in C and C++ in which memory errors are common. Memcheck detects memory-related defects such as bad or repeated frees of heap blocks (blocks allocated with malloc, new, or new[]), memory leaks, heap buffer overflows, and wild reads and writes. It detects these defects by recording two kinds of metadata: (a) heap block information such as size, location, and a stack trace for its allocation point, and (b) a single *A bit* ('A' for "addressability") per memory byte, which indicates if the byte is legally accessible.

Memcheck detects dangerous uses of undefined values by shadowing every register and memory byte with 8 *V bits* ('V' for "validity") that indicate if the value bits are defined (i.e., initialized, or derived from other defined values). It updates and propagates these bits through memory and registers in parallel with normal computation. To minimize false positives Memcheck only warns users about uses of undefined values that can change the program's behavior. These uses fall into four cases.

1. The program uses an undefined value in a conditional branch or conditional move, potentially changing the program's control flow.
2. The program uses an undefined value as a branch target address, potentially changing the program's control flow.
3. The program provides an undefined value as an input to a system call, potentially changing the program's side effects.
4. The program uses an undefined value as the address in a load or store, potentially causing an incorrect value to be used, which could subsequently change the program's control flow or side effects.

In contrast, Memcheck does not warn about benign uses, such as copying or operating on undefined values, because these operations are not inherently erroneous and are very common. For example, programs copy not-yet-initialized fields in C structs and copy empty bytes in structs due to padding between fields. This delayed reporting avoids many false-positive warnings. For example, the simplest "hello world" program causes hundreds of false positive errors with eager reporting.

For undefined inputs to system calls, Memcheck already gives useful origin information since the value is usually in a memory block; for example, the undefined value may be in a filename string argument. However, for undefined conditions, load/store addresses, and branch target address, the undefined value is in a register, and Memcheck has no useful origin information for it. Users of Memcheck complain about this case enough that it warranted an entry in Valgrind's frequently asked questions (FAQ) file (starting in version 3.2.2) [26].

5.2 Memcheck Implementation Details

This section describes how we modify Memcheck to piggyback origins on undefined values. Since the values are undefined, we can assign them whatever value we choose. First we present how our implementation encodes program locations, then how it redefines program operations to accommodate piggybacked undefined values, and finally how it reports origins when an undefined value is used in a potentially harmful way.

Representing Origins Memcheck stores *context-sensitive* origins for heap-allocated variables and *context-insensitive* origins for stack-allocated variables.

Memcheck already computes and stores a stack trace for each heap block at allocation. The stack trace includes the allocation site and its dynamic calling context. Memcheck stores stack traces in a calling context tree [4] and uses the 32-bit address of this data structure as the *origin key*, i.e., a value from which it can identify an origin. Our modified Memcheck paints each newly allocated heap block with repeated copies of the origin key. The exception is heap blocks allocated with `calloc`, which must be initialized with zeroes.

Memcheck takes a similar but lighter-weight approach to undefined values allocated on the *stack*. Because stack allocations are so frequent, recording a stack trace for every one would be expensive. Instead, Memcheck records a static (context-insensitive) code location for each stack allocation, which it it determines at instrumentation time for efficiency. When the stack allocation occurs, our modified implementation paints the memory block with the appropriate origin key.

Modifying Program Operations No changes are required for copying or for operations that involve normal program values. Nor are any changes required to Memcheck's shadow values operations. Thanks to value piggybacking, the origin values get propagated for free. Unlike the Java implementation, origins can be lost if certain operations modify undefined values. For example, if an undefined value holding an origin key is added to a nonzero defined value, the resulting value will be undefined, but its origin information has been lost.

Reporting Origins To report the origins of undefined value warnings, we modify Memcheck at undefined value warning checks to read the origin key from the undefined value and look up the corresponding program location. If the origin key exists, Memcheck reports the location as the likely origin for the undefined value. Conditional branches and moves complicate identifying the origin key since their input is a single-bit condition. We need to search backwards to find one or more 32-bit values from which the undefined condition bit was derived. At instrumentation time, our implementation performs a backwards dataflow trace to find all 32-bit values that are ancestors of the condition bit. For example, if the program compares two 32-bit values and then uses the result as a condition, then each of the values is reported as a possible origin if it is undefined at run time (determined by examining the V bits when the warning is issued). The backwards searching is limited because Valgrind instruments code in small chunks (superblocks) that are usually 3–50 instructions in length. This limitation can degrade the accuracy of the origin report.

