Dynamic Error Remediation: A Case Study with Null Pointer Exceptions

Stephen W. Kent
Supervising Professor: Dr. Kathryn McKinley

May 2, 2008

Abstract

Managed programming languages have improved the robustness of software by reducing some classes of errors. Despite these improvements, programs still contain errors. Even with help from existing static and dynamic analysis tools, the errors are often difficult to detect and can be even more difficult to successfully eliminate. This work introduces a new approach, called *dynamic error remediation*, that further improves the robustness of software, by tolerating some null pointer exceptions until a patch can be created to fix the exception. This paper describes dynamic error remediation, and its effectiveness, with different strategies for handling the exceptions. We show that dynamic error remediation is successful in allowing programs to continue execution and to do meaningful work in many cases. We then describe the bug suite created to test dynamic error remediation, as well as providing a methodology for creating other bug suites. Finally, we describe *origin tracking*, a JVM modification which exposes the origin of null values that cause null pointer exceptions. Because dynamic error remediation is successful in many cases, we assert that it is a viable strategy for increasing availability and reliability in many programs. While increasing availability over correctness is not ideal in all cases, we show that there are real examples where it is preferred, and we provide dynamic error remediation as a tool for developers and users to survive null pointer exception bugs.

1 Introduction

While many advances in managed programming languages, such as garbage collection, disciplined pointer usage, and array bounds checking, prevent previously common errors, programs still contain bugs. Because these bugs, if undetected, can cause software to crash, testing and bug detection tools help developers locate and eliminate bugs in software before deployment. Static analysis checks a developer's source code [9][18], and dynamic analysis determines if runtime program invariants are being violated [4] [6] [8]. These tools find a variety of common and not so common bugs. Despite the wide use of these tools, bugs still escape analysis and make it into deployed code.

One insideous bug is the null pointer exception, which by its "null" nature is hard for programmers to fix. These bugs indicate that nothing was found in memory where something should have been, giving programmers very little to work with to fix the bug besides a stack trace. Null pointer exceptions sometimes show themselves only with certain inputs, making these bugs difficult to find before deployment. As Table 1 shows, 1-2% of developer code is devoted to identifying null objects.

This paper introduces a new method of tolerating null pointer exceptions, *dynamic error remediation*, which is designed to make programs more reliable in the face of null pointer exceptions. Implemented in the

Program	# of Files	# of Lines	Lines with null checks	% of lines with null checks
JODE	117	11465	342	2.9
Checkstyle	1576	101871	1605	1.5
FreeMarker	275	39639	928	2.3
JFreeChart	1907	229873	4946	2.2
JRefactory	1188	110824	1651	1.5
Mckoi SQL DB	277	47617	634	1.3
JGAP	379	37476	418	1.1

Table 1: The number of lines containing null checks in each program tested with Runtime Error Tolerance. Number of Lines calculated without counting comments or blank lines.

exception handler of Jikes RVM [10], dynamic error remediation changes the behavior of the VM so that, instead of a null pointer exception resulting in a software crash, the runtime system allows the software to continue executing past the normally fatal exception. This behavior is achieved by modifying the VM such that, when a null pointer exception occurs, the VM can handle the exception through a variety of strategies and return control back to the software instead of terminating it. We investigate five possible strategies, three of which we implement in Section 3.

This method is similar to previous bug tolerance techniques [2] [15] described in Section 2 where an error is tolerated by the runtime system when it occurs. However, the majority of previous techniques achieved tolerance of memory errors in unmanaged languages such as C and C++ through modifications to the environment. Dynamic error remediation uses an approach similar to *failure-oblivious computing* [17], designed to tolerate null pointer exceptions in Java by acting directly on the program values causing the error.

To test the effectiveness of dynamic error remediation in handling null pointer exceptions, we create a suite of programs culled from SourceForge [19]. The programs in the suite were chosen such that each contained one or more reproducible null pointer exceptions. For each program in the suite, we create a test case which throws the null pointer reliably. For each case, we describe the cause of the exception, as well as showing how dynamic error remediation handles the case using each implemented strategy, and how the program behaves after recovery.

To determine the usefulness of dynamic error remediation, we analyze how the program behaves and how well it handles the null pointer exception. This analysis provides the usefulness by answering the question of whether or not it is possible to recover from a null pointer exception successfully, and how happy a user would be with the results. The results of the paper show that dynamic error remediation allows the program to continue execution to a successful end, albeit not always 100% correctly, in 9 of the 9 null pointer exception cases with at least one of the implemented strategies, and increases user happiness in 4 of the 9 cases. The case study and the results of dynamic error remediation are described in detail in Section 4.

We also used this suite of bugs with four other bugs from Jython and Eclipse to test *origin tracking* [3]. Origin tracking is a modification to the runtime system which provides information about the origin of null values which cause null pointer exceptions. Currently, when a null pointer exception occurs, runtime systems only report that the exception occurred and the stack trace of the user application at the time of the exception. This information is often not enough, as the null pointer exception can originate far from the offending instruction that triggers the null pointer exception. Origin tracking uses *value piggybacking* to maintain the origin of null values in place for reporting the origin at the time of the exception, giving users more information about the null pointer exception. With the origin, users can track through the program from the beginning of the null value to the exception, instead of tracking backwards from the last call on the stack. Origin tracking is described in more detail in Section 5.

The main contributions of this paper are:

- A simple methodology for creating a suite of buggy programs with a specific bug for use in testing bug tolerance and bug diagnostic systems.
- A suite of 7 programs representing 9 cases with null pointer exceptions for testing origin tracking and dynamic error remediation.
- The introduction, implementation, and evaluation of dynamic error remediation for null pointer exceptions in deployed software.

While dynamic error remediation cannot guarantee correct execution in every case, we believe that in some cases it is better to increase the availability and reliability of programs over the correctness of output. In the case of GUI programs for example, we assert that it is often better to have a window render incorrectly, or not display at all, than have the program crash as a result of a null pointer exception. Conversely, programs which perform scientific calculations, are examples where dynamic error remediation would not be suited for use. Because the goal of these programs is to provide an accurate result quickly, a solution to a crash which could not guarantee correctness is unacceptable.

As the results of the paper show, programs such as JODE or JFreeChart produce acceptable results with dynamic error remediation in place. With JODE, had the programmer not terminated on an assertion failure and continued to the next input files, JODE would have decompiled those files correctly. With JFreeChart, the example illustrates that in a GUI program the window being drawn fails to draw, but the rest of the UI could draw correctly. Because cases exist where dynamic error remediation is useful, we believe that dynamic error remediation represents a viable strategy for increasing the robustness of certain programs in the face of null pointer exceptions that would normally result in a crash. Dynamic error remediation can be set as a configuration parameter by the developers or the users of a program allowing both parties to use dynamic error remediation as necessary for surviving bugs which could arise.

