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

UT Austin Technical Report TR-07-18. Submitted to OOPSLA 2007.

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

Despite extensive testing, deployed software still crashes. Other than a stack trace, these crashes offer little guidance to developers, making them hard to reproduce and fix. This work seeks to ease error correction by providing diagnostic information about the origins of null pointer exceptions and undefined variables. The key idea is to use *value piggybacking* to record and report useful debugging information in undefined memory. For example, instead of storing zero at a null store, store the *origin* program location, then correctly propagate this value through assignment statements and comparisons. If the program dereferences this value, report the origin. We describe, implement, and evaluate low-overhead value piggybacking for *origin tracking* of null pointer exceptions in deployed Java programs. We show that the reported origins add useful debugging information over a stack trace. We also describe, implement, and evaluate origin tracking of undefined values in C, C++, and Fortran programs in a memory error testing tool built with Valgrind. Together these implementations demonstrate that value piggybacking yields useful debugging information that can ease bug diagnosis and repair.

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

Despite advances in programming languages, analysis tools, and testing protocols, deployed programs often contain bugs. When programs do fail, today's run-time systems produce, at best, a stack trace identifying the point of failure. This information is often insufficient for a programmer to determine the underlying defect. For example, Liblit et al. report that examining methods in the stack trace does not identify the method containing the error for 50% of the bugs they examine [19]. We find similar results: out of 12 null pointer exceptions that we examine, the stack trace is only directly useful for locating four defects.

This paper focuses on a class of bugs due to *unusable values* that either directly cause a failure or result in unpredictable behavior. These bugs are common to both managed and unmanaged languages. In Java, the *null value* is unusable and causes a null pointer exception when dereferenced. In languages like C and C++, *uninitialized variables* are unusable because they are undefined. Their use can cause a variety of ill effects, including silent data corruption, altered control flow, or a segmentation fault. These failures are especially difficult to debug because (1) the origin of the value may be far from the point of failure, having been propagated through assignments, operations, and parameter passing, and (2) unusable values themselves (i.e., null, zero, or random bits) yield no useful debugging information. The key question for such errors is: why is the value unusable?

We solve this problem by introducing *origin tracking*, which tracks the *origin*, or assigning program location, of unusable values through the program, and at the time of

failure identifies the origin of the offending value. In many cases, this information is exactly what the programmer needs to diagnose the defect, and is particularly valuable for failures involving unusable values stored in the heap. Consider a crash that occurs when a pointer field in an object is unexpectedly null. We know the immediate cause of the crash: the reference is null. What we want to know is why. Origin tracking will tell us where the program originally stored null. The challenge of debugging null pointer exceptions, and the usefulness of origins, is recognized by industry practitioners. Eric Allen writes the following in his column *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.*

Origin information is equally useful for diagnosing uninitialized variables in C and C++. Memcheck [28] is a memory checking tool built with Valgrind [23] that can detect dangerous uses of undefined values, but gives no origin information about undefined values. Requests for such information from Memcheck users are common enough that the FAQ explains the reason for this shortcoming [23].

The key insight of this work is that we can add origin tracking to existing systems with no space overhead by storing origin information *in the unusable values themselves*. We call this technique *value piggybacking*. When the program creates an unusable value, i.e., each null reference in Java and each uninitialized variable in C and C++, value piggybacking instead stores the *origin*, i.e., the current program location. We add support to correctly compare these origin values. Origin values otherwise flow freely through the program, being copied, stored in the heap, or passed as parameters. They thus serve as normal values until the program attempts to use them. To summarize: *origin tracking* is *what* we do, and *value piggybacking* is the mechanism for *how* we do it.

We add origin tracking to Jikes RVM [3, 15], a high-performance virtual machine. Our Java implementation initializes each *tagged null* pointer at its origin point with an encoding of the method and line number. If the program attempts to dereference a null, the runtime system retrieves the method and line number from the tagged value and reports it to the programmer. For our C and C++ implementation, we modified Memcheck to store the program location in undefined memory when the program allocates it. Memcheck already identifies dangerous uses of undefined values, and our enhanced version reports the originating program location.

Our results show that value piggybacking adds minimally to execution time: 3% on average for Java programs, and is effective at reporting the origins of null pointer exceptions. It correctly reports the origin in 11 of the 12 Java null pointer bugs, and in 72% of the bugs involving 32-bit data in C, C++, and Fortran programs. We examine the 12 Java reports in depth to demonstrate the utility of these error messages. In eight of the 12, the origin was *not* due to a method in the stack trace, and in six cases, the origin is most likely or definitely useful for fixing the bug. The main contributions of this work are:

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

These applications are notable because previous value piggybacking conveyed only one or two bits of information per value (e.g., [18]), whereas we show it can convey more information. Since this approach requires only minor changes to the Java virtual machine, and incurs very low overhead, this technique could rapidly be put into wide use to help programmers find and fix bugs.

2. Related Work

Previous work on finding and diagnosing bugs can be roughly divided into two categories: static analysis, which attempts to identify bugs at compile time, and dynamic analysis, which tracks program behavior at run time. Origin tracking is a dynamic analysis for recording and propagating information about the program locations that generate bad values. This approach is complementary to software testing since it can help testers locate defects.

