Changing Java's Semantics for Handling Null Pointer Exceptions

A Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science

Computer Science

by

Kinga Dobolyi

June 2008

Approvals

This thesis is submitted in partial fulfillment of the requirements for the degree of

Master of Science

Computer Science

Kinga Dobolyi

Approved:

Westley R. Weimer (Advisor)

Mary Lou Soffa (Chair)

John C. Knight

Greg Humphreys

Accepted by the School of Engineering and Applied Science:

James H. Aylor (Dean)

July 2008

Abstract

We envision a world where no exceptions are raised; instead, language semantics are changed so that operations are total functions. Either an operation executes normally or tailored recovery code is applied where exceptions would have been raised. As an initial step and evaluation of this idea, we propose to transform programs so that null pointer dereferences are handled automatically without a large runtime overhead. We increase robustness by replacing code that raises null pointer exceptions with error-handling code, allowing the program to continue execution. Our technique first finds potential null pointer dereferences and then automatically transforms programs to insert null checks and error-handling code. These transformations are guided by composable, context-sensitive recovery policies. Error-handling code may, for example, create default objects of the appropriate types, or restore data structure invariants. If no null pointers would be dereferenced, the transformed program behaves just as the original.

We applied our transformation in experiments involving multiple benchmarks, the Java Standard Library, and externally reported null pointer exceptions. Our technique was able to handle the reported exceptions and allow the programs to continue to do useful work, with an average execution time overhead of less than 1% and an average bytecode space overhead of 22%.

Contents

Chapter 1

Introduction

We envision a world where no exceptions are raised; instead, language semantics are changed so that operations are total functions. Either an operation executes normally or tailored recovery code is applied where exceptions would have been raised. As an initial step and evaluation of this idea, we propose to transform programs so that null pointer dereferences are handled automatically without a large runtime overhead. We introduce APPEND, an automated approach to preventing and handling null pointer exceptions in Java programs.

Removing null pointer exceptions is an important first step on the road to dependable total functions in Java, by having specific and well-defined code to execute for both valid and invalid pointer dereferences. Consider the example of a system failure on the USS Yorktown in September 1997:

> A system failure on the USS Yorktown last September temporarily paralyzed the cruiser, leaving it stalled in port for the remainder of a weekend...The source of the problem on the Yorktown was that bad data was fed into an application running on one of the 16 computers on the LAN. The data contained a zero where it shouldn't have, and when the software attempted to divide by zero, a buffer overrun occurred – crashing the entire network and causing the ship to lose control of its propulsion system. [1]

Preventing the divide-by-zero that caused the massive failure that lead the ship to be "dead in the water" [1] would have been preferable. Although this example showcases the potential conse-

quences of an uncaught divide-by-zero exception, null pointer exceptions can be just as severe. A *total function* is one that is well-defined for all potential inputs, possibly using specific recovery actions to handle problems. Having total functions for pointer dereference and division would help to prevent such errors.

Unfortunately, a manual implementation of such functionality is rarely practical. Checking for null pointers by hand is tedious and error-prone [24, 52], especially when pointer values are created by external components or are part of a chain of object references. Though developers do sometimes check for some null pointer dereferences manually, there is no reason this checking cannot be automatically and systematically extended to all pointer dereferences in a program.

1.1 Summary of Technique

APPEND is a two-step transformation on Java code applied at compile time, to prevent runtime null pointer exceptions. First, the program is analyzed to locate possible null pointer dereferences, replacing the human observations with static analysis. Then, APPEND inserts null checks and error-handling code. The error-handling code is specified at compile-time via composable, context-sensitive recovery policies. Recovery policies describe how the program should behave when a null pointer exception is prevented, so that a total function is created for both valid and invalid pointer dereferences. Generated handling code might, for example, create a default object of an appropriate type to replace the null value, skip instructions, perform logging, restore invariants, or some combination of the above. This approach is desirable in web services or dynamic web content, where users interpret the final results with respect to an acceptability envelope [38] and high availability is of paramount importance.

Because program behavior is preserved when no null pointer exceptions are thrown, this approach can be applied to any Java program. APPEND's goal is to transform programs so that null pointer exceptions are avoided and programs can continue executing without incurring a high runtime cost in space or speed. An added benefit with using APPEND is that the checking and recovery actions can be implemented in the object files, rather than the source, thereby reducing code com-

plexity and size, as explained in Chapter 4.

Chapter 2

Background

Null pointer exception management is a logical starting point for changing Java's semantics for exception handling, because of the simplicity and regularity and null pointer exceptions. Null pointer exceptions, while conceptually simple, remain prevalent in practice. They have been reported as the most common error in Java programs [11]. Null pointer dereferences are not only frequent [48], but also catastrophic [49] and are "a very serious threat to the safety of programs" [11]. Many classes of null pointer exceptions can be found automatically by static analyses [25], and they have been reported as one of the top ten causes of common web application security risks [3]. Addressing such risks with fault-tolerance techniques is a promising avenue. For example, techniques that mask memory errors have successfully eliminated security vulnerabilities in servers [40].

Though Java already provides an infrastructure for exceptions, the current state of the language is only a partial solution. Java makes a clear distinction between *checked* and *unchecked* exceptions. The former are included in method type signatures and must be addressed by the programmer; the latter may be ignored without compiler complaint. Unchecked exceptions should also be documented and properly handled by the language, in a systematic and universal manner. Java treats null pointer exceptions as unchecked by default, while APPEND's approach to null pointer prevention is similar to the way Java treats checked exceptions: an undesirable situation or behavior is identified by the programmer, and some error handling code is generated.

One reason null pointer exceptions are not treated as checked by Java is that there are many

potential sources of null pointer dereferences and different recovery situations would have to be embedded in multiple local catch blocks explicitly: a time-consuming and error-prone task. First, it would be difficult to identify, for each null pointer exception that propagated upwards, what kind of recovery code could be applied, without knowing context information. Secondly, Java's current exception handling mechanisms also open up the possibility of a breakdown of encapsulation and information hiding, as implementation details from lower levels of scope are raised to the top level of the program. A solution to null pointer exceptions that is able to prevent or mask them in a way that is both reasonable and accessible to the programmer has yet to be implemented.