2 Related Work

This section describes work which has been done in the area of bug analysis and bug test suites. Work related to bug analysis typical falls into two categories: bug detection and bug tolerance. There has been a large amount of previous work related to preventing bugs from reaching deployed software. Bug detection tools use either static analysis or dynamic analysis to identify places in source code which represent possible bugs. However bugs still make it into deployed code despite bug detection tools and work has been done to develop several different ways to tolerate bugs in deployed code. Failure-oblivious computing [17] tolerates bugs by using methods which ignore faulty behavior by the program. DieHard [2] and Exterminator [15] both tolerate bugs by probabilistically lowering the chances that they occur with memory replicas. Rx tries to manipulate the environment in several different ways and then roll back to a previous point in execution to tolerate each bug. Unfortunately, while each of these methods can succeed at helping with environmentally affected bugs, they currently falls short of tolerating semantic errors, with the exception of failure-oblivious computing, which is most closely related to dynamic error remediation.

Unlike bug detection tools, there has been relatively little work done to create bug suites, as well as a standard for bug tests suites. Ruter et al. describes the bug suite used in their bug detection tool analysis. Their suite contains few programs however and it is unknown which bugs each program contains. Lu et

al. introduces a more robust bug suite as well as guidelines from personal experiences which they believe should be followed when creating a bug test suite for benchmarking bug detection tools. However, their guidelines do not allow for creating bug suites for the purpose testing of software with a specific application such as origin tracking or dynamic error remediation. BugBench would be useless for our work, therefore we create our own bug test suite, described in Section 4.

2.1 Bug Detection

Bug detection tools are generally used to find and detect places in code which could potentially cause bugs if not corrected. This analysis can be done statically by analyzing the source of a program, or dynamically by monitoring a specific set of statistics of a running program to determine any anomalous behavior that may occur as a result of a bug in the program. Bug detection tools are most often designed to be used by developers and testers before code deployment to find and eliminate bugs.

2.1.1 Static Analysis

Static analysis is commonly employed by bug detection tools because of the inherent usefulness for finding bugs of all types through source code analysis. Many bug manifestations occur under a deterministic set of circumstances which can then be found by bug detection tools using techniques such as dataflow analysis or common coding idioms. The downside to these tools, however, is that they can suffer from a large number of false positives as well as not finding every occurrence of a particular type of bug if unsound [18].

FindBugs is a bug detector that relies on static analysis and common coding idioms to detect bugs [9]. These coding idioms often represent errors, and therefore if one exists, an error most likely exists as well. The idea behind using coding idioms is that while simple, they represent real, commonly occurring problems, and could potentially be as useful as the other sophisticated techniques. They implemented a variety of detectors to detect errors such as inconsistent synchronization, potentially null dereferencing statements, or if a stream is never closed.

Ruter et al. analyze the similarities and differences of a few current bug finding tools for Java: PMD, JLint, Esc/Java, Bandera, and Findbugs, which each use static analysis, as well as introducing a meta-tool to report the union of all of the outputs from each program [18]. Each bug detection tool uses different heuristics to determine possible bugs. The analysis for each detector then includes determining which bugs each detector found and what technique the detector used for finding bugs. Ruter et al. also determine if there is a correlation between similar bug finding tools and the bugs they found. The results of the paper show that the bugs found for each detector are largely uncorrelated even when two detectors use a similar method. Because the data is largely uncorrelated, Ruter et al. propose a meta-tool which is designed to output the union of the bugs found, giving the user the most complete bug analysis.

2.1.2 Dynamic Analysis

Bug detection tools using dynamic analysis are also employed by developers to find bugs. These tools instrument code to gather specific metrics at runtime to establish invariants of a correctly running program [1] [7]. Using these metrics, these tools can then detect bugs which cannot be easily detected at compile time, such as memory leaks or dangling pointers, by determining if the established invariants have been violated. Unfortunately these tools incur high overhead limiting their used to testing time only.

DIDUCE and *Daikon* are automated bug detectors that use dynamic invariants to report potential sources of bugs [6] [8]. To create these dynamic invariants, DIDUCE and Daikon require training runs on correct program behavior. Once the invariants have been established, they are then run with the program to detect any bugs that result in these invariants being violated. When a bug is detected, DIDUCE and Daikon report the bug and the information related to the invariant violation. Unfortunately, DIDUCE and Daikon cannot detect every bug because not every bug will violate the dynamic invariants. Also, because each instruments the program, they incur a high execution overhead, limiting them to pre-deployment testing.

HeapMD is a bug detector that analyzes the heap of a program to find heap invariants, such as arity of pointers in/out of an object [4]. HeapMD then checks for anomalies based on previously determined invariants, similar to DIDUCE. For example, in a normal execution, the number of the objects on the heap that have one incoming reference is always about the same amount, but in a buggy execution, the number increases as a function of time. Like DIDUCE, HeapMD uses the notion that at least one of these heap properties will remain invariant throughout the program, and a bug occurs when the invariants no longer hold for the heap. Because HeapMD has high overhead, it is only suitable for testing environments.

2.2 Bug Tolerance

Software which seeks to tolerate bugs is different from bug detectors because it is not used to prevent bugs, but used to allow a program to handle the occurrence of a particular type of bug. The goal of bug tolerance is to mask the bug from the user, allowing the program to seem as though it is behaving as if no bug had occurred. Bug tolerance software is ideal for use with deployed code as the software has been put in place to circumvent any bugs which have escaped the developers during the bug detection process and made it into deployed code. Bug tolerance can be classified into two categories based on its methodology for masking bugs: environment altering tolerance or state altering tolerance.

2.2.1 Environment Altering Tolerance

Tolerance methods which alter the environment tolerate bugs by modifying the environment in way which negates the original effects of the bug. For example, DieHard can negate the effect of buffer overflow errors by approximating an infinite-sized heap, and ideally placing objects far enough apart that no writes can reach another object's memory in the heap. These approaches work well for memory errors, but cannot tolerate bugs which are semantic bugs, such as null pointer exceptions in Java. These bugs are not affected by changes to the environment, as the value causing the bug never changes.

DieHard seeks to prevent memory errors in unmanaged languages like C/C++ such as buffer overflows and dangling pointers by creating a system that approximates an infinite heap [2]. With an infinite heap, a program would never contain buffer overflows and dangling pointers because each object would have infinite memory to use and would thus never enter the memory space of a previous or current object. To approximate the infinite heap, DieHard uses heaps two or more times larger than than needed so objects are spaced out. These heaps, with randomization and replication of the memory, probabilistically lower the chances of memory errors. With multiple randomly allocated replicas, there is less of a chance that a buffer overflow or a dangling pointer will corrupt every replica. DieHard detects errors by finding discrepancies between replicas. The replica that does not agree is statistically likely to contain the error.

Rx is a method for surviving software failures invisibly and safely [16]. Unlike dynamic error remediation, Rx seeks to recover from a software failure deterministically and safely. To do this, Rx maintains checkpoints of the program state which Rx then uses to roll back to a point in the program which is correct. From

this point Rx then makes a change in the environment to try to avoid this error the next time around. If this change does not fix the problem, Rx repeats the process of rolling back until a change allows the program to continue safely and correctly. While Rx succeeds at being able to fix bugs related to the environment such as buffer overflows, Rx cannot correct bugs caused by faulty semantics. In this case, the user would have to wait an extended period of time to discover that the program could not be fixed.