Dynamic Analysis. Origin tracking can be considered a special case of *dynamic program slicing*, which keeps track of dynamic data and control dependencies for each dynamic statement [1, 22]. Dynamic slices (1) provide all the statements leading to an exception or program point of interest, rather than just the origin statement, and (2) can track any value, not just null and uninitialized values. However, dynamic slicing suffers from high space and time overhead, e.g., 10-100X slowdowns are typical. Because origin tracking has well-defined start and end points, and because it does not record intermediate program locations, it adds no space overhead and only a few percent time overhead. It is thus

efficient enough to use all the time in a production environment.

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 [5]. Traceback provides control flow information leading up to the fault, while origin tracking provides the data flow origin of the faulting value. Traceback adds 59% overhead on average to SPEC CPU2000, while our Java implementation adds only 3%, making it efficient enough for deployed use.

Purify is a heavyweight debugging tool that identifies memory errors including undefined value errors [12]. It eagerly reports some loads of undefined value, which is closer to their source, but can cause false positives. 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 [24]. 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 extra space and time (shadow operations to propagate the taint values). Value piggybacking would not be appropriate for TaintCheck, because tainted values cannot have other values piggybacked onto them.

Saber stores special *canary value* in undefined values [18], which indicate *undefinedness*, a binary distinction. Our approach instead stores program locations and thus provides debugging guidance to developers.

Zhang et al. improve on dynamic slicing by identifying *omission errors*, statements that lead to an error because they did *not* execute [33]. Their work locates omission errors by flipping the predicate on a potentially relevant conditional branch, and re-executing. Some null pointer exceptions and uninitialized 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 [11, 19, 20, 34]. Origin tracking is complementary to these approaches, since it is not clear that invariant violations can diagnose all null pointer exceptions, which are very common [14]. In addition, anomaly-based bug detection adds overhead too high for deployed use [11] or requires multiple runs to detect bug causes [19, 20, 34], whereas our Java implementation of origin tracking works efficiently in a single deployed run.

A recent promising direction is to tolerate bugs automatically at run time [6, 26, 27]. For example, *Rx* rolls back to a previous state and tries to re-execute in a different environment [26]. *DieHard* uses random memory allocation, padding, and redundancy to probabilistically decrease the chances of errors [6]. These approaches are most suitable for pointer and memory corruption errors, rather than semantic errors such as null pointer exceptions and uninitialized 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 [25]. FindBugs uses dataflow analysis to identify null dereferences, and includes a notion of confidence to reduce the false positive rate [13, 14]. ESC/Java uses a theorem prover to check that pointer dereferences are non-null [9]. Both FindBugs and ESC/Java are primarily intraprocedural, however, relying 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 [10, 16]. The advantage of static analysis is that it detects bugs without having to execute the buggy code. When based on dataflow analysis, these tools can often identify the reason for the bug. 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.

3. Origin Tracking in Java

This section describes origin tracking, implemented via value piggybacking, for null references in Java. We modify the VM to use program locations, instead of the standard value zero, to represent null. To accommodate these nonzero nulls, the modified VM redefines operations such as null assignment, object allocation, and reference comparison. We first describe how to implement origin tracking in any VM, and then describe details specific to our Jikes RVM implementation.

3.1 Program Locations

Instead of the value zero, we modify the VM to use alternative values for nulls that encode program locations. To ensure correct program execution, we select values that dis-

	Java semantics	Standard VM	Origin tracking
(a) Assignment of null constant	`obj = null;`	`obj = 0;`	`obj = this_location;`
(b) Object allocation	`obj = new Object();`	`foreach ref slot i` `  obj[i] = 0;`	`foreach ref slot i` `  obj[i] = this_location;`
(c) Null reference comparison	`if (obj == null) {`	`if (obj == 0) {`	`if ((obj & 0xf0000000) == 0) {`
(d) General reference comparison	`if (obj1 == obj2) {`	`if (obj1 == obj2) {`	`if (((obj1 | obj2) & 0xf0000000) == 0 ||` `  obj1 == obj2) {`

Table 1. Java code (first column) and its corresponding meaning (using C semantics) for an unmodified VM (second column) and with origin tracking (third column).

tinguish themselves from legal addresses and that generate hardware traps when they are dereferenced. The key is for the VM to reserve and protect a range of addresses that will be used for these values via a system call to the operating system. Our implementation encodes program locations in the lowest 16th of the 32-bit address space, `0x00000000`–`0x0fffffff`. This range affords us 28 bits to store a program location. We divide up the bits to encode a <method, line number> pair. We use one bit to specify which of two different layouts are used for the remaining 27 bits:

1. The default layout uses 14 bits for method ID and 13 bits for bytecode index, which is easily translated to a line number.
2. The alternate layout uses 8 bits for method ID and 19 bits for bytecode index, and is used only for bytecode indices that do not fit in 13 bits. The few methods that fall into this category are assigned separate 8-bit identifiers, which are stored in a small "large method" table.

We find this layout handles all the programs we use to evaluate the performance and usefulness of origin tracking. Alternatively, one could assign a unique 28-bit value to each program location that assigns null (null constant assignments and many object allocations) via a lookup table. This approach would use space proportional to the size of the program, while our approach adds no space except for per-method IDs.