Despite their relative simplicity, null pointer dereferences remain a problem in software because they tend to be markers of misuse or misunderstanding of various software components. Creating a generic detection or solution to their existence would be analogous to being able to identify conceptual errors in code. While programming idioms have been examined as conceptual errors [24], generalizing a static analysis tool to catch all potential null pointer dereferences would mean that every object dereference would have to be considered. As a result, not all defect reports from static analysis tools are addressed [24,51]. Programs ship with known bugs [32], and resources may not be available to fix null pointer dereferences.

Fixing null pointer exceptions is also difficult because some programming idioms make static null pointer analyses unattractive. For example, many programs simplify database interaction by creating and populating objects with field values based on columns in database tables (e.g., [5, 9, 26]). The validity or nullity of a reference to such an object depends on what is stored in the database at run-time. Conservative static analyses typically flag all such uses as potential null dereferences, but some reports may be viewed as spurious false positives if there are external invariants requiring the presence of certain objects. For these specific types of examples, in particular, it might be possible to patch invalid null objects with default values. Later in this thesis we will explore specific instances where APPEND's approach to null pointer handling, even with default values, is preferable to other static analyses and recovery methods.

APPEND is a *program transformation* that changes Java's null pointer exception handling by automatically inserting null checks and error-handling code. No program annotations are required,

and developers need not wade through defect reports. Programs are modified according to composable *recovery policies*. Recovery policies are executed at compile-time and, depending on the context, recovery code is inserted that is then executed at run-time if the null checks fail. Recovery policies are conceptually related to theorem prover tactics and tacticals or to certain classes of aspect-oriented programming. If no null values are dereferenced at run-time, the transformed program behaves just as the original program. If the original program would dereference a null value, the transformed program instead executes the policy-dictated error-handling code, such as creating a default value on the fly or not calculating that expression. Previous research has suggested that programs might successfully continue even with discarded instructions (e.g., [39]); we present and measure a concrete, low-level, annotation-free version of such a system, and extend it to allow for user-specified actions.

The idea behind this approach is that null pointer dereferences are undesirable, especially in circumstances where they are involved in non-critical computations where the program is forced to crash if one is encountered. If we had a way of preventing the program from ceasing execution, while logging and performing some sort of recovery code instead of raising an exception, we hypothesize that there are many applications where this behavior would be preferred. Therefore, it becomes possible to check every pointer dereference in the code for nullness, and to include recovery code for every such instance. We argue that this is a practical and preferable way to deal with null pointer exceptions in Java.

Because we intend to check every pointer dereference for nullness, we could have chosen to take advantage of the existing null checking of the Java virtual machine. Given the low overhead of our tool (which we will describe in Chapter 6), we chose to work on the application level instead, to remain portable and not have to rely on a single modified JVM instantiation.

The transformation can be implemented directly atop existing program transformation frameworks and dovetails easily with standard development processes. It can be applied to individual source or class files, entire programs, and separate libraries, in any combination. The main contributions of this thesis are a presentation of our technique (Chapter 4, including our definition of soundness in Section 4.3), our notion of recovery policies (Chapter 5), and experimental evidence

(Chapter 6) to support the claim that our approach can handle null pointer exceptions in practice with minimal execution time overhead and low code size overhead (Section 6.4). We begin with an assessment of the amount of null checking currently taking place in some benchmark examples.

2.1 Null Checking Analysis

Before explaining our program transformation, we will explore null pointer checking in terms of code coverage. By analyzing the number of pointer dereferences, we can come up with an estimate for an upper bound on the number of null checks needed to make all dereferences safe. In addition, we want to examine the amount of null checking already in place by the programmer.

The following code snippet, taken from JBPM, shows a common paradigm used to prevent null pointer dereferences:

```
1 if ( (action!=null)
2       && (action.getActionDelegation()!=null)
3       && (action.getActionDelegation().getClassName()!=null)) {
4   buffer.append(",");
5   buffer.append(action.getActionDelegation().getClassName());
6 }
```

The short-circuit `if` statement is responsible for testing the variable `action` and the results of its nested accessor functions to check for any null values before they may be dereferenced. Ideally null checking should occur at value introduction and creation, rather than at value use, but in practice it is often very difficult, if not impossible, to know in advance where nulls might be introduced. For programs and libraries created by multiple developers, or for values obtained from databases, checking for nullness on use is a standard defensive programming practice.

While the embedded testing of code with null checking is a step in the right direction, its current application in code development is not optimal for two reasons. First, null checking is generally not done systematically to the point where all possible null dereferences are checked, and second, the null checking clutters the code. Ideally, one would want to check all possible values for null dereference, and not clutter the code with the checks. APPEND proposes to achieve both of these

goals with an automated checking at runtime: we propose to obtain the same checking coverage shown in the example above without the explicit three-line `if` statement.

The extent to which manual null checking lacks completeness can be estimated by examining the occurrence of function call chains in Java code. We use *function call chain* to refer to the invocation of a method on an object that may or may not be null; for example:

```
person.getAddress( )
```

or

```
person.getPhoneNumber( ).getAreaCode( )
```

In the second example, both `person` and `person.getPhoneNumber` must be non-null for the code to work correctly, but the nullness of `person.getPhoneNumber` cannot be checked until `person` is verified to be non-null. The *height* of a function call chain is the number of consecutive function calls in the expression; the first example has a height of one and the second has a height of two.

We view compound expressions such as:

```
person.getAddress(var.getName( ))
```

as multiple separate chains — in this case, two chains each of height one.

For perfect defensive null-checking coverage, a call chain of height X requires X separate null checks. We studied the number and height of call chains and null checks present in existing Java programs; Figure 2.1 summarizes the results.