Exterminator is a runtime system that is designed to detect, tolerate, and correct heap-based memory errors [15]. To detect heap-based memory errors, Exterminator uses DieFast, a probabilistic debugging allocator based on DieHard [2]. DieFast extends DieHard, to, instead of just probabilistically tolerating memory errors, also detect them. Exterminator maintains information about each detected bug and uses this information to create a patch for the program which fixes the bug. Using a correcting allocator, Exterminator can use a created patch to automatically pad allocated objects which would incur a buffer overflow, or defer object deallocations to prevent dangling pointers.

2.3 State Altering Tolerance

Unlike tolerance methods which alter the environment, state altering methods act on a program's values rather than the environment in which the program runs. By modifying the values of the program, state altering methods are able to tolerate semantic errors, such as null pointer exceptions, which environment altering methods are incapable of tolerating. This tolerance can be achieved through a variety of implementations such as manufacturing values for faulty reads, or injecting a value in place of null and then re-running the original instruction.

Rinard et al. introduce *failure-oblivious* computing for memory errors in C/C++ [17]. Failure-oblivious computing tolerates errors by ignoring errors so the program can continue execution. For example, a buffer overflow in a server implementing failure-oblivious computing would detect a write beyond the bounds and ignore it. The user who sent the faulty request sees an incorrect result, but the server does not fail later due to the buffer overflow. Failure-oblivious computing adds overhead during normal execution due to the necessity of extra code instrumented into the program by the compiler to detect and ignore the various types of errors.

Dynamic error remediation is similar to failure-oblivious computing in that they are both state altering tolerance methods. However, unlike failure-oblivious computing, dynamic error remediation tolerates null pointer exceptions through an active approach rather than an oblivious approach. Where failure-oblivious computing handles a null pointer dereference obliviously, by manufacturing a value for the dereference and then proceeding on, dynamic error remediation can handle a null pointer exception by replacing the null value with an object of the correct type and then re-running the original instruction. This approach performs analysis on which type the null value should represent, remedying the exception instead of obliviously moving past it.

2.4 Bug Test Suites

Not directly related to detecting bugs, bug test suites are created for the purpose testing bug detecting and tolerating techniques. Bug test suites are created such that each program in the suite contains a real bug as opposed to an injected one, and that a large number of bug types and program types are represented. These suites can determine the effectiveness of various bug detection tools such as the ones described in this section on the basis of if they found the error correctly or not. Also, it can determine the effectiveness of tolerance methods on the basis of several different criteria.

Lu et al. introduce a bug test suite, called BugBench, specifically for the purpose of running benchmark tests on the effectiveness of several bug detection tools [12]. Lu et al. first detail several guidelines that should be followed when approaching the bug test suite creation process. The guidelines they suggest are that every program included should contain real bugs not injected ones, cover a variety of bug cases, be portable and accessible, and not biased towards any detection tools. Using these guidelines, Lu et al. introduces BugBench, which is a bug suite containing 19 programs, each program containing a bug of one of the three major bug types, memory bugs, concurrency bugs, or semantic bugs. This bug test suite would be particularly well suited for evaluating bug detection software, as it covers a wide variety of bug types.

The bug suite used in the analysis of the bug comparison tools in Ruter et al. [18] was created by the authors of the paper specifically for the purposes of their comparison. Their bug suite contains five programs, and were chosen because they were mid-sized, at various states of maturity, and represented a wide range of functionality. However, the authors did not choose their programs based on known bugs, as was the case in Lu et al., and our bug suite. This limits the effectiveness of their bug suite, as the specific bugs are not known for future work in the area to focus on. We believe that Ruter et al. would have benefitted from doing their analysis with the bug suite provided by Lu et al. as it represents a more robust suite with known bugs that the authors could have specifically focused their results on.

3 Dynamic Error Remediation

This section describes dynamic error remediation and its motivation, as well as detailing possible strategies to be leveraged in implementing state-altering tolerance. This section also describes the implementation of dynamic error remediation and briefly describes the experimental results.

3.1 Dynamic Error Remediation

As software becomes a more integral part of our everyday lives, the availability of software also becomes more important. Unfortunately even with improvements to programming languages, testing, and bug detection tools, some bugs still go undetected and lead to runtime exceptions that cause software to crash, making the program unavailable until the software is restarted. This can be very costly; developer time and money is spent implementing code which helps in the detection and prevention of errors such as null pointer exceptions, shown in Table 1. Along with developer time and money spent in error checking, NIST estimated in June of 2002 that software errors cost the U.S. economy $59.5 billion dollars annually [14].

Bug tolerance is the idea of modifying the runtime system such that a program is can proceed past an error which normally causes a crash or safely proceed past a bug which would cause faulty computation. By making a program more tolerant to former type of errors, bug tolerance can increase the availability of programs. Instead of crashing as is the case in today's systems, with bug tolerance a program can continue past the error. This approach increases availability, but does so at the cost of correctness. However, if a bug occurs in a non-crucial module of a program, we assert that it would be better to continue executing at the expense of that module's correctness than crashing the entire program.

Previous work has been done to create several methods of bug tolerance. Failure-oblivious computing [17] ignores the error and discards faulty writes to allow the program to proceed as usual, negating writes which could cause buffer overflows, for example. Rx [16] implements bug tolerance through checkpointing and environment modification. Diehard [2] and Exterminator [15] create bug tolerant systems through memory replicas and an approximation of an infinite-sized heap. However because each of these methods, with the

exception of failure-oblivious computing, implements environment-altering tolerance, they each fall short of tolerating semantic errors such as null pointer exceptions in Java. If a semantic error occurs, no environment changes will help, rendering each of the current techniques useless.

Unlike these approaches, we propose dynamic error remediation which takes actions similar to what the programmer might do to fix the bug. Dynamic error remediation, described in this section, is a state-altering method similar to failure-oblivious computing. Whereas failure oblivious computing implements tolerance in an oblivious way, dynamic error remediation implements tolerance remedially which takes an action when a null pointer exception occurs, modifying the program's state to remediate the bug and allow execution to continue. While neither previous techniques nor dynamic error remediation guarantee correctness after a bug occurs, dynamic error remediation can guarantee, in most cases, a set of semantics that will be maintained by handling the runtime exception. For dynamic error remediation, we considered the following strategies.

3.2 Strategies

Skipping to the next machine instruction. Perhaps the simplest strategy to implement is to skip to the next machine instruction upon an error. This approach has the advantage of minimizing the amount of code that must be skipped over as the only code that is skipped is the machine instruction which represents the point at which the error occurred. Also, this allows us to stay in the current scope, minimizing the amount of work skipped. Unfortunately, this strategy does not have a defined result; skipping to the next machine instruction leaves the register or memory location which was being stored in an undefined state. This strategy is similar to failure-oblivious computing, but it does not work well for Java we show later.

Skipping to the next Java instruction. Skipping to the next Java instruction is very similar to the previous strategy except that the amount of code that is skipped over is slightly greater. Skipping to the next Java instruction is semantically equivalent to placing a try catch block around the faulting instruction. This strategy is preferable to the previous strategy, as it is handles the exception in a way which represents an approach a programmer might take. A downside to this approach is that the optimizing compiler often obscures the boundary of original Java instruction. Another downside is that more than likely the value of the failing Java instruction will immediately be needed, causing more errors.