Figure 7 shows an example undefined value warning that includes an origin. The defect involves allocating a heap block in `main()`, passing an uninitialized 32-bit value to a function `f()`, which then passes it to `g()`, which then compares the value to another in an if-then-else. The first half of the warning is what is printed by Memcheck without origin tracking. The second half of the warning is added by origin tracking: it identifies the origin—the original heap allocation point.

```
Conditional jump or move depends on uninitialised
 value(s)
   at 0x8048439: g (a.c:20)
   by 0x8048464: f (a.c:25)
   by 0x80484A5: main (a.c:34)
 Uninitialised value has possible origin 1
   at 0x40045D5: malloc (vg_replace_malloc.c:207)
   by 0x804848F: main (a.c:32)
```

Figure 7. A Memcheck undefined value warning involving an origin.

5.3 Discussion

We discuss here some limitations of origin tracking due to Memcheck and the C and C++ programming model, potential solutions, and related topics.

Missed Origins Due to Small Values The biggest limitation of origin tracking in Memcheck is that it cannot track program locations well in fewer than 32 bits, and therefore it does not report origins for undefined value defects involving 8- or 16-bit data. Sixteen bits is just not enough to store a code location for even moderately large programs. This problem does not affect the Java implementation because Java null pointers are always word-sized.

To address this problem, we tried performing "partial matching": reporting every origin for which the 8 or 16 bits in the undefined value matched an 8- or 16-bit fragment of an origin key. But this approach resulted in many incorrect matches, particularly in the 8-bit case. Unfortunately, the 8-bit and 16-bit cases occur often (Section 5.4).

Another possible approach is to store less precise information, such as a file or method identifier in these small values. But this would result in less information for the 32-bit case as well—we cannot tell ahead of time which parts of a memory block will be used as 32-bit values and which as 8- or 16-bit values. Alternatively, it would be possible to execute the program again with information from the first run to narrow down the number of matches, although this would be less convenient for programmers. A complete solution would store the origins separately, as is done for the V bits, rather than use piggybacking, but this solution requires extra space and time, losing the advantages of piggybacking.

Missed Origins Due to Other Reasons Origins can fail to be identified in the 32-bit case for two reasons. First, if an undefined value is modified, it will no longer match its origin key. We could prevent modifications of origin values, but the extra checks required for almost every operation would be expensive. We tried unsuccessfully to perform "fuzzier matching": requiring that only three of the four bytes in the undefined value matched the origin key. Second, if an unmodified, undefined 32-bit value is loaded via an unaligned load, the undefined value will not match a key because the bytes will be out of order. Again, fuzzier matching could help: if there are no matches, Memcheck could try rotating the bytes of the value before matching. We find that unaligned accesses are not common enough to warrant this addition.

Incorrect Origins There is a small chance that a 32-bit undefined value that has been modified may match a different origin key, giving the wrong origin. For this reason we always describe an origin as a "possible origin" in the warning messages.

Changing Program Behavior Assigning origin keys to undefined memory, instead of the often fortuitously found zero, may change a program's *behavior*. However, this is acceptable as it does not change the program's *semantics*—the changes are within an envelope of behavior that is already undefined. Also, in this setting, we assume the programmer is trying to identify defects, and is fixing defects in the order Memcheck identifies them. Execution becomes increasingly unreliable as more errors occur.

64-Bit Machines The technique extends simply to 64-bit machines. It is worthwhile to keep the origin keys 32-bits, because integers on 64-bit machines are 32-bits. We can either use partial matching of 32-bit undefined values against 64-bit origin keys, or just use 32-bit origin keys (e.g., associate a random 32-bit key with each stack trace instead of using the address of the stack trace data structure).

5.4 Accuracy of Origin Tracking in Memcheck

To determine the accuracy of Memcheck's origin tracking, we found 20 C, C++, and Fortran programs for which the unaltered version of Memcheck issues at least one undefined value warning. Three of these (facerec, parser, twolf) are from the SPEC CPU2000 suite. The other 17 were found just by trying programs present on a Linux machine more or less at random. Undefined value errors are common enough that we found the 17 programs in about two hours.

Table 3 summarizes the results. For these programs, the unaltered Memcheck issues 147 undefined value warnings that lack origin information. Memcheck does not reissue warnings that look similar to previous ones in order to avoid uninteresting duplication of warnings. Nonetheless, it is possible that a single undefined value defect can result in more than one warning. Of the 147 undefined value warnings issued by Memcheck, 100 involve 8-bit or 16-bit undefined values, for which our technique cannot identify origins. Two programs, ps2pdf and xpdf, account for 57 of these small value warnings. We suspect that many of these are 8-bit warnings related to defects involving strings.