We examined random files from Java benchmark applications (iReport, jbpm, mondrain, neogia, scheduler, squirrel) and noted the number of call chains and the number of null checks associated with those call chains. Of the 463 null checks required for full defensive code coverage, only 15% were actually present. This observation argues for a systematic approach to inserting null checks and the relevant cleanup code. In addition, if all 463 null checks were inserted and each check required one line of code, null checks would account for 13% of the code around the call chains, thus dramatically increasing clutter. The results of our study strongly motivate two key desired properties for APPEND: that it automatically and systematically address all null-pointer dereferences, and that it allow programmers to specify error-handling code without local clutter.

Bench-mark	Call Chains	Required checks (total)	Programmer checks as % of total required	LOC	Required checks as % of LOC
1	36	36	0%	241	15%
2	0	1	100%	284	0%
3	10	14	0%	86	12%
4	14	21	33%	315	4%
5	97	98	0%	207	47%
6	8	12	33%	128	6%
7	9	13	31%	101	9%
8	43	60	23%	319	14%
9	1	1	0%	273	0%
10	12	21	14%	137	13%
11	34	42	12%	230	16%
12	0	1	100%	229	0%
13	4	11	55%	223	2%
14	14	33	55%	217	7%
15	39	61	0%	185	33%
16	13	19	32%	206	9%
17	0	0	100%	171	0%
18	18	19	0%	94	20%
TOTAL	352	463	**15%**	3646	**13%**

Figure 2.1: Measurements of null checking in benchmark samples. *Call Chains* gives the actual number of call chains based on object dereferences in the code. *Required checks (total)* lists the number of null checks required for full defensive programming coverage of those chains; note that this number can be greater than the number of chains when chain height exceeds one. *Programmer checks as % of total required* gives the percentage of null checks required that were already added by the programmer. The *LOC* is the lines of code in each file. The *Required checks as % LOC* column indicates how much of the code around the call chain logic would have to be devoted to null checking to obtain full coverage. The benchmark files used were: 1. ProxyConnection.java , 2. ColumnDisplayDefinition.java, 3. SOSCrypt.java, 4. SOSCommandScheduler.java, 5. CommunityServices.java, 6. CategoryContentWrapper.java, 7. ReportHelper.java, 8. PaymentWorker.java, 9. Enumeration.java, 10. Converter.java, 11. CacheMemberReader.java, 12. Task.java, 13. Timer.java, 14. VariableContainter.java, 15. CrosstabColumnDragged.java, 16. SyntaxDocument.java, 17. Measure.java, and 18. Holiday.java

Chapter 3

Motivating Examples

In this chapter we walk through the application of our technique to a simplified example and to a publicly reported defect. We illustrate the process taken by our automatic transformation and highlight the difficulties in manually handling null pointer exceptions.

In practice it is common to perform null pointer checks before dereferencing an object. Unfortunately, manually inserting null pointer checks is tedious and error-prone. Null pointers can arise from program defects or violated assumptions, but are perhaps more insidious when they result from external sources or components. For example, many database APIs that convert table entries into objects for ease of programmer manipulation may return null objects if the requested entity is not in the database. Runtime dependency on such external systems can significantly reduce the effectiveness of testing in finding potential null pointer dereferences [35, 45]. The following example code illustrates this situation:

```
1 Person prs = database.getPerson(personID);
2 println("Name: " + prs.getName());
3 println("Zipcode: " + prs.getAddr().getZip());
```

In the example above, if the requested person is not in the database or if the database has been corrupted, a null `Person` object will be returned. One standard fault-tolerance approach is to guard statements with non-null predicates:

```
1 Person prs = database.getPerson(personID);
2 if (prs != null)
```

```
3     println("Name: " + prs.getName());
4   if (prs != null && prs.getAddr() != null)
5     println("Zipcode: " + prs.getAddr().getZip());
```

This way, if a valid `Person` is returned, the information is printed out normally. If a null pointer is returned, whether as a valid part of the program API or as an invalid record from the database, the null pointer dereference will be prevented.

Note that even when a valid `Person` object is returned, the `Address` object within the `Person` may be null, and must also be explicitly checked. While this example is for an object from a database, any value that is dereferenced could be a null pointer, and should be checked to avoid a null pointer exception (NPE). The number of NPEs encountered in practice and the research devoted to preventing them is a testament to the lack of consistency of null pointer prevention [25]. At the same time, manually placing checks in the code is not only time-consuming and error-prone, but can also make the code difficult to read and increase code complexity.

One real-world example of problematic handling of NPEs comes from JTIDY, a tool for analyzing and transforming HTML. This example is taken from a bug report submitted by a user on a public mailing list [37]. In the code below, the NPE occurs on line 36:

```
30  Doc xhtml = tidy.parseDOM(in, null);
31  // translate DOM for dom4j
32  DOMReader xmlReader = new DOMReader();
33  Document doc = xmlReader.read(xhtml);
34
35  Node table = doc.selectNode("/html/body");
36  System.err.println("table:" + table.asXML());
```

In some instances the `table` returned from `selectNode` is null, but it is always dereferenced without being checked. In this case it would be advantageous to guard the deference with a null pointer check. In Chapter 6 we show how APPEND is able to automatically prevent the NPE from being raised in this example, allowing JTIDY to do other useful work even if the `"/html/body"` `Node` cannot be retrieved.

As these examples show, recovery from null pointer exceptions can often be handled in an

idiomatic way. For example, developers trying to display the fields of a partially initialized object may be satisfied with displaying only the available fields. For such applications, preventing the runtime NPE by either printing a null value or skipping the print statement is likely preferable to crashing the program. The acceptability of such idiomatic behavior is determined by context, however, which is why APPEND allows programmers to specify different recovery policies for different program logic.

We must also consider situations in which it is not possible to recover from, or not known how to recover from, a null pointer exception in a useful way. In such cases logging may be a reasonable compromise: often times masking and reporting an error is preferable to crashing the program; denials of service are avoided and the application continues to work at some degraded level of service. Because it is impossible to know, in general, what kind of an effect masking NPEs might have, APPEND supports recovery actions that are programmer-specified. This allows programmers to customize recovery for NPEs to individual programs if additional knowledge or invariants about program behavior are available. Programmers can also fall back on APPEND's default NPE prevention behavior, which we will describe in Section 4.2.