Returning control to the enclosing scope. By returning control to the enclosing scope, such as the end of a loop, the runtime system can model returning from an error inside of a block of work. If the block of work fails, return to its enclosing scope, and continue. This strategy has the advantage of handling an error inside of a block of work, without skipping more code than necessary. Unfortunately this strategy could still introduce undefined values, if for example, a loop fails during the instantiation of an array, the rest of the array would be undefined. Also, this strategy requires a greater deal of analysis done by the VM to determine exactly where in the program represents the end of the scope, causing increased runtime with respect to the exception handler.

Returning control to the caller of a failing method. This strategy models the behavior of a method placed inside of a try-catch block. Common coding conventions are to write methods such that there is a return value, which has one value that represents a failing case. By returning from a failing method, the runtime system can directly model this behavior. This strategy has the advantage of handling the error in a way that the program is most likely to be able to handle, due to a failing case being common. The downside

to this approach versus the other approaches is that it skips the most code in the program. However, it is also the safest as it always has a defined set of semantics.

Injecting an object in place of a null value. The last strategy is specific to null pointer exceptions. If a null pointer exception occurs, then the user expected an object to exist. Thus the runtime system could create an object or reuse an object already in the system and place it where the null previously existed, restarting the instruction that failed, so that the next time through the instruction succeeds. The advantage of creating or reusing an object is that no code needs to be skipped, and the program can continue executing where it expects to be executing from. Also, in some cases, such as where a new object, with none of its fields instantiated, is expected, dynamic error remediation would allow the program to continue correctly.

3.3 Implementation

The previous section detailed why dynamic error remediation should be considered as a feasible option for VMs and described five strategies for active bug tolerance. This section describes a simple implementation of three strategies in the Jikes RVM [10]: returning from a method, skipping to the next machine instruction, and injecting an object in place of a null value. For simplicity, we currently only implement dynamic error remediation in the baseline compiler, as this gives us easy access to the bytecodes which are used to determine the correct type of the object needed for injecting an object in place of the a null value. Working in the baseline compiler also simplifies the implementation of returning from a method as the baseline compiler does not perform inlining.

3.3.1 Dynamic Error Remediation in Jikes RVM

Dynamic error remediation is implemented in the Jikes RVM exception handler. Normally, when control flow reaches the exception handler due to a runtime exception, the exception handler searches the call stack looking for a catch block to catch the exception. If a catch block is found, control is passed to the catch block with the exception and its relevant information as the arguments. If a catch block is not found, the program is terminated, outputting a stack trace for diagnosis, because with a standard VM exception handler, runtime exceptions are not recoverable.

With dynamic error remediation, the exception handler is modified so that on top of the original exception handler, it also contains a remediating exception handler, which tries to tolerate the exception. When a runtime exception is detected, the VM determines how the exception handler should behave, based on specific command-line arguments which determine the conditions under which runtime exceptions are handled using the remediating exception handler. If the conditions are met, the VM directs the exception handler to use the remediating handler in order to tolerate the exception.

Using the remediating handler, the exception handler chooses between one of the three strategies that have been implemented based on the command-line argument specified:

- Return control to the machine instruction occurring after the failing machine instruction.
- Return control to the caller of the failing method, as if a normal return from the failing method had occurred. It returns a null value for object return types, or a default value, such as false or zero, for primitive return types.

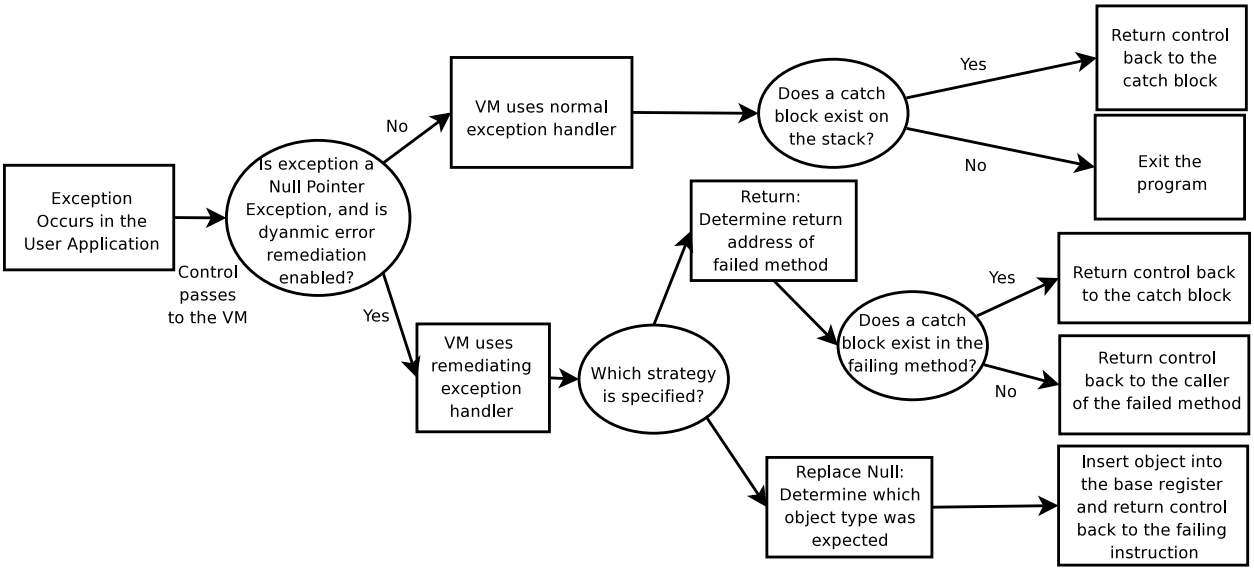

Figure 1: **Dynamic Error Remediation.** Control flow of the VM with dynamic error remediation in place.

- Inject an object of the correct type in place of the null value and then rerun the failing machine instruction.

Of the five strategies described in the previous section, we implemented these three strategies because of their variety of scope and method of tolerating the exception. Returning control to the enclosing scope to the next Java instruction requires information about the control flow of the program which we have not yet implemented. Returning control to the next Java instruction is very similar to the strategy of returning control to the next machine instruction and is thus left unimplemented to date.

3.3.2 Returning to the Caller

If the strategy of returning to the caller is specified, the exception handler needs to decide two things: (1) where does control return to, and (2) what is the return type, if any, of the failing method. To determine where to return, the handler gets the return instruction of the failing method, which corresponds to where it would return after successfully completing the method. The handler sets the instruction pointer register to this value, which establishes where to return to. Then using the return type of the method, the handler then returns null, zero, false, or nothing. The runtime exception handler returns these values for their respective return types because it is common coding practice to use these values to represent that the method failed. Returning null for objects has a useful side effect that if the return value is used, this causes a null pointer exception which is then handled again by the runtime exception handler, eventually causing control to resume outside the scope of the return value.