3.2 Redefining Program Operations.

This section describes how we redefine Java operations to accommodate representing null values with a range of values rather than with zero.

Null Assignment. At null assignments, our modified VM assigns the 28-bit value corresponding to the current program location (method and line number) instead of zero. The VM computes this value at compile time. Table 1(a) shows the Java code and its corresponding translation (using C semantics) for a standard VM and for the modified VM.

Object Allocation. Java specifies that when a program allocates a new object, whether scalar or array, its reference slots are initialized to null. VMs implement this efficiently by allocating objects into mass-zeroed memory. Since our modified VM uses nonzero values to represent null, our modified VM adds code at object allocations that initializes each reference slot to the program location. Table 1(b) shows Java code and its corresponding translation for a standard VM and a VM with origin tracking.

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 method and edge profiles collected by Jikes RVM and other VMs), the compiler inlines the code shown in the last column of Table 1(b). If the number of slots is a small, known constant (true for all small scalars, as well as small arrays with size known at compile time), the compiler flattens the loop.

Static fields are also initialized to null, but during class initialization. Our current implementation does not initialize static fields to program location values, but we plan to add this support for the final paper. Of the 12 bugs we evaluate in Section 4, one manifests as a null assigned at class initialization time.

Null Comparison. Standard VMs implement checking if a reference is null by comparing the reference to zero. Since origin tracking uses the range `0x00000000`–`0x0fffffff` for null, and non-null references are in the remaining range, `0x10000000`–`0xffffffff`, a reference is null if and only if its high four bits are zero. Thus, we implement the null test by performing a bitwise AND with `0xf0000000`. Table 1(c) shows how a standard VM and the modified VM implement null checks.

General Reference Comparison. A more complex case is when a program compares two references. A standard VM can simply compare their values. However, with origin tracking, two references may have different underlying values but both represent null. Two references are the same if and only if (1) each reference's high four bits are zero or (2) their values are the same. Table 1(d) shows the modified VM implementation. We optimize this test for the common case when both references are non-null. The instrumentation first tests if either reference is null by bitwise OR-ing the references together. If so, it jumps to an out-of-line basic block which

Program	Lines	Exception description	Origin?	Trivial?	Useful?
Checkstyle	47,871	Empty default case	Yes	Nontrivial	Most likely useful
Eclipse	2,425,709	Malformed XML document	Yes	Nontrivial	Most likely useful
Eclipse	2,425,709	Close Eclipse while deleting project	Yes	Trivial	Not useful
FreeMarker	64,442	JUnit test crashes unexpectedly	No	N/A	N/A
JFreeChart	223,869	Stacked XY plot with lines	Yes	Somewhat nontrivial	Marginally useful
JFreeChart	223,869	Plot without x-axis	Yes	Nontrivial	Definitely useful
JODE	44,937	Exception decompiling class	Yes	Nontrivial	Most likely useful
Jython	144,739	Use built-in class as variable	Yes	Nontrivial	Potentially useful
Jython	144,739	Problem accessing *__doc__* attribute	Yes	Somewhat nontrivial	Marginally useful
JRefactory	231,338	Package and import on same line	Yes	Trivial	Not useful
JRefactory	231,338	Invalid class name	Yes	Nontrivial	Definitely useful
Mckoi SQL DB	94,681	Access closed connection	Yes	Nontrivial	Definitely useful

Table 2. The diagnostic utility of origins returned by origin tracking in Java. Bug repositories are on SourceForge [29] except for Eclipse [8] and Mckoi SQL Database [21].

performs the more complex test. Otherwise, the common case performs a simple check for reference equality, since both references are now known to be non-null.

3.3 Implementation in Jikes RVM

We implement origin tracking in Jikes RVM, a high-performance Java-in-Java virtual machine [4, 15]. Jikes RVM uses two compilers at run time. When a method first executes, Jikes RVM compiles it with a non-optimizing, baseline compiler. When a method becomes hot, Jikes RVM recompiles it with an optimizing compiler at successively higher levels of optimization. We modify both compilers to redefine Java operations to support value piggybacking on null values.

Our implementation stores program locations instead of zero (Table 1(a)-(b)) only in application code, not in VM code or in application libraries. This choice reflects developers' overriding interest in source locations in their application code. Since the VM is not implemented 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(c)-(d)) in the application, libraries, and VM.

Jikes RVM by default catches null pointer exceptions using a hardware trap handler. Since the VM protects the address range `0x00000000`–`0x0fffffff`, dereferencing a null pointer causes the the hardware to generate the signal SIGSEGV. Jikes RVM's custom hardware trap handler detects this signal and returns control to the VM, which throws a null pointer exception. We modify the trap handler to identify and record the base address responsible for the trap, which is the value of the null reference. When control is returned to the VM, it decodes the value into a program location (method and line number), and reports it together with the null pointer exception.

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.