Chapter 4

Proposed Technique

Our goal is to prevent null pointer exceptions in Java programs in a way that avoids failures without incurring a high cost. In doing so, we are taking the first step towards changing Java's semantics for exception handling to a total function, where specific code is executed for both normal and exceptional cases. We specifically target areas where producing some output is better than having the program crash. For example, an e-commerce application could lose customers and revenue if it fails to display a webpage because of an NPE raised somewhere in the back end. Perhaps the webpage was displaying product information from a database, and the database contained null values as in the example in Chapter 3. Similarly, it would be undesirable for a vital system to crash entirely due to an obscure NPE not along a critical path of execution (see Chapter 1). We propose that the application prevent the NPEs, avoid crashing, and continue to do useful work.

APPEND addresses null pointer exceptions in an automatic manner by transforming programs. To be practical, most such transformations cannot require user annotations and must not incur high run-time overhead costs. We propose a source-to-source (or bytecode-to-bytecode) analysis and transformation as part of the compilation process. For maximal ease-of-use the transformations can be applied to bytecode object files, so as not to clutter source code with visible null checks. In situations such as debugging, where having the source code and the bytecode align is of paramount importance, the transformation can also be applied at the source level and the resulting code with additional null checks can be compiled as normal. Although it may seem strange to have code

that, from the developer's perspective, is only added to the object file, passing a program through APPEND's transformation is analogous to double-bagging groceries; we are interested in maintaining some liveness properties in cases where the program itself has broken, and the wrapper around it prevents the errors from being visible. In addition, many other high-level language constructs, such as the `synchronized` keyword in Java, result in under-the-hood code generation (e.g., lock acquisition) that is not visible at the source level. As an example from one application area, when combined with applications that use databases, an APPEND-transformed program is able to separate the concerns of database information integrity with respect to null values into a separate shell around the program.

There are two key steps in our technique. First, in the analysis phase, a set of potential null pointer dereferences is determined. Second, in the transformation phase, a null check is inserted to guard each such potential dereference. The transformation takes place according to a user-supplied top-level recovery policy. The top-level recovery policy uses context and location information to compose and query lower-level policies at compile-time. Each policy transforms the program and inserts error-handling routines that are executed at run-time if the null checks fail. The analysis and transformation are carried out on a standard intermediate representation. We use the SOOT transformation and analysis framework in our prototype implementation [47].

Our technique takes as input an unannotated program, a global recovery policy, and optionally a number of other context-specific recovery policies. After the all potential null pointer dereferences are identified, the program is transformed according to the global recovery policy; a null check is used to guard each potential null pointer dereference, and depending on what the recovery policy states, recovery code is inserted for each null check in the case the check fails.

4.1 Finding Potential Null Pointers

The number of dereference sites we identify affects both the completeness and the overhead of our approach. Flagging all pointer dereferences could lead to unacceptable levels of overhead from inserted checks. Flagging too few dereferences may prevent actual NPEs from being guarded.

For our purposes, the preference is to flag more pointer dereferences rather than less, because the checking code will typically only be visible in the object file and the overhead will be low (see Chapter 6). However, because our tool could be applied at the source-level directly, we must still make an effort not to flag extraneous instances of dereferences where we know it is impossible that a null value could exist at run-time.

For this reason, we employ a conservative flow-sensitive intraprocedural dataflow analysis to statically determine if an expression is non-null. Expressions that are not known to be non-null are flagged for transformation. Some types of assignments and object dereferences are assumed to always be correct:

1. We do not flag expressions that our dataflow analysis determines can never be null.
2. We do not flag the results of constructors, such as `new Person()`, which, by Java convention, return a valid object or raise an exception.
3. We do not flag global field accesses, such as `System.out`.
4. We do not flag static function calls.
5. We do not flag array accesses, such as `p[i]`, and view array bounds check elimination as an orthogonal research problem.

Any false negatives that would arise through the use of APPEND would result from the five assumptions made here. If perfect null-pointer dereference coverage is desired, those five cases can be flagged and instrumented as well at the cost of additional code size and run-time overhead.

A more precise interprocedural analysis would result in lower overhead in transformed programs. However, analysis time is also important for our technique if we propose to use it as part of the compile chain. Recent work has made context-sensitive flow-sensitive analyses more scalable (e.g., [19]), but we chose a flow-sensitive intraprocedural analysis for performance and for predictability. Java programmers are already used to simple and predictable analyses, for example in Java's *definite assignment* rules, and understanding the transformation simplifies reasoning about

and debugging the transformed code. Predictability in the analysis used by APPEND is also important in situations where programmers are writing their own recovery policies and expect potential NPEs to be flagged in a certain way.

4.2 Error Handling Transformations

Raising an exception or otherwise terminating the program represents the current state of affairs. APPEND improves on the state of the art by inserting null checks guarding every dereference that has been flagged as potentially null. However, we must also insert behavior in the case where the check fails. The simplest solution would be to simply skip any offending statement that would have thrown a NPE, and continue with execution. However, such a proposal could result in cascading errors where the program is unable to recover in a meaningful manner. APPEND improves upon this by taking specific actions when a NPE is prevented. Our technique is modular with respect to user-defined recovery actions.