Jikes RVM follows the x86 calling conventions for handling returns [10]. Since dynamic error remediation models a runtime exception as a return we need to model these specifications exactly. For a return in x86, the VM first updates the frame pointer and stack pointer to reflect the return back to the caller. Then the VM restores all of the non-volatile registers. Finally the return value is stored in *T0*, a temporary register, or *T0* and *T1* for an 8 byte return, and the *ip* register is set to contain the instruction that a method is returning to.

With the return value set and the *ip* register set to the return location, the handler then removes the stack

frame of the failing method off of the stack by treating it as if it had returned successfully, as is defined by x86 calling conventions. The runtime exception handler returns control back to the user application at the return instruction. This implementation successfully turns a null pointer exception into a return from the method with a null, zero, or false result.

3.3.3 Returning to the Next Machine Instruction

Another strategy which can be specified is for the exception handler to return to the next machine instruction, which has a very simple implementation. When the VM detects an exception, it passes control to the exception handler, with the machine's instruction pointer pointing to the next machine instruction after the failing instruction. To implement returning control to the next machine instruction, all the runtime exception handler needs to do is to return, which gives control back to the program starting at the next machine instruction.

Unfortunately, although simple, in practice this strategy was found to be useless for continuing execution. Because control is passed to the machine instruction following the failing machine instruction without fixing the exception, the VM always segmentation faults with our bug suite. After studying the machine code, this is because a machine instruction that fails with a null pointer exception almost always is putting something into a register or into memory that will be used by the next instruction. Because the exception does not get handled, the value in the register or memory location becomes undefined. This undefined value gets used later, causing a segmentation fault.

3.3.4 Replacing Null with an Object

If the user specifies the strategy of replacing the null value with an object, the exception handler needs to determine several things when a null pointer exception occurs. The first is what the address of the failing instruction is. To determine this address, the VM's C hardware exception handler is modified to provide this information as an argument to the exception handler. The next is what type of object was expected where the null was found. To determine what the type is, the exception handler determines the bytecode of the failing instruction. Since null pointer exceptions only occur on field references, method references, and array accesses, the VM can determine the object type that was expected by analyzing the bytecode of the failing instruction.

The next step for the exception handler is to replace the null value with an object of the correct type. To implement this step, we first modify the VM so that upon resolving a class it also creates an instance of that type and stores it with the class information. When the exception handler determines the type expected, it looks up the class information and pulls out a reference to the instance we have created. Because the base register of the failing instruction is also provided to the exception handler from the C hardware exception handler with the address of the failing instruction, the exception handler only needs to store the value in the base register and return. Upon returning the program re-executes the failing instruction due to the exception handler setting the instruction pointer to the failing instruction. One caveat to this approach, is that for method references, if the method has never been called before, the exception handler also needs to place the address of the instance on the top of the stack, as the method resolver uses this location, where it expects the object to be, to determine for what type of object it needs to resolve the method.

Program	Exception description	Finishes?	More work?	User Happiness	Stack Depth/Returned
Checkstyle	Empty default case	Yes	Yes	Happy	10 methods/1 method
FreeMarker	JUnit test crashes unexpectedly	Yes	Yes	Happy	24 methods/1 method
JRefactory	Package and import on same line	Yes	Yes	Happy	10 methods/1 method
JRefactory	Invalid class name	Yes	Yes	Happy	8 methods/1 method
JFreeChart	Stacked XY plot with lines	Yes	Yes	Depends	4 methods/1 method
JFreeChart	Plot without x-axis	Yes	Yes	Depends	3 methods/1 method
JGAP	Missing Comparator	Yes	Yes	Unhappy	3 methods/1 method
JODE	Exception decompiling class	Yes	No	Moderate	7 methods/1 method
Mckoi SQL DB	Access closed connection	Yes	No	Depends	5 methods/2 methods

Table 2: The utility of returning from a method for each exception in Java. Bug repositories are on SourceForge [19] and Mckoi SQL Database [13].

Program	Exception description	Finishes?	More work?	User Happiness
Checkstyle	Empty default case	Yes	Yes	Happy
FreeMarker	JUnit test crashes unexpectedly	Yes	Yes	Happy
JRefactory	Package and import on same line	Yes	Yes	Happy
JGAP	Missing Comparator	No	No	Unhappy
JRefactory	Invalid class name	No	No	Unhappy
JFreeChart	Stacked XY plot with lines	Unknown	Unknown	Unknown
JFreeChart	Plot without x-axis	Unknown	Unknown	Unknown
Mckoi SQL DB	Access closed connection	Unknown	Unknown	Unknown
JODE	Exception decompiling class	Unknown	Unknown	Unknown

Table 3: The utility of replacing the null value with an object for each exception in Java. Bug repositories are on SourceForge [19] and Mckoi SQL Database [13].

3.4 Experimental Results

We ran dynamic error remediation using each of the three strategies described above on the programs in the bug suite described in Section 4. In each case, we analyzed the results of dynamic error remediation with the success or failure determined by whether or not the program continued to a graceful termination, and had a capacity to do more work.

Using the strategy of skipping to the next machine instruction, every case failed due to a segmentation fault, as the null value was used immediately following the failing instruction. With the strategy of replacing the null value with an object, dynamic error remediation successfully allowed three of the nine cases to do more work and continue to a graceful termination, failing on two of the cases, and being unable to test the other four due to unimplemented features. This strategy also would have improved user happiness in 3 of the 9 cases. Using the strategy of returning from the method, Dynamic error remediation succeeded in allowing all nine cases to terminate gracefully, with just two cases, JODE and McKoi being unable to do anymore work, and increasing user happiness in 4 of the cases.

Each case in the study is analyzed in more detail in Section 4.

4 The Bug Suite

This section details the method used for finding and creating the suite of programs that contain bugs suitable for testing origin tracking and dynamic error remediation. In particular it details the process of finding programs on SourceForge [19], locating suitable null pointer exceptions in the programs, and creating the test cases to reproduce the null Pointer Exception. To build this suite we follow the guidelines specified by Lu et al. with the exception of having a variety of bug types, as origin tracking and dynamic error remediation are specific to null pointer exceptions. This section also details the case study of the eight null pointer exceptions in six programs that were used to determine the usefulness of dynamic error remediation. The eight null pointer exceptions represent a subset of the 12 original exceptions, the eight exceptions added later to the original suite of four exceptions. JGAP was not a part of the original suite used for origin tracking and was added after testing origin tracking for the purposes of testing dynamic error remediation.

4.1 Finding the Bugs on Sourceforge

Finding Buggy Programs. To test origin tracking and dynamic error remediation, we needed a suite of programs of varying function and maturity, that contain null pointer exceptions. To find these programs, we used SourceForge [19]. SourceForge is especially useful for this because it is a database of open source programs which each contain a bug repository. Open source is an important aspect of the programs used in the suite because once we find a program that contains a null pointer exception, we then need to use the source and the stack trace to analyze the cause of the null pointer exception, as well as determining if there is a fix for the exception.

With Sourceforge there are a number of ways to find programs. The easiest and most useful method in my experience is to use the program search and use the search filter to filter out any results which do not fit the criteria for the suite. Our criteria was any Java programs which narrowed down the search from 94,837 programs to 23,134 programs.