4. 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 different programs. These results are summarized in Table 2, which contains the lines of code measured with the Unix `wc` command, if the origin was identified, if the origin was trivially identifiable, and how useful we found the origin report. We describe the five most interesting cases in detail below and the other seven 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 5), there is no reason not to provide this extra information, which speeds debugging even when a defect is relatively trivial.
- Origin tracking provides extra information for users to put in bug reports. Often bug reports do not contain sufficient information for developers to diagnose or reproduce a bug. Since origin tracking is cheap enough to include in deployed software, it gives users more to report than just a stack trace.
- It is not always clear whether the defect lies in the code producing the null value, or in the the code dereferencing it (i.e., 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 applica-

tion code and library code. Therefore, even when they 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.

4.1 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? Our implementation misses one origin due to an uninitialized static variable (Section 3.2).

Triviality. Is the stack trace alone (along with the source code) sufficient to identify the origin? In seven of 12 null pointer exceptions, the origin is not easy to figure out via inspection.

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 in two cases, is marginally useful in two cases, is potentially useful in one case, is most likely useful in three cases, and is definitely useful in three cases. We uploaded potentially useful information to reports for unfixed bugs.

4.2 Origin tracking case studies

This section describes the five most interesting failures and shows origin tracking's role in discovering the program defect.

Case 1: Eclipse #1: Malformed XML Document The Eclipse integrated development environment version 3.2 [8] 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 tells specifically that the XML documents lacks a root element.

Figure 1(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 1(b), reveals exactly what is wrong with the XML file. The null value originates in the code that checks the consistency of

```
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 1. Case 1: VM output for Eclipse bug 1. (a) The stack trace alone just indicates a parsing error; (b) Origin tracking identifies the specific problem with the XML file.

```
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 2. Case 2: VM output for JFreeChart bug 2. (a) The stack trace shows a failure inside the library; (b) Origin tracking identifies the specific error in the user code.

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 help fix a bug in the XML parser.

Case 2: JFreeChart #2: Plot Without X-Axis The next case involves a user-provided program that causes a null pointer exception inside JFreeChart 1.0.2, a graphing library (Bug 1593150). This case represents an important class of failures for which origin tracking is useful: the failure is caused by a defect in 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 the code provided by the user, annotated with line numbers:

```
11: public static void test() {
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);
17: }
```

Figure 2(a) shows the exception stack trace. The method `FastScatterPlot.draw()`, called from line 16, 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 2). 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 fix implemented by developers, in version 1.0.3 causes the constructor to fail with an error message if the x-axis parameter is null.

Case 3: 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 properly maintained.

Figure 3(a) shows the stack trace produced by this failure. The exception occurs during initialization of an inner class because a pointer to information about its outer class is null. With only the stack trace and the source code, we were unable to understand the circumstances that left this pointer null. Furthermore, we could not find source for the class to be decompiled (provided in the bug report), which makes sense since the user reporting the bug wanted to decompile it!

Origin tracking, shown in Figure 3(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 highly useful to developers, since it reveals a problem with the logic involved in class loading, and we added it to the official bug report.

Case 4: JRefactory #2: Invalid Class Name While trying to reproduce a known bug (see appendix) in JRefactory 2.9.18, a tool for refactoring Java programs, we triggered a previously unknown failure. A null pointer exception occurs when the name of a class accidentally includes ".java":

```
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 3. Case 3: 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 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 4. Case 4: VM output for JRefactory bug 2. (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.

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

Figure 4(a) shows the stack trace produced by this exception. 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 `ExceptionPrinter.singleton` is null when `ExceptionPrinter.getInstance()` is called.

Figure 4(b) shows that `ExceptionPrinter.singleton` is set to null by the class initializer, and never initialized anywhere else. On inspection, we found that the other methods in `ExceptionPrinter` automatically initialize the `singleton` field whenever it is null, and we believe this fix is needed in the `getInstance()` method. We recently submitted this bug and the suggested fix to JRefactory's bug tracker (Bug 1674321).

```
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 5. Case 5: VM output for Mckoi SQL Database bug. (a) The stack trace alone just shows that the query failed; (b) Origin tracking suggests that the failure is due to a closed connection.

Case 5: Mckoi SQL Database: Access Closed Connection Mckoi SQL Database is a database management system for Java. We found a bug report on the product's mailing list that manifests as a null pointer exception in version 0.93 (Message 02079). The user reports that the database throws a null pointer exception when the user's code attempts to execute a query. This case highlights another important benefit of origin tracking: it identifies a null store far away from the point of failure (possibly in another thread), and location of that store identifies the precise reason the query fails.

The original user message does not include example code, so we used information from the developer's responses to construct a test case. Our code artificially induces the failure, but captures the essence of the problem. It is likely that the actual application is more complex.

The stack trace is shown in Figure 5(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 simply indicates that the query failed, with no error message or exception indicating exactly why.

Our origin information, shown in Figure 5(b), reveals the reason for the failure. The null store occurs in `AbstractJDBCDatabaseInterface.internalDispose():213`. This method is part of closing a connection, and line 213 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 is useful to both the application user and the database library developers. Users can probably guess from the name of the method `AbstractJDBCDatabaseInterface.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 in that case. The Mckoi developers came to this same conclusion on the database mailing list.

5. Java Runtime Performance

This section evaluates the performance impact of using value piggybacking to perform origin tracking for null values in Java programs.