As a concrete example of an error-handling policy, we consider inserting well-typed default values. If a null value would be dereferenced we replace it with a pointer to a default initialized value of the appropriate type. We obtain such values by calling the default constructors for the given class; this policy is only applicable if such a default constructor is available for the type under consideration. Although the default object value is probably not what the programmer intended, it may still allow useful work to be conducted. Consider the following pseudocode:

```
1  r6 = virtualinvoke r4.<java.util.Vector:
2      java.lang.String toString()>();
```

If the value of `r4` may be null, then a check would be placed before this line of code to prevent a null pointer dereference. If `r4` is of type `Vector`, the transformed code would be:

```
1  if (r4 == null)
2      r4 = new Vector();
3  r6 = virtualinvoke r4.<java.lang.Vector:
4      java.lang.String toString()>();
```

In the case of `java.lang.Vector`, a default constructor exists for the object, which is used if `r4` was null. In this manner `r4` is sure to be non-null before it is dereferenced, thereby avoiding the NPE. In addition, if `r4` is subsequently referenced without any intervening assignments to it, no additional checks are necessary. This eliminated the cascading-error effect for the object that may have resulted if we only skip the offending statement. In this specific example of code, we do not know what the value of `r4` should have been: it may have been returned from some other component or library. Regardless, after we apply APPEND's recovery policy, we are able to use the `Vector` object normally, adding values to it, resulting in useful work. It may have been the case that `r4` was supposed to be an empty `Vector`, instead of a null value, in which case APPEND's recovery actions have actually corrected the code. Although this interpretation is highly optimistic, with some insight into program behavior by the developer, tailored recovery actions may exist that are able to result in this kind of elegant masking and recovery from NPEs. In Chapter 5 we categorize and describe possible recovery policies in more generality.

4.3 Soundness

We cannot sacrifice preserving correct program execution for the sake of a safety wrapper. Our notion of soundness is that the transformed program should behave exactly as the original program behaves in cases where the original program would *not* produce a null pointer exception. If a NPE would be raised we apply the appropriate error-handling behavior. To implement this definition of soundness, some assumptions about program behavior are necessary. First, we explicitly assume that programs do not rely on NPEs for uses beyond signaling errors (e.g., programs do not use `try` and `catch` with NPEs as non-local `gotos`). This assumption avoids the situation where APPEND's transformation inserts a null check and recovery code around an NPE that was meant to be caught in a different part of the code. While it may be possible to detect or annotate such cases, we argue that the use of NPEs in such a manner is generally bad programming practice and unlikely to be encountered in the real world [52].

We further assume that the user-specified error-handling and recovery code will result in accept-

able behavior. This expectation is riskier than the first one because we cannot know what kind of Java programs APPEND will be applied to. A tradeoff exists between using APPEND and addressing all NPEs automatically, versus writing and directly or formally verifying that code is correct. APPEND is an attractive alternate when the cost of verification is high (e.g., when the code base is large, when annotations are not available, when high test coverage is costly, etc.) and it can be used to supplement incomplete manually-added null-pointer checks.

Soundness is thus dependent on the user-specified error handling code. For example, in the particular case of default constructors, we assume that referencing the default object will not have unintended, permanent side effects beyond the scope of program execution, such as storing the result of a computation involving these default values back in a database. Such assumptions are common for domain-specific recovery actions [4, 31, 38, 40], but, admittedly, may result in unexpected or unintended consequences. Although we cannot offer a foolproof solution for such a situation, we believe that careful policy construction, combined with logging functionality will minimize the risk of such unwanted situations, and will allow for directed debugging efforts in the rare instances when any APPEND inserted recovery code is called.

Chapter 5

Error-Handling and Recovery Policies

Chapter 4 discussed how APPEND locates potential null pointer dereferences. In this chapter, we describe a framework for user-specified, composable recovery policies that are applied at *compile-time* to instrument the code with context-specific recovery actions. At a high level, a recovery policy is meant to identify certain classes or idioms of null pointer dereferences, and assign specific actions to each type of instance. At compile time, a location of a potential NPE in the source code is matched with a NPE pattern or idiom, and appropriate recovery code is then applied.

A *recovery policy* is a first-class object that is manipulated and executed at compile-time and adheres to a particular interface. Each recovery policy has a method `applicable` that takes as input the program as a whole and the location of the potential NPE and outputs a boolean indicating whether that policy can be applied to that location in that context. Here the *context* represents the standard information that a compiler or source-to-source transformation would have available (e.g., class hierarchies, abstract syntax trees, control flow graphs) and the *location* gives the particular statement or expression that contains the potential error. The `applicable` method is responsible for mapping actual flagged potential NPEs to the NPE idioms defined in the policy. Each policy also has an `apply` method that takes as input the program as a whole and the location of the potential NPE and outputs a transformed program that has been adapted to follow the recovery policy at that location. The `apply` method is responsible for inserting recovery code after a particular potential NPE in the code has been identified as being of a certain pattern.

Input: The program context C and an error location L.

1: **if** the dereferenced object at L has a policy P_1
   $\wedge\ P_1$`.applicable`(C,L) **then**
2: **return** P_1`.apply`(C,L)
3: **else if** the context class at L in C has a policy P_2
   $\wedge\ P_2$`.applicable`(C,L) **then**
4: **return** P_2`.apply`(C,L)
5: **else if** the context method at L in C has a policy P_3
   $\wedge\ P_3$`.applicable`(C,L) **then**
6: **return** P_3`.apply`(C,L)
7: **else**
8: **if** `logging.applicable`(C,L) **then**
9: $C,L \leftarrow$ `logging.apply`(C,L)
10: **end if**
11: **if** `constructor.applicable`(C,L) **then**
12: $C,L \leftarrow$ `constructor.apply`(C,L)
13: **end if**
14: **return** (C,L)
15: **end if**

Figure 5.1: An example global recovery policy. This policy checks the dereferenced object and the enclosing class for an overriding policy. If no such specific policy is found, it applies both the `logging` and `constructor` policies.

Figure 5.1 shows an example skeleton for a recovery policy. One example of a recovery policy is the default constructor insertion described in Section 4.2, shown on lines 11–13. That policy is `applicable()` when the type under consideration has a default constructor with no arguments. A `logging` policy is another example (lines 8–10): its `apply()` method inserts calls to a logger and it is `applicable()` whenever the enclosing context is not that logger class (i.e., to avoid infinite recursion at run-time). As a final example, a particular `skip` policy's `apply()` function elides the problematic computation and it is `applicable()` if the location under consideration is not a `return` statement, so as not to propagate likely design errors.