From here we need a method for further improving our chances of finding programs that contain null pointer bugs. There are a number of useful criteria to sort the remaining programs: rank, activity, and downloads. In our experience, the number of downloads represented the most reliable way to sort the programs to maximize the chances of finding a program with a reproducible bug. Downloads represents a good representation of how many people are using a particular program. As Liblit et al. [11] point out, the users of the program bring far more resources for testing a piece of software than the people who designed it. The more users a program has, the more likely it is to have a reproducible bug for our suite. With the list sorted, we then used brute force and went program by program looking for a reproducible bug, in this case a null pointer exception.

Finding Reproducible Bugs. With a methodology for searching through the programs, we now need a methodology for finding reproducible bugs. We examined the bug repository for each program. Depending on the type of bug you are looking for, you might proceed through the bug repository differently. If you are trying to create a suite of bugs that have yet to be solved, you would narrow your search to the open bug reports. In our case, it was important to be able to understand the cause and solution of the null pointer exception so we searched both open and closed bug reports. We proceeded next by searching through the repository by using key words that relate to our bug i.e., ”null”, ”null pointer exception”, and ”npe”.

Once we have only the bug reports of potential bugs, we need to analyze the bug report to determine if the

bug will be useful for our suite. The main criteria for determining if a bug will be useful is if the reporter of the bug or someone commenting on the bug included instructions for reproducing the bug. If the answer to that is no, then the bug is unusable. Unfortunately, this represents the majority of the cases. Only about one in 10 null pointer bug reports was useful in our experience. Often reporters will leave the stack trace assuming the developers can figure out the bug from that, which as noted by Bond et al., is often not helpful for determining the cause of a bug [3].

Creating the Suite. Once a bug was found to be reproducible, the next step was to create a test case. To create a test case, we needed two things, the version of the program that coincided with the bug as well as the instructions for reproducing the bug. We create a test case by following the instructions provided in the report. The final step was to run the test case to see if the bug occurred. If the bug occurred then we successfully found a bug for our suite! We then continued the above process until we had either exhausted the list of programs, or found enough bugs to satisfy our test criteria. Table 4 represents the suite used for testing Origin Tracking, and Table 2 and Table 3 represent the bug suite used for testing dynamic error remediation. The bug suite used with dynamic error remediation is a subset of the bugs from origin tracking because we have not been able to get dynamic error remediation working with eclipse and jython to date. The suite also contains JGAP which was added after testing origin tracking.

4.2 A Case Study for Each Bug

This section details the cause of each bug, if known, the behavior of the program with dynamic error remediation using the strategy of returning from the method, and replacing the null value with an object.

The results of origin tracking for each bug, on the basis of if origin tracking found the source, and if the source was trivial and/or useful, is listed in Table 4.

4.2.1 Evaluation Criteria

For each bug, we evaluate the performance of dynamic error remediation using the following criteria:

Graceful Termination. Does the program progress to a graceful termination, defined as a termination by the program, after the failing exception is handled by dynamic error remediation?

Usefulness. Does dynamic error remediation provide the program with the ability to continue execution, and does the subsequent execution do any meaningful work, or does it have the potential to do meaningful work?

User Happiness. Do the results with dynamic error remediation in place increase user happiness vs the results with the null pointer exception?

Stack Depth/Number of methods returned. If Returning from the failing method was an effective strategy, how deep was the stack at the time of the exception, and how far did the program return before returning correctly?

4.2.2 The Case Studies

Case 1 and 2: JFreeChart: Stacked XY Plot with Lines and Plot Without X-Axis. JFreeChart is a graphing library. If a user-provided program tries to create a Stacked XY Plot with lines using JFreeChart 1.0.2, the program throws a null pointer exception (Bug 1593156). This null pointer exception is the result of the constructor of the graph initializing the lines to null, and then this object never changes. When the graph is drawn, this null value causes a null pointer exception.

The other case in JFreeChart also occurs because of drawing a graph, but the cause is much less interesting than the previous case. The null pointer exception in this case occurs because if a FastScatterPlot is created with null axes then the library code throws a null pointer exception (Bug 1593150). The developers assumed that the users would never create a chart with null axes and thus do not include a check for null axes.

Using the strategy of returning from the method, the user-provided program, in each case, tries to draw the graph which then causes the null pointer exception for the above reasons. Since the test case for each bug was so simple (it only drew the graph), the VM returned control back to the main method, and then the program idled indefinitely due to the graphical nature of the program. For each case, the program terminates gracefully, and in the case of a more complicated program, the program would be able to continue drawing other graphs. These cases illustrate the usefulness of dynamic error remediation using this strategy. Since GUI programs are inherently modular, providing different graphical components, the failure of one component will not affect the other components.

Unfortunately, we were unable to get GTK support working with Jikes-2.9.1 to test these cases with the strategy of injecting an object in place of the null value. We believe that when this case is working, it will show the usefulness of this strategy with GUI componenets as the both graphs should draw, albeit incorrectly, and then the program should continue to show the incorrect graph until the user closes the program.

Case 3: JODE: Exception Decompiling Class. We reproduce a bug in Java Optimize and Decompile Environment (JODE), which is a package that includes a decompiler that decompiles a .class file into the original .Java file if possible. The bug occurs using version 1.1.1 while trying to decompile a particular class. During initialization of an inner class, the information about its outer class is null (Bug 821212). We did not fully understand the cause of the bug because unfortunately we did not have the source of the class file we were trying to decompile.

For JODE, the strategy of returning from the method was useful in allowing it to terminate gracefully, however it does not succeed much further. When the class is finished decompiling (incorrectly because of the error), JODE asserts a verification error then terminates, which is the correct behavior given the faulty output caused by the error. However, dynamic error remediation would be useful for JODE if the behavior upon an assertion error were to move to the next input file upon an error with the current input file, as JODE would continue decompiling the other files correctly.

With the strategy of injecting an object, unfortunately JODE tries to index into a null array, and the current implementation does not handle the case of a null array access.

Case 4: JRefactory #1: Package and Import on Same Line. The next bug occurs in JRefactory, a refactoring tool for Java. A null pointer occurs in JRefactory 2.9.18 when it is given a Java source file that contains either an import as the very first thing in the file, or another declaration occurs on the same line and before an import. Originally the bug was reported to occur when a package declaration and an import declaration were on the same line, but through further analysis motivated by the results of dynamic error

remediation we found that the bug occurred in the above listed cases as well (Bug 973332). The cause of this bug is because the programmer forgot to check the case inside the while loop if an object `lastToken` was null after the while loop finished. The programmer then dereferences `lastToken` which causes the exception.

Returning from the method performs exceptionally well in this case because this method is the correct behavior since the JRefactory method is supposed to remove the last token, and no last token exists. Thus dynamic error remediation allows the program to continue, and continue correctly, also making this the ideal solution to this problem.

Injecting an object also performs exceptionally well. Because the dereference comes during a comparison, when an object is injected, the field it is comparing to another object is null, which causes the condition to return false, which is the correct behavior in light of the fact that the last token does not exist, therefore its kind could not match the kind it was comparing to.