Program (static instrs)	Total	32-bit values Origin	32-bit values None	<32-bit None
dvips (42K)	1	0	0	1
facerec (38K)	1	1	0	0
firefox (1770K)	6	0	0	6
glibc (12K)	8	8	0	0
ispell (82K)	1	1	0	0
kanagram (401K)	8	0	1	7
kbounce (453K)	14	0	5	9
kpdf (612K)	1	1	0	0
ooffice (2967K)	1	0	0	1
parser (41K)	2	0	0	2
pdf2ps (166K)	13	5	0	8
ps2ascii (231K)	4	4	0	0
ps2pdf (150K)	24	1	0	23
pstree (19K)	3	0	0	3
python (81K)	13	10	3	0
twolf (52K)	2	2	0	0
vim (76K)	2	0	0	2
xfig (135K)	4	0	2	2
xfontsel (90K)	2	0	0	2
xpdf (230K)	37	1	2	34
Total	147	34	13	100

Table 3. Undefined value warnings and Memcheck's success at identifying their origins.

Of the 47 undefined value warnings involving 32-bit values for which Memcheck could possibly report origins, it reports 34 (72%). The 13 cases where it failed to identify an origin must involve unaligned memory or modified undefined values, as discussed in Section 5.3. We tried some partial (24-bit) and fuzzy (rotated) value matching in an attempt to identify more origins, but only a single extra origin was identified, for python.

We have not evaluated the usefulness of the Memcheck origin reports since we focused in this paper on the Java implementation, but we suspect undefined value origins will help programmers since they tell programmers something they would otherwise need to figure out manually.

5.5 Overhead of Origin Tracking in Memcheck

To measure the overhead of origin tracking in Memcheck, we run SPEC CPU2000 benchmarks (except for galgel, which gfortran failed to compile). We use the training inputs, which are smaller than the reference inputs, but the experimental runs took more than 24 hours with training inputs alone, and we believe the results would not give noticeably different results in this case. Prior, unrelated experiments with Memcheck also found that using the larger inputs made made minimal differences to timing results. The test machine was a 2.6GHz Pentium 4 (Northwood) with 2GB RAM and a 512KB L2 cache. We implemented origin tracking in a pre-3.3.0 development version of Memcheck.

Figure 8 shows the results of measuring Memcheck with and without origin tracking. Memcheck alone slows pro-

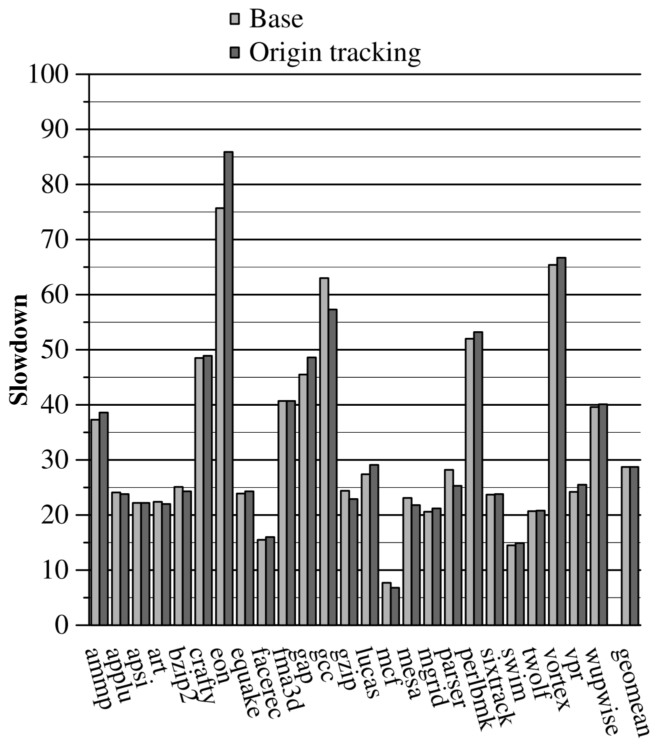

Figure 8. Memcheck's slowdown without and with origin tracking.

grams down by a factor of 28x (comparable to prior results [31]). The main cost of origin tracking is the painting of heap and stack blocks with the origin keys. However, the overall performance impact of origin tracking is negligible. At worst, Memcheck with origin tracking worsens the slowdown factor for eon from 75.7x to 85.9x, and at best it improves the slow-down factor for gcc from 63.0x to 57.3x. This level of variation is likely due to factors such as different cache behavior caused by the extra writes to uninitialized memory blocks.

6. Related Work

Previous work related to origin tracking and unusable values can be divided into three categories: (a) dynamic approaches that help diagnose bugs when they occur, (b) static approaches that detect bugs before the program is run, and (c) language design approaches which reduce or eliminate the chance of such bugs occurring altogether.

Dynamic Approaches Origin tracking can be considered a special case of *dynamic program slicing* [1, 25]. Dynamic slices include most or all statements that affect a value via control or data-dependence. Dynamic slicing provides information for any value, not just unusable values, and dynamic slicing provides all statements affecting a value, not just the value's origin. However, dynamic slicing is very expensive, e.g., 10-100X slowdowns are typical, while origin tracking uses value piggybacking to make it efficient enough for deployed software.