5.1 Methodology

Execution Jikes RVM runs by default using *adaptive* methodology, which dynamically identifies frequently executed methods and recompiles them at higher optimization levels. Because it uses timer-based sampling to detect hot methods, the adaptive compiler is nondeterministic. To measure performance, 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 high 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 with a heap size fixed at three times the minimum possible for that benchmark with a generational mark-sweep garbage collector. We report the minimum of five trials. We take the minimum since it represents the run least perturbed by external effects.

Benchmarks We evaluate the performance of origin tracking using the the DaCapo benchmarks (version 2006-10), SPEC JVM98, and a fixed-workload version of SPEC JBB-2000 called `pseudojbb` [7, 30, 31]. We omit the DaCapo benchmarks `chart` and `xalan` because we could not get them to work with Jikes RVM and replay compilation, but will try to include them in the final paper.

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.

5.2 Space Overhead

Origin tracking adds no space overhead because it uses value piggybacking stores program locations *in place* of null references.

5.3 Execution Time Overhead

Figure 6 shows the overhead of running programs with origin tracking. We report the second run of replay methodol-

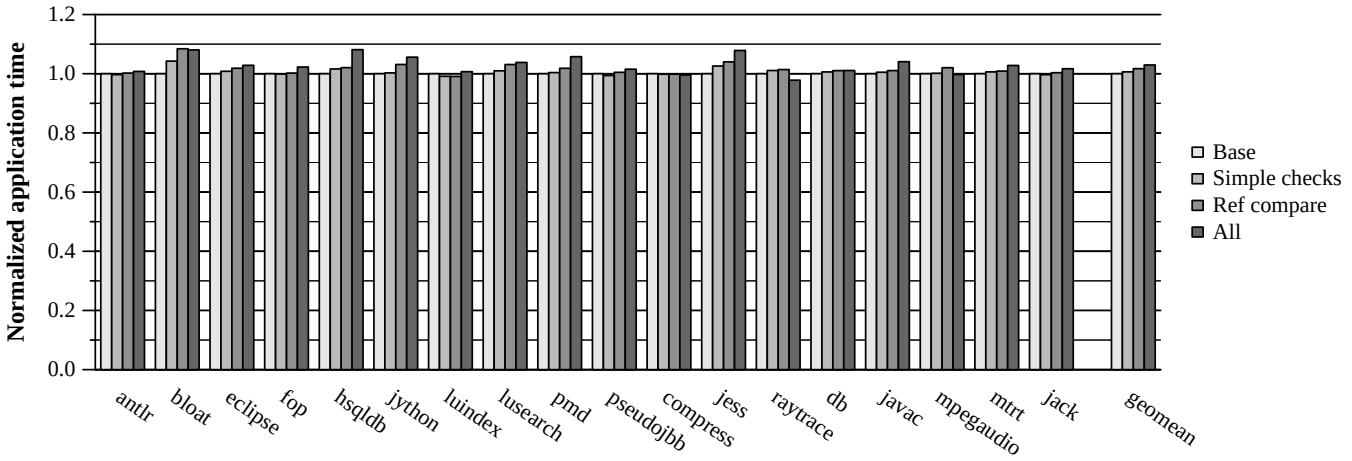

Figure 6. Application execution time overhead of origin tracking.

ogy, 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.
- *All* adds initialization of nulls at object allocation time to *Ref compare*, and this configuration contains all functionality needed to report origins for null pointer exceptions. *All* adds 1% over *Ref compare*, for a total of 3% on average.

5.4 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. We do not focus on compilation overhead because it is generally represents a small fraction of total execution time, particularly for long-running programs.

Figure 7 shows the overhead origin tracking adds to compilation for the same configurations as in Section 5.3. We determine compilation time by measuring time spent in the compiler during the first run of replay compilation, which compiles and executes the application. Origin tracking adds 12% compilation overhead on average. Not surprisingly, most of the compilation overhead comes from the addition of nontrivial instrumentation at general reference comparisons and at object allocation.

The maximum overhead, 52% for `pseudojbb`, is most likely a result of overly aggressive inlining and unrolling of instrumentation added by origin tracking: the code bloat causes downstream optimizations to take longer. We plan to try decreasing the aggressiveness of instrumentation expansion for the final paper.

6. Undefined Variables in C & C++ Programs

This section demonstrates that value piggybacking is able to identify the origin of undefined values in programs written in languages such as C, C++, and Fortran, and is thus a widely useful approach. This section first describes *Memcheck* [28], a tool for validating memory usage in programs, which is built with Valgrind [23]. We then describe how we modified Memcheck to use value piggybacking on undefined values to perform origin tracking, and discuss some alternatives for improving coverage of 32-bit and smaller memory locations. 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 17 C, C++ and Fortran programs. Of these warnings, 47 are for 32-bit values (in which it is possible to fit a program location). Our enhanced MemCheck reports the origin for 34 (72%) of the 47 undefined values.