Case 5: JRefactory #2: Invalid Class Name. We discovered the next bug in JRefactory while trying to produce the previous bug. In creating the test case for the previous bug, we inadvertently added .Java to the name of the class which caused JRefactory to throw a null pointer exception. The null pointer exception arises because the program detects the misnamed class and tries to throw a parse error, but the object to print exceptions is null at the point of use, causing the program to throw the null pointer exception. We submitted the bug and the suggested fix to the bug repository of JRefactory (Bug 1674321).

With dynamic error remediation, JRefactory terminates the refactoring and then proceeds to refactoring the next input file. JRefactory terminates gracefully with dynamic error remediation, and correctly terminates due to the original parse exception and then proceeds on to the next input file. In this case, dynamic error remediation is very useful and exhibits the ideal solution. An equally ideal solution would be to replace the Exception Printer object and re-run the instruction, as an Exception Printer needs no initialization, so the program just needs to have a new Exception Printer object instead of null.

With the strategy of injecting an object in place of the null value, this case fails to complete. The exception occurs on an interface method call, which has no known instantiable subclasses at the time of the exception. Therefore there is no object to replace the null value with, and this strategy fails.

Case 6: Mckoi SQL Database: Access Closed Connection. Mckoi SQL Database [13] is a database management system for Java in which we found a null pointer exception that occurs when the user's program tries to execute a query on a database in which the connection has already been closed. The bug report was found on the product's mailing list (Message 02079) that occurs in version 0.93. The cause of the bug is due to the lack of a check for a null database connection in the method that executes the query. This omission causes the null pointer exception to occur, which crashes the program. To recreate this bug, we create a simple test case that closes the connection and then executes a query. It is likely that in a normal program which accesses a database, closing the connection would be less obvious.

With this bug, dynamic error remediation using the method return approach handles the bug ideally. Because the connection is closed, the query will return nothing. Thus by returning null, dynamic error remediation models this correctly. Returning with a null value allows the program to terminate gracefully, although no useful work can be done afterwards, as the connection remains closed.

Using the method of injecting an object does not work with this example because the current implementation cannot instantiate an object and therefore does not have one to replace the null value with.

Case 7: Checkstyle: Empty Default Case. Checkstyle is a program that takes a Java source file as input and checks it for compliance to a coding standard given by an XML file. The null pointer exception in Checkstyle occurs when an input file is passed that contains a default switch case that contains no statements (Bug 1472228). The exception occurs because a variable `lastStmt` is null, and the statement on that line dereferences the null. This case represents a feasible case, which the developers overlooked.

An analysis of the results with the strategy of returning from a method leads to the hypothesis that null is indeed the expected value of `lastStmt` and the developers overlooked this case when analyzing the behavior of the program. Analyzing the method return tells us that if `lastStmt` is null, the method should return false. False is the return of dynamic error remediation when a failing method returns a boolean. Because dynamic error remediation returns what is expected, Checkstyle correctly fixes the program in accordance with the coding standard. Dynamic error remediation handles the bug ideally as the return of false is the correct return given the analysis of the method, this allows the program to continue with the rest of the checks, and then terminate gracefully.

Dynamic error remdiation also handles this bug correctly with the strategy of injecting an object. An analysis of the method reveals that by injecting an object in place of the null, the if statement is allowed to complete with a false condition. Then, because the implementation of this strategy only places the address of the object into the register, making it a temporary fix, the next time `lastStmt` is used it is null again, which is the expected value of lastStmt and the method returns with the correct result. Checkstyle then finishes the rest of its checks, terminates gracefully and outputs the correct results.

Case 8: FreeMarker: JUnit Test Crashes Unexpectedly. FreeMarker is a template engine that provides a way to generate textual output from data. The exception occurs because `wrap()` tries to dereference `defaultObjectWrapper` which was statically initialized to the value of another static field. The programmer assumes that the latter object will be initialized before the former, when this is not the case. The static field, `WrappingTemplateModel.instance` is null when `defaultObjectWrapper` is initialized (Bug 1354173).

For this case, returning from the method is successful at both allowing the program to terminate gracefully, as well as giving the program the opportunity to continue doing other work. Because the JUnit test succeeds, we know that the program terminates gracefully. By inspection of the test program, we see that the program finishes generating the template that would have normally failed, allowing the program to continue doing useful work. Thus for Freemarker, dynamic error remediation is very useful in providing a mechanism for handling the null pointer exception.

Because the implementation of injecting an object in place of the null value includes creating an instance of a class each time it is resolved, a nice, perhaps unexpected, side effect is that FreeMarker does not throw the exception at all when using this strategy. because both classes are initialized at the time where the exception would usually occur.

Case 9: JGAP: Missing Fitness Comparator. JGAP is a Java framework which provides genetic algorithms for computing evolutionary solutions using the principles of natural selection. The exception occurs in JGAP because while computing evolutionary data, a fitness comparator is expected to compare the data however it was never initialized(Bug 1578631).

For JGAP, returning from a method allows JGAP to proceed past the uninitialized fitness comparator, and then it allows JGAP to proceed past the null pointer exceptions which occur each time JGAP tries to compare two solutions using the null comparator. JGAP continues to do more evolution calculations before eventually

terminating. While dynamic error remediation allows JGAP to finish, it is very clear that the results are going to be incorrect, which for this case is not ideal, as the correctness of the output is more important than availability.

JGAP unfortunately does not work with injecting an object in place of the null value as JGAP includes some sort of error checking which does not allow an object of a certain type in JGAP to be instantiated more than once, so JGAP terminates upon detecting that the object has already been instantiated when it tries to create one. While injecting an object does not work for this case, an analysis of the code shows that if JGAP had not included its error checking this strategy would have corrected the problem and allowed JGAP to continue correctly to conclusion. This fact is because the fitness comparator only contains methods and no fields, meaning a new object would have sufficed.

5 Origin Tracking

This section details origin tracking, a modification to the Jikes RVM [10] that uses *value piggybacking* to record and report useful information about null values in Java. Origin tracking reports the location that each null value was assigned at, and maintains this location in the unused bits of the null value. Using this technique, origin tracking is able to maintain the origin of the null until the null is dereferenced, in which case it can then report the origin with the stack trace to provide extra information to users. The following sections overview the motivation, implementation, and experimental results of origin tracking on the suite of programs in Table 4.

5.1 Motivation

Unfortunately for users, programs are rarely bug free. Bugs often lead to the program terminating unexpectedly, leaving the user with little information about the actual bug. This is especially true with a null pointer exception. When a null pointer exception occurs, the program terminates, and upon termination reports the exception that occurred, along with a stack trace. This information alone is often not enough to reproduce or understand the bug for fixing the error, as the origin of the null value is, more often than not, located somewhere other than methods on the stack. It is also a problem for trying to create a bug suite, as this information alone is rarely enough to create a suitable test case (see Section 4.1).

Origin tracking seeks to alleviate this problem for the users and developers of a program by reporting the source location of the assignment of the null value that was dereferenced. With null values that originate off the stack, it is difficult for users to find the source. With the origin, the user and developer can better understand the cause of the null pointer exception by tracing how the value proceeded from the origin of the null value to the dereference.