TraceBack records a fixed amount of control flow (similar to dynamic slicing but without dataflow) during execution and reports it in the event of a crash [6]. Traceback provides control flow information leading up to the fault, while origin tracking provides the data-flow origin of the faulting value.

Purify and Memcheck detects a similar class of errors. Purify eagerly reports some undefined value as soon as they are loaded from memory. These operations are closer to the undefined values' origins, but eager reporting can cause false positives. Although, Purify prunes these reports by using static analysis to identify undefined values that definitely do not cause errors. Memcheck instead delays reporting of undefined values to the point where they change program behavior, which yields fewer false positives and together with origin tracking is more accurate.

TaintCheck is a security tool that tracks which values are tainted (i.e., from untrusted sources), and detects if they are used in dangerous ways [27]. TaintCheck shadows each byte, recording for each tainted value: the system call from which it originated, a stack trace, and the original tainted value. Thus TaintCheck uses a form of explicit origin tracking that requires both extra space and time (extra operations are required to propagate the taint values). Value piggybacking would not be appropriate for TaintCheck because tainted values cannot have other values piggybacked onto them as they do not have spare bits.

Saber is a memory-checking tool that stores a special one bit *canary value* in undefined values [20], which indicates *undefinedness*. Our Memcheck implementation instead notes defined/undefined separately in its V bits and stores more complex information (origins) in the undefined values.

Zhang et al. improve dynamic slicing by identifying *omission errors*, statements that lead to an error because they did *not* execute [36]. Some undefined value errors are the result of omission errors, but our origin tracking approach reports only statements that executed, rather than statements that should have executed but did not.

Recent novel work on anomaly-based bug detection uses multiple program runs to identify program behavior features correlated with errors [13, 21, 23, 37]. Origin tracking is complementary to these approaches since they detect the causes of many types of bugs besides unusable value errors, while origin tracking may be able to find bug causes not detected by invariant violations. Anomaly-based bug detection adds overhead too high for deployed use [13] or requires multiple runs to detect bug causes [21, 23, 37], whereas our Java implementation of origin tracking works efficiently in a single deployed run.

An alternative to detecting bugs is to tolerate them automatically at run time [7, 29, 30]. For example, *Rx* rolls back to a previous state and tries to re-execute in a different environment [29]. *DieHard* uses random memory allocation, padding, and redundancy to probabilistically decrease the chances of errors [7]. These approaches are most suitable for pointer and memory corruption errors, rather than semantic errors such as undefined values.

Static Analysis Previous bug detection work includes a number of static analysis algorithms for detecting bugs. Pattern-based systems such as PMD are effective at identifying potential null dereferences but lack the dataflow analysis often needed to identify the reason for the bug [28]. FindBugs uses dataflow analysis to identify null dereferences and includes a notion of confidence to reduce the false positive rate [14, 15]. ESC/Java uses a theorem prover to check that pointer dereferences are non-null [11]. Both FindBugs and ESC/Java are primarily intraprocedural, and rely on user annotations to eliminate false positives due to method parameters and return values. JLint and Metal include an interprocedural component to track the states of input parameters [12, 18].

The advantage of static analysis is that it detects bugs without having to execute the buggy code. Unfortunately, static tools suffer from two significant limitations. First, they often produce many false positives because they rely on coarse approximations of dynamic program behavior, since context and flow-sensitive analysis is too expensive for large programs. Second, few if any build a model of the heap precise enough to track null values through loads and stores. In contrast, our origin tracking approach reports information only for errors that occur, and tracks the origin through arbitrarily long and complex code sequences, including loads and stores to the heap, without losing precision.

Several static bug detectors, including PMD, FindBugs, and Metal, are made intentionally *unsound* to reduce the false positive rate. This choice, however, allows code with bugs to pass silently through the system and fail at run-time. Origin tracking complements these systems: it can diagnose the more complex bugs that they miss.

Language Design The unusable value bugs discussed in this paper are a consequence of two particular language features: (a) pointer values are overloaded to represent a pointer to *something* (non-null) or a pointer to *nothing* (null), and (b) the declaration of a variable is separate from its initialization and the assignment can be easily forgotten. An alternative to detecting these bugs is to prevent them from happening by choosing a language that does not have these features.

Java, Chalin and James [9] propose extending Java with "never null" pointer types, which are the default, and requiring "possibly null" pointers to be annotated specially. This feature makes it harder to forget to initialize pointers, but as long as null pointers are possible, the problem can still occur.

Functional languages avoid both problems. First, variables are only introduced when they are bound to values, so it is impossible for a variable to be uninitialized. Second, they use explicit types to represent "nothing," allowing the type checker to make sure that programs handle these cases. However, it is still possible for variables to have unexpected values, causing a program to fail at run-time.