A recovery policy for a specific pattern of NPE can be global or it can be associated with a particular class, both as a subject and as a context. For example, a *global* policy of "use a default constructor" could insert a default constructor in every situation in the code where one exists for a potential NPE. A *local* policy specific to a `DatabaseObject` class could specify that no default object that was created by APPEND should ever be written back to the database. In this example, the global and local policies can be used together through the `applicable` and `apply` methods.

Recovery policies can query and compose the actions of other recovery policies. Our notion of composable recovery policies is inspired by the cooperating decision procedure and tactical approach used in many automated theorem provers. In this context, decision procedures (or abstract interpreters) for separate areas, such as linear arithmetic and uninterpreted function symbols, work together on a common substrate to soundly decide queries that involve both of their domains [36]. In interactive theorem proving, proof obligations in the object language can be manipulated and simplified by *tactics* (see e.g., [21, 23, 41]), programs written in a metalanguage. Tactics can be composed using combining forms called tacticals, allowing users to express notions such as "repeat" and "or else". Just as a theorem prover tactic might embody a notion such as, "try to instantiate universally-quantified hypotheses on in scope variables, and if that does not work try algebraic simplification", a recovery policy in our system might embody a notion such as, "try to instantiate the default constructor for this object, and if that does not work try to log the error and continue."

5.1 Policy Granularity

The recovery policies must be robust enough to handle different combinations and priorities of recovery actions. At a minimum, APPEND requires the user to provide a global recovery policy or use the default ones furnished. The default policy uses default constructors wherever they are available, or when they do not exist, the offending statement is not executed.

Individual classes and contexts can be annotated with specific recovery policies if desired. Example pseudocode for the `apply` function of a global recovery policy is given in Figure 5.1. At compile-time, during program analysis and transformation, we invoke `global.apply()` on each potential null pointer dereference. The resulting modified code is the final result of our source-to-source transformation. Because we do not change any user-provided null checking functionality already implemented in the source code, APPEND will not override such null checks and recovery instances because they will already by flagged as not-null by our static analysis. As mentioned earlier, the only situation where APPEND's transformation can interfere with "correct" code is when NPEs are used in an unexpected way, in conjunction with `try-catch` blocks. If the program already contains null checks, APPEND's static dataflow analysis (see Section 4.1) will not flag them as potential instrumentation points.

The example global policy in Figure 5.1 gives priority to policies associated with the potentially-null object and with the surrounding class, by checking for this case first in line 1. As an example of the former, a particular application might require that all NPEs associated with GUI `Widget` objects be handled by recreating the default widget set and redrawing the application, rather than by creating a newly-constructed and unattached widget and operating on it.

An application might also associate a policy with a class context. In Figure 5.1, such a situation is given lower priority than policies associated with the object itself, by performing this checking at line 3. For example, in a `UserLevelTransaction` class, any null pointer error encountered might be replaced by "`throw new AbortException()`" since the caller presumably knows how to handle transactional semantics. Policies might also be specified at the method level, as shown on line 5 in Figure 5.1; a particular method expected to return a value might make a best-effort

substitution and return. Sidiroglou et al. have examined various heuristics for determining an appropriate return value for a non-void function [44]. In general, attempts that stop the execution of a block or function when a NPE is prevented are variations of fail-stop computing at different levels. It is important to note that in our system, the code for these halting actions is stored with the policy and is present in the transformed code but not the program source code.

5.2 Data Structure Consistency

Previous sections have shown that skipping one or more statements that depend on the dereferenced value may be reasonable in some circumstances (e.g., if the value is merely being printed). An orthogonal approach to such fail-stop options is to enforce data structure consistency. The program may be in an unsafe state when the NPE is prevented and the transformed code is executed instead. Local handling of errors may have unexpected effects on the rest of the program if important invariants are not restored. For example, an object created by default in our `constructor` policy might be written to a database that expects post-processed, validated objects. In this situation, logging alone of the execution of the recovery code may not be enough to satisfy the properties of the application. Many proposals exist for using user-defined or computer-generated constraints (e.g., [15, 17, 18]) on data structures in the program or database to enforce consistency. A simple recovery tactic to prevent cascading errors in such cases would be to prevent APPEND from persisting any recovery-generated objects in the database.

If such constraints were provided as part of the policy, they could be used to transform the code in such a way that the invariants are maintained. Figure 5.2 shows how a class-specific policy might make additional changes to the code to enforce that only objects matching a particular *invariant* were written to the database. A simple conservative dataflow analysis could be used to find all of the database write statements that the potentially-null object might reach. Only those write statements are then guarded with invariant checks. In practice such a policy would benefit from dead-code elimination or other ways of preventing the insertion of duplicate checks.

The user may also be able to specify context-based, rather than object-based, recovery actions

Input: The program context C and an error location L.
1: **if** `other_policy.applicable`(C,L) **then**
2: $C,L \leftarrow$ `other_policy.apply`(C,L)
3: **end if**
4: **for all** database writes $W(x)$ reached by L **do**
5: $C,L \leftarrow$ replace $W(x)$ by "if *invariant*(x) then $W(x)$ else throw new DatabaseException()"
6: **end for**
7: **return** (C,L)

Figure 5.2: An example class-specific recovery policy that maintains an invariant. This policy recovers from NPEs in objects that can be stored in a database. The "if *invariant*(x) ..." code is added at compile-time and executed at run-time. The `other_policy` represents any other policy that might be composed with this one, such as the `constructor` policy from Chapter 5.

related to object consistency. Context at the class level, as opposed to task blocks as described by Rinard [38], are a lower-level version of compartmentalization. For example, the corruption of an object could imply, based on the policy, that no operations be performed with that object, such as passing it as a parameter to a function. This would involve a context-sensitive disabling of execution associated with the corrupt object at runtime.