6.1 Memcheck

Memcheck [28] is implemented in Valgrind [23], a framework for heavyweight dynamic binary instrumentation. Memcheck works on programs written in any language, but it is most useful for C and C++ in which memory errors are common. Memcheck detects a range of 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

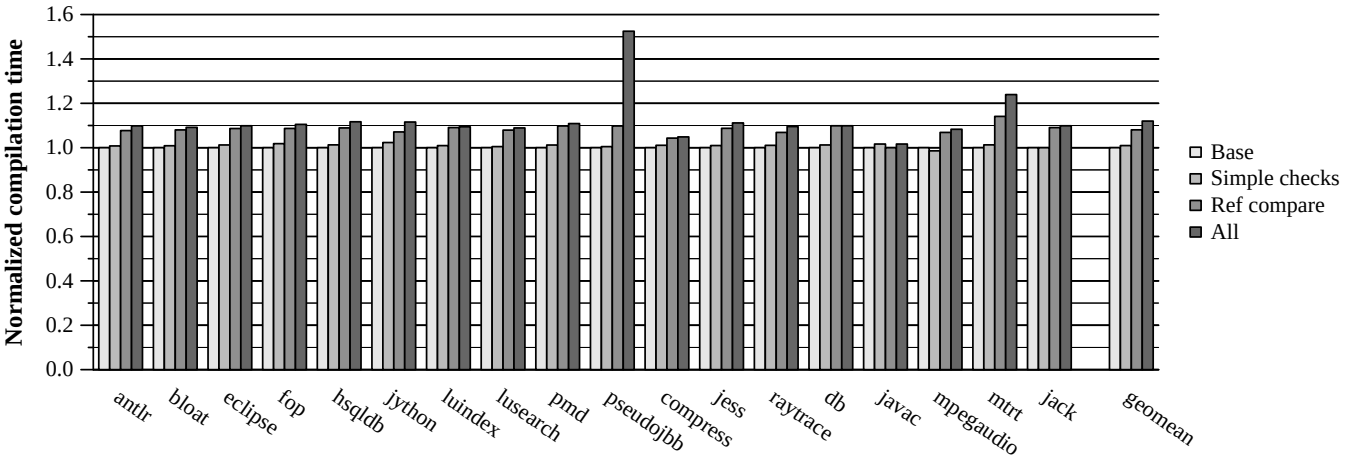

Figure 7. Compilation time overhead of origin tracking.

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 can also detect dangerous uses of undefined values by shadowing every register and memory byte with 8 *V bits* ('V' for "validity") which 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 relating to undefined value defects, Memcheck only warns users about the following four uses of undefined values that can change a program's behavior:

1. When the program uses an undefined value in a conditional branch or conditional move, potentially changing the program's control flow.
2. When the program uses an undefined value as the address in a load or store, potentially causing a segmentation fault.
3. When the program uses an undefined value as a branch target address, potentially changing the program's control flow.
4. When the program provides an undefined value as an input to a system call, potentially changing the program's side-effects.

In contrast, Memcheck does not warn about benign uses, such as copying undefined values which is not inherently error prone and is very common. For example, programs copy not-yet-initialized fields in C structs, and even more commonly, they copy empty bytes in structs due to padding between fields. Neither does Memcheck report when programs operate on an undefined value, such as adding an undefined value to another value. 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. Thus fixing undefined value defects from undefined value warnings can be difficult. Users of Memcheck complain about this case enough that it warranted an entry in Valgrind's frequently asked questions (FAQ) (starting in version 3.2.2) [23].

6.2 Details of Origin Tracking

Precisely because the values are undefined, we can give them whatever values we want, i.e., information about their origin. We make the following changes to Memcheck to implement origin tracking.

Memcheck computes and stores a stack trace for each *heap block*. The stack trace includes the line number for each allocation, and the associated calling context. The stack trace is stored in a data structure, and we use the 32-bit address of this data structure as the "origin key," i.e., a value from which we can identify an origin. We then paint each newly allocated heap block with repeated copies of the origin key. The exception is heap blocks allocated with 'calloc', which requires zeroes.

Memcheck takes a similar, but more lightweight approach to undefined values allocated on the *stack*. Because stack allocations are so frequent, recording a stack trace for every one would be slow. Instead, Memcheck records a static code location for each stack allocation. It is able to determine these code locations at instrumentation time rather than run time, which makes it much faster.