7. Conclusions

Developers need all the help they can get when debugging. We present a lightweight approach for tracking the origins of null pointers in Java programs and undefined values in Memcheck. The key to origin tracking's efficiency is that program locations are stored *in place* of null and undefined values, avoiding space overhead and significant time overhead since the locations propagate via normal program data flow. The Memcheck implementation of origin tracking adds no overhead, on average, to provide origin information at testing time, and finds origins for 72% of the 32-bit undefined value errors. The Java implementation adds just 4% to overall execution time. Origins are useful for 7 of 12 real bugs, and the toughest bugs are those helped most by origin knowledge. Given its minimal footprint and productivity benefits, origin tracking is ideal for commercial VMs where it can enhance debugging in existing and future deployed software.

Acknowledgments

We would like to thank Julian Seward for many helpful discussions about the Memcheck implementation; Jason Davis and Ben Wiedermann for testing out our implementation of origin tracking; Ben Wiedermann for help understanding the Jython source; and Simha Sethumadhavan, Julian Seward, and the anonymous reviewers for valuable feedback about the paper text.

Appendix

This appendix describes the exceptions from Table 2 not covered in Section 3.

Case 4: JRefactory: Invalid Class Name This case shows how origin tracking helps diagnose an improperly initialized reference that is used later and elsewhere in the program. We triggered a previously unknown failure in JRefactory 2.9.18 by accident while trying to reproduce Case 11's failure. Perhaps due to pre-submission fatigue, one of us wrote the following class declaration (it should be just `Bug`, not `Bug.java`):

```
public class Bug.java {
```

Figure 9(a) shows the stack trace produced when JRefactory processes a class containing the incorrect class declaration. Inspection of the code at the point of the exception shows that JRefactory correctly detects the invalid class name and attempts to print a useful error message. However, the error message code fails because it dereferences the return value of `ExceptionPrinter.getInstance()`, which returns `ExceptionPrinter.singleton`, which is null.

```
java.lang.NullPointerException:
  at net.sourceforge.jrefactory.factory.ParserFactory.
        getAbstractSyntaxTree():46
  at org.acm.seguin.pretty.PrettyPrintFile.apply():102
  at org.acm.seguin.tools.builder.PrettyPrinter.
        visit():77
  at org.acm.seguin.io.DirectoryTreeTraversal.
        traverse():91
  at org.acm.seguin.io.DirectoryTreeTraversal.run():43
  ...
  at PrettyPrinter.main():54
```

(a)

```
Origin:
  org.acm.seguin.awt.ExceptionPrinter.<clinit>():70
```

(b)

Figure 9. Case 4: VM output for a JRefactory bug. (a) The stack trace alone shows that a parse error was encountered. (b) Origin tracking shows that the error reporting data structure was not properly initialized.

```
java.lang.NullPointerException:
  at gnu.xml.dom.DomDocument.checkNewChild():315
  at gnu.xml.dom.DomDocument.appendChild():341
  at org.eclipse.pde.internal.builders.XMLErrorReporter.
        endDocument():159
  at gnu.xml.stream.SAXParser.parse():669
  at javax.xml.parsers.SAXParser.parse():273
  ...
  at org.eclipse.core.internal.jobs.Worker.run():76
```

(a)

```
Origin: org.eclipse.pde.internal.builders.
            ManifestConsistencyChecker.checkFile():76
```

(b)

Figure 10. Case 5: VM output for an Eclipse bug. (a) The stack trace alone indicates a parsing error. (b) Origin tracking identifies the source of the exception.

Figure 9(b) shows the origin: the class initializer sets `ExceptionPrinter.singleton` to null, and it is not modified after that. On inspection, we found that the other methods in `ExceptionPrinter` automatically initialize the `singleton` field whenever it is null, and we believe this behavior is needed in the `getInstance()` method as well. We submitted this bug and the suggested fix to JRefactory's bug tracker (Bug 1674321), but as of the camera-ready copy there has been no response.

Case 5: Eclipse #1: Malformed XML Document The Eclipse integrated development environment (IDE) version 3.2 [10] can fail when a user provides an improper XML document specifying a plugin project's extensions. If the XML document is malformed and contains no root element, Eclipse throws a null pointer exception when attempting to parse the document. While the stack trace indicates that the failure occurred during parsing, the origin information expresses that the XML documents lacks a root element.

Figure 10(a) shows the stack trace produced by this null pointer exception. Without origin tracking, determining the cause using the stack trace alone would be quite difficult, since the null value is an input parameter to the method. Further investigation with the debugger would involve following the value back through a series of method calls.

The origin tracking information, shown in Figure 10(b), reveals exactly what is wrong with the XML file. The null value originates in the code that checks the consistency of the file. The specific origin location indicates that the XML `fRootElement`'s value is initialized to null and never set to any other value. This value is then passed to `endDocument`, and on to `appendChild` and finally to `checkNewChild`.