5.2 Implementation

To report the origin of the null value, origin tracking maintains the location information by using value piggybacking. It represents null by an address in 0x00000000 - 0x07ffffff, using the lowest 27 bits of the null value to store the location. Origin tracking stores the location as a <method, line number> pair in the lowest 26 bits, using the 27th bit to encode one of two ways for storing the location.

With the origins now attached to null values in the lowest 27 bits, origin tracking then redefines program

Program	Lines	Exception description	Origin?	Trivial?	Useful?
Eclipse	2,425,709	Close Eclipse while deleting project	Yes	Trivial	Not useful
JRefactory	231,338	Package and import on same line	Yes	Trivial	Not useful
JFreeChart	223,869	Stacked XY plot with lines	Yes	Somewhat nontrivial	Marginally useful
Jython	144,739	Problem accessing *__doc__* attribute	Yes	Somewhat nontrivial	Marginally useful
Checkstyle	47,871	Empty default case	Yes	Nontrivial	Most likely useful
Eclipse	2,425,709	Malformed XML document	Yes	Nontrivial	Most likely useful
FreeMarker	64,442	JUnit test crashes unexpectedly	Yes	Nontrivial	Definitely 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
JRefactory	231,338	Invalid class name	Yes	Nontrivial	Definitely useful
Mckoi SQL DB	94,681	Access closed connection	Yes	Nontrivial	Definitely useful

Table 4: The diagnostic utility of origins returned by origin tracking in Java. Bug repositories are on SourceForge [19] except for Eclipse [5] and Mckoi SQL Database [13].

operations affected by the new null representations. For null assignments, the JVM to insteads assign the program location defined earlier in this section. For object allocations, the JVM to assigns fields of an object that are assigned null to the program location instead. Finally, the JVM to handles comparisons to null and comparisons to other objects by first bitwise ANDing objects with 0xf8000000, to remove the location in the comparison.

With the modified operations, the origin that is stored in the null value will now be transferred through the program via assignments until it is dereferenced which causes a null pointer exception. The VM then reports the origin by decoding the <method, line number> pair.

5.3 Experimental Results

Origin tracking was tested using the suite of bugs listed in Table 4. Table 4 also shows the usefulness of origin tracking for each bug, along with the triviality of the origin, and whether or not the origin was reported by origin tracking. The triviality of the origin is determined by whether or not the reported origin is located on or off of the stack trace.

For every bug, origin tracking was able to locate the origin of the null that caused the null pointer exception to occur. In 9 of the 12 cases, origin tracking was useful or probably useful, in one of the 12 it was potentially useful, and in two of the 12 it was not useful. For the two that origin tracking was not useful for, the origin was trivial. These results establish that for a bug with a trivial origin, origin tracking is not useful, and for a nontrivial origin, origin tracking is either useful or probably useful. Since 10 of the 12 bugs were deemed to have nontrivial origins, we can see that origin tracking is useful in the majority of the cases.

Origin tracking uses no space overhead because the locations are stored *in place*. The execution time for origin tracking is also very small, being reported at a average slowdown of 3%. Since origin tracking adds instrumentation to the application, the compilation overhead is also a consideration. The average compilation overhead is 10%, with a maximum reported overhead of 12% in `pseudojbb`.

6 Conclusion

While there have been improvements to programming languages and bug detection tools, bugs still escape into deployed code, costing money and reducing availability in a world more and more reliant on it. Bug tolerance methods are used for making programs more tolerant of these bugs so that deployed software can continue past the bugs until the bugs can be fixed by developers.

This case study is intended to provide a proof of concept for dynamic error remediation as a viable method for handling semantic bugs in a non-oblivious, remediating way which is unlike previous bug tolerance techniques. By tolerating semantic bugs, dynamic error remediation increases availability and reliability of deployed programs. The results of the case study show that dynamic error remediation successfully allows all nine cases to reach a graceful termination with at least one of the implemented strategies, as well as allowing 7 cases to continue to do more work. Returning from the method provided the best means of reaching a graceful termination as all 9 cases reached graceful termination, as well as providing the best means for continuing to do more work with 7 cases continuing to do more work. Finally, this strategy increased user happiness a lot in 4 cases, a little in 1 case, and in 3 cases the increase or decrease are dependent on a more complex program, as the test cases were very simple. Inserting an object in place of the null value allowed 3 of the 9 cases to reach a graceful termination, continue to do more work, and increase user happiness, with the results of two cases being unknown due to as of yet unimplemented solutions for those cases and two being unknown due to a lack of GTK support with JikesRVM-2.9.1 and the CS linux machines.

Given these results, especially those of returning from a method, we believe that dynamic error remediation is a viable option for tolerating semantic errors to increase availability and reliabilty, which we believe will have a more important focus as the world becomes more and more reliant on the availability of information and services. While we understand that there are cases where correctness is more important than availability, the results of the case study show that cases do exist where increased availability and reliability can be achieved while minimizing the loss in correctness. For these cases, we provide dynamic error remediation to developers and users as a tool for surviving null pointer exceptions.

7 Future Work

As dynamic error remediation introduces a new idea to the field of bug analysis, there is a large amount of future work to be undertaken. In the more immediate time frame, the current implementation of injecting an object in place of the null value needs to be completed to allow the other 3 unknown cases to be analyzed for completeness of that portion of the study. Also, the implementation of this strategy should be improved to allow the objects to be created as needed and not as the current implementation does, by maintaining an instance of every class seen by the VM during execution.

Other future work includes extending dynamic error remediation to include the other strategies described in Section 3. Given more strategies, and because not every strategy is ideal for every case, we also propose adding analysis which would automatically chose the best strategy for each situation. With more strategies and autonomous strategy selection, dynamic error remediation can be improved to be more robust as well as potentially guarantee correct output with the already improved availability in more situations.

Acknowledgments

I would like to thank Dr. Kathryn McKinley for giving me the opportunity to pursue undergraduate research, and for all of her direction over the past year and a half. I would like to thank Michael Bond and Ben Wiedermann for their help with my research. Thank you, also, Dr. Emmett Witchel and Dr. Gordon Novak for participating in my Thesis Defense.

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. 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.

[3] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. *SIGPLAN Not.*, 42(10):405–422, 2007.

[4] T. M. Chilimbi and V. Ganapathy. HeapMD: Identifying Heap-based Bugs using Anomaly Detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

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

[6] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[7] L. Fei and S. P. Midkiff. Artemis: Practical Runtime Monitoring of Applications for Execution Anomalies. In *Conference on Programming Language Design and Implementation*, pages 84–95, 2006.

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

[9] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

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

[11] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug Isolation via Remote Program Sampling. In *Conference on Programming Language Design and Implementation*, pages 141–154, 2003.

[12] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In PLDI Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

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

[14] National Institute of Standards and Technology. Software Errors Cost U.S. Economy 59.5 Billion Annually, 2007. http://www.nist.gov/public_affairs/releases/n02-10.htm.

[15] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2007. ACM.

[16] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies — a safe method to survive software failure. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Oct 2005.

[17] 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.

[18] N. Rutar, C. B. Almazan, and J. S. Foster. A Comparison of Bug Finding Tools for Java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

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