Chapter 6

Experimental Results

Although source code complexity need not increase with our transformation, bytecode size, running time and utility must be considered. To address these issues, we have conducted several experiments that demonstrate APPEND's:

- effectiveness at preventing NPEs in sample code
- effectiveness at preventing NPEs in the Java Standard Library
- effect on running time and class file size

To provide a baseline for measurement, our experiments used our default policies: if the `constructor` policy from Section 4.2 is applicable (i.e., if the dereferenced object has a default constructor), we apply it. Otherwise, if the `skip` policy from Chapter 5 is applicable (i.e., if the statement under consideration is not a `return`), we apply it. Otherwise we do nothing. In our experiments default constructors were unavailable 65% of the time, and thus this policy *did* involve making compile-time decisions about which transformation to apply.

6.1 Example Transformation

Before illustrating how APPEND can be applied to error prevention and recovery situations, we will walk through an example program transformation to show where null-checks are inserted.

The following code is taken from `AbstractDocumentOutputHandler.java` of SKARINGA version `r3p7`, a Java-XML binding API. The class contains three functions, one which appends the namespace declarations from some XML schema and instance to the attributes of an element, and two others that set and get properties contained in a private `HashMap` object. This code already contains a null check on line 31 for `Attributes attrs`; we will show how APPEND does not interfere with this functionality. On the other hand, the `String name` on line 44 could potentially be null, as it is a parameter to the `appendNamespaceDeclarations` function, and is not checked before being dereferenced.

Original Application Code

```
1  package com.skaringa.javaxml.handler;
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  import org.xml.sax.Attributes;
7
8  import com.skaringa.javaxml.impl.NSConstants;
9
10 /**
11  * An abstract base helper class for DocumentOutputHandlers.
12  */
13 public abstract class AbstractDocumentOutputHandler
14   implements DocumentOutputHandlerInterface {
15
16   private Map _propertyMap = new HashMap();
17
18   /**
19    * Append the namespace declarations form XML schema and instance
20    * to the attributes of an element.
21    * @param name The name of the element.
22    * @param attrs The attributes.
23    * @return The attributes extended with the namespace declarations.
24    */
25   protected final Attributes appendNamespaceDeclarations(
26     String name,
27     Attributes attrs) {
28
29     AttrImpl attrImpl;
30     if (attrs == null || attrs.getLength() == 0) {
31       attrImpl = new AttrImpl();
32     }
33     else {
34       attrImpl = new AttrImpl(attrs);
35     }
36
37     // decls for both schemas and instances
38     attrImpl.addAttribute(
```

```
39          "",
40          "",
41          "xmlns:" + NSConstants.SCHEMA_NS_PREFIX,
42          "CDATA",
43          NSConstants.SCHEMA_NS_NAME);
44        if (!name.equals("xsd:schema")) {
45          // decls for instance only
46          attrImpl.addAttribute(
47            "",
48            "",
49            "xmlns:" + NSConstants.SCHEMA_INSTANCE_NS_PREFIX,
50            "CDATA",
51            NSConstants.SCHEMA_INSTANCE_NS_NAME);
52        }
53
54        return attrImpl;
55      }
56
57      /**
58       * Set the properties of this DocumentOutputHandler.
59       * @param propertyMap The properties.
60       */
61      public final void setProperties(Map propertyMap) {
62        _propertyMap = propertyMap;
63      }
64
65      /**
66       * Get the properties of this DocumentOutputHandler.
67       * @return The properties.
68       */
69      public final Map getProperties() {
70        return _propertyMap;
71      }
72  }
```

The following shows the raw disassembled bytecode code after the program has been transformed by APPEND. The programmer-implemented null checking on line 30 of the original code has remained unaffected in the APPEND version, and appears on line 22. The new null checking functionality for the `name` variable from line 44 takes place on lines 34–37; first `name`, which corresponds to `r1`, is checked for null, and if this check fails, APPEND calls the default constructor for the `String` class. After line 35, the `name` variable (`r1`) is guaranteed not to be null for the rest of the program (unless it is re-assigned), and can be safely dereferenced.

Transformed Code

```
1  package com.skaringa.javaxml.handler;
2
3  import java.util.HashMap;
4  import org.xml.sax.Attributes;
```

```
5  import java.util.Map;
6
7  public abstract class AbstractDocumentOutputHandler implements
8    com.skaringa.javaxml.handler.DocumentOutputHandlerInterface
9  {
10     private Map _propertyMap;
11
12     public AbstractDocumentOutputHandler()
13     {
14         _propertyMap = new HashMap();
15     }
16
17     protected final Attributes appendNamespaceDeclarations(String
   r1, Attributes  r2)
18     {
19         AttrImpl r3=null;
20         label_0:
21         {
22             if (r2 != null && r2.getLength() != 0)
23             {
24                 r3 = new AttrImpl(r2);
25                 break label_0;
26             }
27
28             r3 = new AttrImpl();
29         } //end label_0:
30
31
32         r3.addAttribute("", "", "xmlns:xsd", "CDATA",
33           "http://www.w3.org/2001/XMLSchema");
34         if (r1 == null)
35         {
36             r1 = new String();
37         }
38         if ( ! (r1.equals("xsd:schema")))
39         {
40             r3.addAttribute("", "", "xmlns:xsi", "CDATA", "
41               http://www.w3.org/2001/XMLSchema-instance");
42         }
43         return r3;
44     }
45
46     public final void setProperties(Map  r1)
47     {
48         _propertyMap = r1;
49     }
50
51     public final Map getProperties()
52     {
53         return _propertyMap;
54     }
55 }
```

As a final note, the raw disassembled bytecode shown above is not meant to be read by devel-

opers. We reveal the transformed code as source code only to demonstrate the changes APPEND is making.

6.2 Examples from Application Programs

In this section we show how APPEND can be applied to real-world examples of NPEs. We obtained NPE examples from bug repositories and program support forums. After locating a NPE example and verifying that it could be reliably reproduced, we applied our transformation. We then executed the resulting files, making sure that the NPE was no longer raised.