We reported this bug to Eclipse developers (Bug 176500). Developers determined the bug was in the underlying XML parser, which is separate from Eclipse, and they were unable to reproduce the bug in the latest version of Eclipse (3.3). Given that the origin shows that `fRootElement` was never initialized, we believe this report would most likely help fix a bug in the XML parser.

Case 6: Checkstyle: Empty Default Case Checkstyle checks Java source code for compliance to a coding standard. Checkstyle's bug tracker contains a bug report describing how to reproduce a null pointer exception in Version 4.2 (Bug 1472228). The exception occurs when Checkstyle processes code where the `default` case has no statements. We reproduced this bug by providing a class to Checkstyle with the following `switch` statement:

```
switch(x) {
  case 0:
    test = true;
    break;
  default:
}
```

Without origin tracking, Figure 11(a) shows the resulting exception stack trace. We show code with line numbers from `FallThroughCheck.checkSlist()`:

```
195:  DetailAST lastStmt = aAST.getLastChild();
196:
197:  if (lastStmt.getType() == TokenTypes.RCURLY) {
198:    lastStmt = lastStmt.getPreviousSibling();
199:  }
200:
201:  return (lastStmt != null) &&
202:    isTerminated(lastStmt, aUseBreak, aUseContinue);
```

The exception occurs because `lastStmt` is null at line 197. However, it is not clear what that null value signifies, or whether it is safe to simply skip processing `lastStmt` in the case that it is null.

The extra information provided by origin tracking, shown in Figure 11(b), helps to answer this question. Origin tracking shows that the null value originates in the Antlr parser component, meaning that the value is set to null during pars-

```
java.lang.NullPointerException:
  at com.puppycrawl.tools.checkstyle.checks.coding.
        FallThroughCheck.checkSlist():197
  at com.puppycrawl.tools.checkstyle.checks.coding.
        FallThroughCheck.isTerminated():168
  at com.puppycrawl.tools.checkstyle.checks.coding.
        FallThroughCheck.visitToken():136
  at com.puppycrawl.tools.checkstyle.TreeWalker.
        notifyVisit():500
  at com.puppycrawl.tools.checkstyle.TreeWalker.
        processIter():625
  ...
  at com.puppycrawl.tools.checkstyle.Main.main():127
```

(a)

```
Origin: antlr.ASTFactory.make():323
```

(b)

Figure 11. Case 6: VM output for Checkstyle bug. (a) Stack trace indicates an error checking a fall-through case. (b) The origin shows the null value originated when constructing the program AST.

```
java.lang.NullPointerException:
  at jode.decompiler.ClassAnalyzer.<init>():96
  at jode.decompiler.ClassAnalyzer.initialize():220
  at jode.decompiler.ClassAnalyzer.dumpJavaFile():620
  at jode.decompiler.ClassAnalyzer.dumpJavaFile():613
  at jode.decompiler.Main.decompileClass():184
  at jode.decompiler.Main.decompile():376
  at jode.decompiler.Main.main():203
```

(a)

```
Origin: jode.bytecode.ClassInfo.forName():157
```

(b)

Figure 12. Case 7: VM output for JODE bug. (a) The stack trace alone indicates that an inner class has a null pointer to its outer class. (b) Origin tracking tells us that the outer class may never have been initialized.

```
java.lang.NullPointerException:
  at org.python.core.ListFunctions.__call__():48
  at org.python.core.PyObject.invoke():2105
  at org.python.pycode._pyx8.f$0():<console>
  at org.python.pycode._pyx8.call_function():<console>
  at org.python.core.PyTableCode.call():155
  ...
  at org.python.util.jython.main():178
```

(a)

```
Origin: org.python.core.PyClass.__findattr__():178
```

(b)

Figure 13. Case 8: VM output for a Jython bug. (a) Exception output provided by the VM. (b) Extra information provided by origin tracking.

ing rather than in some (possibly erroneous) part of the Checkstyle code itself. Thus the origin should help developers determine that the null value is not erroneous and simply indicates a case with no statements. A response to the bug report notes that adding a null check fixes the bug, and developers implemented the fix in Checkstyle 4.3.

Case 7: JODE: Exception Decompiling Class Java Optimize and Decompile Environment (JODE) is a package that includes a decompiler and optimizer for Java. We reproduced a bug in version 1.1.1 that occurs when trying to decompile a particular class file (Bug 821212). While the stack trace identifies the immediate cause of the failure, the origin information shows that an important logical dependence (between inner and outer classes) is not being maintained properly.

Figure 12(a) shows the stack trace produced by this failure. The exception occurs during initialization of an inner class because a pointer to information about the outer class is null. With the stack trace and the source code, we were unable to understand why this pointer was null. Furthermore, we could not find source for the class to be decompiled (provided in the bug report), which is not surprising since the user reporting the bug wanted to decompile it!