No changes are required to copying and operations involving normal program values. Nor are any changes re-

```
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 8. A Memcheck undefined value warning involving an origin.

quired to Memcheck's shadow values operations. Thanks to value piggybacking, the origin values get propagated within undefined values for free. Unlike the Java case, origins can be lost if undefined values are operated on in certain ways.

More changes are required for the conditional branch, load/store address, and branch target address cases. First consider the load/store address case. Before each load or store, Memcheck examines the V bits for the address. Assuming a 32-bit machine, Memcheck examines the V bits for the 32-bit address and if any are undefined, it issues an undefined value warning. Before doing so, it inspects the undefined 32-bit value itself, and tries to match it against any of the origin keys for the recorded stack traces. If it matches an entry, Memcheck mentions the origin key in its warning message as a likely origin for the undefined value. The branch target case is similar.

Conditional branches and moves are trickier. In this case, the final undefined value is a single bit in the condition. We need to search backward to find one or more 32-bit values from which the undefined condition bit was derived. If any of these are undefined, we can use their values to search for origin keys. We perform this searching step at instrumentation time. From each condition, the instrumentation code does a backwards dataflow trace to find any 32-bit values that are ancestors of the condition bit value. For example, if the program compares two 32-bit values and then uses the result as a condition, if either of those two values are undefined (determined by looking at their V bits when the warning is issued), we use those values as possible origins of the undefined value. The amount of backward searching is limited, because Valgrind instruments code in small chunks (superblocks) that are usually 3–50 instructions in length.

Figure 8 shows an example undefined value warning that includes an origin. The defect involved allocating a heap block in 'main', passing an uninitialized 32-bit value from within it into a function 'f', which then passed it to 'g', which then compared the value to another in an if-then-else. The first half of the warning is what is printed when origin tracking is not performed. The second half of the warning is the identified origin—the original heap allocation point.

6.3 Discussion and Potential Enhancements

This section discusses some limitations of origin tracking due to Memcheck and due to the C and C++ programming model, potential solutions, and related topics.

Missed origins due to small values. The biggest limitation of origin tracking in C and C++ is that we cannot track program locations well in fewer than 32 bits, and therefore do not report origins for undefined value defects involving 16-bit or 8-bit data. To address this problem, we tried performing "partial matching": reporting every origin for which the 16 or 8 bits in the undefined value matched a 16-bit or 8-bit fragments of the origin key. This approach resulted in many incorrect matches, particularly in the 8-bit case, and so was of little use. Eight bits is just not enough to store a code location. Unfortunately, the 8-bit and 16-bit cases occur often (see below).

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-bit 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. The "proper" solution is to not use piggybacking but store the origins separately, as is done for the V bits; but this would require extra space and be much slower, thus 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 tried doing "fuzzier matching": requiring that only three of the four bytes in the undefined value matched the origin key. This made little improvement, and increased the likelihood of incorrect matches. We could try to prevent modifications of origin values, the extra checks for almost every operation would be expensive. 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. However, we found 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 origins as "possible origin" in the warning messages.

Changing program behavior. Putting origin keys in undefined memory, instead of the often fortuitously found zero, may change a program's behavior. However, the changes are within an envelope of behavior that is already undefined. In

Program	Undefined warnings	32-bit values Origin	32-bit values No origin	<32-bit No origin
dvips	1	0	0	1
facerec	1	1	0	0
firefox	6	0	0	6
glibc	8	8	0	0
ispell	1	1	0	0
kanagram	8	0	1	7
kbounce	14	0	5	9
kpdf	1	1	0	0
ooffice	1	0	0	1
parser	2	0	0	2
pdf2ps	13	5	0	8
ps2ascii	4	4	0	0
ps2pdf	24	1	0	23
pstree	3	0	0	3
python	13	10	3	0
twolf	2	2	0	0
vim	2	0	0	2
xfig	4	0	2	2
xfontsel	2	0	0	2
xpdf	37	1	2	34
total	147	34 (23%)	13 (9%)	100 (68%)

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

this setting, we assume the programmer is trying to identify defects, and is fixing defects in the order Memcheck identifies them because 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 so the 32-bit case will still be common. 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).

6.4 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 that lacks any useful origin information. Three of these (facerec, parser, twolf) are from the SPEC CPU2000 suite.

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 datum warnings. We suspect that many of these are 8-bit warnings related to defects involving strings.

Of the 47 undefined value warnings involving 32-bit values for which Memcheck could possibly report origins, it reported 34 (72%). The 13 cases where it failed to identify an origin must involve unaligned or modified 32-bit undefined values, as discussed in the previous section. 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.

Although we have not included a detailed evaluation of the quality of the Memcheck origin reports for these programs, we posit they will help programmers in a similar manner to the origins of Java null pointer exceptions.

6.5 Overhead of Origin Tracking in Memcheck

To measure the overhead of origin tracking in Memcheck, we used the SPEC CPU2000 benchmarks (except for galgel, which gfortran failed to compile). We used the training inputs; these are smaller than the reference inputs, but as the experimental runs took more than 24 hours, we believe that using the larger inputs would not give noticeably different 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 9 shows the results of measuring Memcheck with and without origin tracking. Memcheck alone slows programs down by a factor of 28x. 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 slow-down 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.

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. The Java implementation adds 3% overhead on average, making it suitable for all-the-time use in production environments. We evaluate 12 bugs and find that origin tracking provides an origin for 11, a nontrivial origin for nine, and a most likely or definitely useful origin for six.

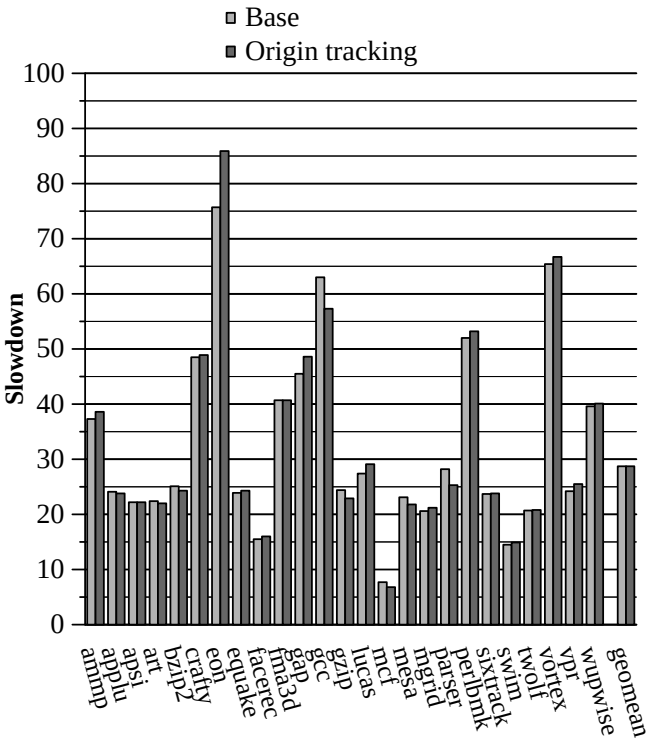

Figure 9. Memcheck's slowdown without and with origin tracking.

The toughest bugs are the ones helped most by origin knowledge. Given its minimal footprint and time-saving information, origin tracking is ideal for commercial VMs and can enhance debugging of deployed software right away.

Acknowledgments

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

Appendix

The appendix describes the exceptions not covered in Section 4.

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:
```

```
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 10. Checkstyle VM Bug output. (a) Exception output provided by vanilla VM. (b) Extra information provided by origin tracking.