The reported origin, shown in Figure 12(b), indicates that the pointer to the outer class is null at the allocation site for the inner class information object. The field is initialized to null and never set anywhere else, leading us to believe that the outer class may not have been loaded and initialized yet. The origin information is likely to be useful to developers, since it reveals a problem with the logic involved in JODE's class loading. We added the information to the official bug report. However, the bug remains unfixed.

Case 8: Jython: Use Built-In Class as Variable Jython is an implementation of the Python language integrated with Java. We reproduce an exception from an entry on the *jython-dev* mailing list archive, dated September 17, 2001 [19], that applies to Jython 2.0. The exception occurs when the user uses a built-in method as a class variable, which should be a valid operation.

Figure 13 shows the exception stack trace for this bug, and the extra information origin tracking provides. We examined the source code and found that the source of the null reference is far removed from the null pointer exception. Thus, the information provided by origin tracking is nontrivial.

We were unable to understand this bug well enough to fix it or to determine if the information provided by origin tracking is useful. A Jython expert examined the source and determined that the implementation was not advanced enough to handle using a built-in class as a variable [35]. The bug is fixed in Jython 2.2 (we could not reproduce the bug in 2.2), although we could not find any record of the fix, and we were unable to determine the fix by inspection because of the large differences between the two versions.

```
java.lang.NullPointerException:
  at org.jfree.chart.renderer.xy.StackedXYAreaRenderer.
        drawItem():457
  at Bug.test():52
  at Bug.main():20
```

(a)

```
Origin:
  org.jfree.chart.renderer.xy.StackedXYAreaRenderer$
      StackedXYAreaRendererState.<init>():138
```

(b)

Figure 14. Case 9: VM output for a JFreeChart bug. (a) The exception stack trace. (b) The null reference's origin.

```
java.lang.NullPointerException:
  at org.python.core.PyObject.getDoc():360
  at java.lang.reflect.Method.invoke():147
  at org.python.core.PyGetSetDescr.__get__():55
  at org.python.core.PyObject.object___findattr__():2770
  at org.python.core.PyObject.__findattr__():1044
  ...
  at org.python.util.jython.main():214
```

(a)

```
Origin: org.python.core.PyObject.fastGetDict():2723
```

(b)

Figure 15. Case 10: VM output for a Jython bug. (a) Exception output provided by vanilla VM. (b) Extra information provided by origin tracking.

```
java.lang.NullPointerException:
  at org.acm.seguin.pretty.PrettyPrintVisitor.
        removeLastToken():1184
  at org.acm.seguin.pretty.PrettyPrintVisitor.visit():979
  at org.acm.seguin.pretty.PrettyPrintFile.apply():129
  at org.acm.seguin.pretty.PrettyPrintFile.apply():105
  at org.acm.seguin.tools.builder.PrettyPrinter.
        visit():77
  ...
  at PrettyPrinter.main():54
```

(a)

```
Origin: org.acm.seguin.pretty.PrettyPrintVisitor.
            removeLastToken():1177
```

(b)

Figure 16. Case 11: VM output for a JRefactory bug. (a) Exception output provided by vanilla VM. (b) Extra information provided by origin tracking.

Case 9: JFreeChart: Stacked XY Plot with Lines JFreeChart is an advanced library for displaying charts in Java applications. A null pointer exception occurs when an application attempts to generate a stacked XY plot that has lines enabled in Version 1.0.0 (Bug 1593156). Figure 14(a) shows the exception stack trace that occurs. The stack trace shows that the exception occurs when JFreeChart dereferences `StackedXYAreaRenderer.lines`, which is null.

Origin tracking reports the constructor for `StackedXYAreaRendered`, where `StackedXYAreaRenderer.lines` is initialized to null (Figure 14(b)). This information tells developers that `lines` is null because it is initialized to null at allocation time, rather than being assigned null later. However, since `lines` is private and is not assigned anywhere besides the constructor, it is fairly easy to reach this conclusion by inspection of the class alone. Developers fixed the bug in the next version of JFreeChart by initializing `lines` to a default `Line2D` object.

Case 10: Jython: Program Accessing __doc__ Attribute We found the second Jython bug on Jython's bug tracker (Bug 1462188). The null pointer exception occurs when trying to access the *__doc__* attribute of a dictionary object.

Figure 15 shows the exception output and the extra information origin tracking provides. Examining Jython's source, we find that `PyObject.fastGetDict()` is called from the line that causes the exception (`PyObject.java:360`). Thus, origin tracking's information is not very insightful because the null reference's source program location is not far removed from its dereference. However, since `PyObject.fastGetDict()` is a virtual method that is overridden in six different subclasses, origin tracking does narrow down the source of the null reference from seven possibilities to one. This information will save a developer time in understanding the bug, especially if the bug is not reproducible.

According to the bug report, developers fixed the bug simply by checking if the value returned by `fastGetDict()` is null. If so, `getDoc()` then returns rather than dereferencing the null pointer.

Case 11: JRefactory: Package and Import on Same Line JRefactory is a refactoring tool for Java. JRefactory 2.9.18 throws a null pointer exception when provided a Java source file with the package and import declarations on the same line (Bug 973332):

```
package edu.utexas; import java.io.*;
```

Figure 16 shows that the null's origin (line 1177) is just seven lines above the exception's occurrence (line 1184), and static inspection of the code reveals that the null pointer exception can only occur when line 1177 is the origin, so the information provided by origin tracking is trivial in this case. We were unable to understand this bug well enough to fix it, and no fix has yet been reported.

Case 12: Eclipse: Close Eclipse While Deleting Project If the user initiates a delete of a project with an open file and then immediately tries to close the Eclipse application before the delete is complete, Eclipse 3.1.2 throws a null pointer exception (Bug 142749). The exception occurs because Eclipse attempts to save the open file upon application

```
java.lang.NullPointerException:
  at org.eclipse.ui.internal.EditorManager$7.run():1323
  at org.eclipse.core.internal.runtime.InternalPlatform.
        run():1044
  at org.eclipse.core.runtime.Platform.run():783
  at org.eclipse.ui.internal.EditorManager.
        saveEditorState():1282
  at org.eclipse.ui.internal.EditorManager.
        saveState():1203
  ...
  at org.eclipse.core.launcher.Main.main():948
```

(a)

```
Origin: org.eclipse.core.internal.resources.Resource.
            getLocation():876
```

(b)

Figure 17. Case 12: VM output for an Eclipse bug. (a) Exception output provided by vanilla VM. (b) Extra information provided by origin tracking.

close, but the save uses a project root object that the delete operation has already nulled.

Using the stack trace and origin information shown in Figure 17, we examined the Eclipse source code and found that the origin of the null is *not* far removed from the null pointer exception: `EditorManager$7.run()` calls a method `FileEditorInput.getPath()`, which calls `Resource.getLocation()`, which returns a null constant because the project no longer exists. Thus, origin tracking provides trivial information that is not helpful for fixing this bug.

References

[1] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *ACM Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[2] E. Allen. Diagnosing Java Code: The Dangling Composite bug pattern. http://www-128.ibm.com/developerworks/-java/library/j-diag2/, 2001.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[4] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *ACM Conference on Programming Language Design and Implementation*, pages 85–96, Las Vegas, NV, 1997.

[5] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.

[6] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel. TraceBack: First Fault Diagnosis by Reconstruction of Distributed Control Flow. In *ACM Conference on Programming Language Design and Implementation*, pages 201–212, 2005.

[7] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *ACM Conference on Programming Language Design and Implementation*, pages 158–168, 2006.

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006.

[9] P. Chalin and P. James. Non-null references by default in java: Alleviating the nullity annotation burden. Technical Report 2006-003, Concordia University, 2006.

[10] Eclipse.org Home. http://www.eclipse.org/.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 234–245, 2002.

[12] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific, Static Analyses. In *ACM Conference on Programming Language Design and Implementation*, pages 69–82, 2002.

[13] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ACM International Conference on Software Engineering*, pages 291–301, 2002.

[14] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. In *Companion to ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 132–136, 2004.

[15] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 13–19.

[16] Jikes RVM. http://www.jikesrvm.org.

[17] Jikes RVM Research Archive. http://www.jikesrvm.org/-Research+Archive.

[18] Jlint. http://jlint.sourceforge.net.

[19] Jython-dev Mailing List. http://sourceforge.net/mailarchive/-forum.php?forum_id=5587.

[20] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: An Interpreter-Based Programming Environment for the C Language. In *Summer USENIX Conference*, pages 161–71, 1988.

[21] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *ACM Conference on Programming Language Design and Implementation*, pages 15–26, 2005.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, 1999.

[23] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[24] Mckoi SQL Database. http://www.mckoi.com/database/.

[25] N. Nethercote and A. Mycroft. Redux: A Dynamic Dataflow Tracer. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[26] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[27] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium*.

[28] PMD. http://pmd.sourceforge.net.

[29] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. In *ACM Symposium on Operating System Principles*, pages 235–248, 2005.

[30] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. Beebee. Enhancing Server Availability and Security through Failure-Oblivious Computing. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 303–316, 2004.

[31] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX Annual Technical Conference*, pages 17–30, 2005.

[32] SourceForge.net. http://www.sourceforge.net/.

[33] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, 1999.

[34] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.

[35] B. Wiedermann. Personal communication, November 2006.

[36] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards Locating Execution Omission Errors. In *ACM Conference on Programming Language Design and Implementation*, pages 415–424, 2007.

[37] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants. In *IEEE/ACM International Symposium on Microarchitecture*, pages 269–280, 2004.