```
      test = true;
      break;
    default:
  }
```

Without origin tracking, Figure 10(a) shows the resulting exception stack trace. We show here the following 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, and whether or not 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 10(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 parsing rather than in some (possibly erroneous) part of the Checkstyle code itself. Consulting the Antlr documentation, the Checkstyle developers can confirm the meaning of this null value and insert the proper null check. The bug report notes that adding this check fixes the bug. Developers fixed the bug in Checkstyle 4.3.

Eclipse #2: 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 close, but the

```
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 11. VM output for Eclipse Bug 2. (a) Exception output provided by vanilla VM. (b) Extra information provided by origin tracking.

save uses a project root object that the delete has already nulled.

Figure 11 shows the exception stack trace, as well as the extra information provided by origin tracking. We examined the Eclipse source code and found that the origin of the 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.

FreeMarker: JUnit Test Crashes Unexpectedly 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 12 shows the exception stack trace. Unfortunately our implemention of origin tracking does not provide an origin for this null pointer because it originates in a static field that is never initialized. Our implementation does not currently handle this case, but we plan to add origin tracking to class initialization for the final paper (Section 3.2).

JFreeChart #1: Stacked XY Plot with Lines JFreeChart is an advanced library for displaying charts in Java applications. A null pointer exception occurs when a user application attempts to generate a stacked XY plot that has lines enabled in version 1.0.0 (Bug 1593156). Figure 13(a) shows the exception stack trace that occurs. Inspecting the stack trace, we find that the exception occurs when JFreeChart dereferences `StackedXYAreaRenderer.lines`, which is null.

Origin tracking reports the origin (Figure 13(b)), which is in the constructor for `StackedXYAreaRendered`, where `StackedXYAreaRenderer.lines` is initialized to null. This information tells the developers that the `lines` is null

```
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: none
```

(b)

Figure 12. VM output for FreeMarker bug. Origin tracking is currently unable to provide an origin for this exception.

```
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 13. VM output for JFreeChart bug 1. (a) Exception output provided by vanilla VM. (b) Extra information provided by origin tracking.

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 not hard to determine this fact by inspection of the class.

Developers fixed the bug in the next version of JFreeChart by initializing `lines` to a default `Line2D` object.

Jython #1: Use Built-In Class as Variable Jython is an implementation of the Python language integrated with Java. We present two null pointer exceptions we found on the Jython discussion forum and that we were able to reproduce.

The first exception comes from an entry on the *jython-dev* mailing list archive, dated September 17, 2001 [17], and 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 14 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

```
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 14. Jython VM output for Bug 1. (a) Exception output provided by vanilla VM. (b) Extra information provided by origin tracking.

and determined that the implementation was not advanced enough to handle using a built-in class as a variable [32]. 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 diff between the two versions.

Jython #2: 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 could save a developer some 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.

JRefactory #1: 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 the exception stack trace and the extra information provided by origin tracking. The null's origin (line 1177) is just seven lines above the exception's occurrence (line 1184), and static inspection of the code shows that the

```
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. Jython VM output for Bug 2. (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. JRefactory VM output for Bug 1. (a) Exception output provided by vanilla VM. (b) Extra information provided by origin tracking.

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.

References

[1] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *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] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Conference*

on Object-Oriented Programming, Systems, Languages, and Applications, pages 47–65, 2000.

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

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

[7] 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 *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006.

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

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

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

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

[12] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Winter USENIX Conference*, pages 125–136, 1992.

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

[14] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 13–19, 2005.

[15] Jikes RVM. IBM, 2005. http://jikesrvm.sourceforge.net.

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

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

[18] 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.

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

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

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

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

[23] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Conference on Programming Language Design and Implementation*, 2007. To appear.

[24] 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*, 2005.

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

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

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

[28] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, pages 17–30, 2005.

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

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

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

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

[33] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards Locating Execution Omission Errors. In *Conference on Programming Language Design and Implementation*, 2007. To appear.

[34] 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 *International Symposium on Microarchitecture*, pages 269–280, 2004.