Returning to the JTIDY example described in Chapter 3, the output of the original program raised an NPE on line 36 due to the following initialization of the `table` variable:

```
35  Node table = doc.selectNode("/html/body");
36  System.err.println("table:" + table.asXML());
```

After passing the test file through APPEND, we obtained this output from line 36:

```
table : null
```

Even though the `selectNode` function at line 35 returns a null, APPEND is able to prevent the NPE while still allowing the `println` statement to execute.

The previous example showed how APPEND can prevent NPEs arising from unexpected or unknown behavior of function calls. NPEs are common in practice, and we had no trouble locating a second defect report [29] for JTIDY related to this code:

```
18  ObjectInputStream in = new ObjectInputStream(
19      new FileInputStream("doc.ser"));
20  Document newDoc = (Document)in.readObject();
21
22  newDoc.getRootElement().addElement("TEST");
```

Here, an NPE on line 22 is caused by behavior in other parts of the program; `newDoc` is not properly initialized, and an element cannot be added to it as above. After running the code sample

through APPEND, the NPE is no longer raised and the result is sensical. Again, APPEND is able to handle the fault and allow execution to continue.

6.3 Java Standard Library Examples

APPEND can also help prevent NPEs in library files. An incremental benefit can be gained by transforming standard libraries or untrusted third-party components, even if an organization is unwilling to transform its primary codebase. This is important because often only bytecode is available for third-party components, making it difficult to be sure what kind of guarantees can be made about the external software. Rather than cluttering the in-house source code with checking functionality, APPEND allows the external library itself to be transformed, even without source code.

We demonstrate this approach on a defect in the Java Standard Library, version 1.1.6 (Sun Developer Network bug ID 4191214). The defect itself lies in the library's `URL` class. The bug report included sample code to elicit the NPE by accessing a `Vector v1` of five `URL`s:

```
1 System.out.println(v1.indexOf( new
2   URL("file",null,"C:\\jdk1.1.6\\src\\test"
3                   + i + ".txt")));
```

The uncaught exception in this example originated from the `hostEqual` method of the `URL` class in the library, which was called form the `equals` method of `URL`, which was itself called by the `indexOf` method of the `Vector` library class. After transforming the library with our technique, the `hostEqual` function no longer raises an uncaught exception, and the overall output is a correct printout of the indices of the `URL`s in the `Vector`. Interestingly, the fix suggested by the defect reporter involves checking that the values passed in to `hostEquals` are not null before they are dereferenced, which is exactly what APPEND implements.

These three examples in Section 6.2 and Section 6.3 show that APPEND is able to prevent real-world NPEs at both the application and library levels, even with a simple recovery policy of calling default constructors, or skipping statements when no default constructor is called. Experiments in the next section show that converting all classes and libraries used incurs little overhead. Ideally,

APPEND would be applied to the entire source package and all libraries, but as demonstrated, an incremental benefit can be observed by transforming even a single file.

We conjecture that using APPEND could improve development efficiency by decreasing source code complexity; instead of having the user patch their source code with manual null checks, they could encapsulate recovery policies into a separate file and rely on APPEND to modify the bytecode directly. In our experimental benchmark applications, only 5% of the null checks required by our tool were already present in the source code. We obtained this number by counting the number of null checks originally in the code, and comparing it to the total number of null checks after the code was instrumented with APPEND. This result is in line with Figure 2.1, implying that around 90% of potential null checking is not taking place.

To gain an idea of the number of extraneous null checks we produce, we randomly sampled transformed files and looked for false positive null checks. A false positive null check occurs when a human could easily verify that the value would never be null. For our three benchmark applications (described in Chapter 6), we found that in the two larger applications, none of our APPEND inserted null checks were obviously useless, while for the smallest application about 20% of our checks could be considered false positives. The cost of false positive null checks is increased bytecode size and increased runtime overhead. We will show in the next section that our overhead is quite low; we thus conclude that our false positive rate is acceptable.

6.4 Performance and Overhead

Because APPEND inserts code into class files for null checking and recovery, to be usable it must have only a minor impact on code size and execution time. Using two separate benchmark suites we compared the running time and bytecode size of unmodified programs as well as programs subject to our transformation. We measured the performance of both of our usage models: transforming the library, and transforming the application.

To measure the impact of transforming the library, we converted classes in Java's `lang`, `net`, `io` and `util` packages with our prototype tool. We then ran the benchmark programs against the

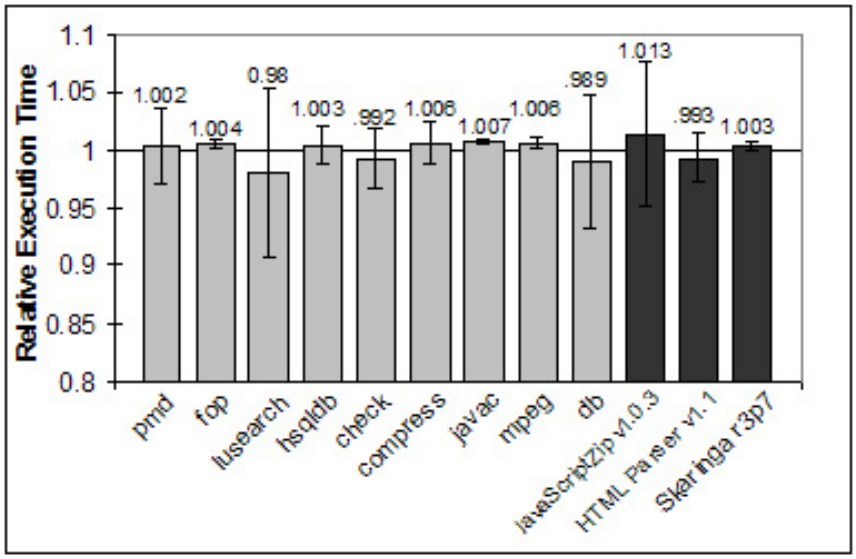

Figure 6.1: Runtime overhead on DaCapo, SpecJVM and application benchmarks. Each column is separately normalized so that 1.0 is the unmodified execution time. Higher values indicate slowdowns. The nine light columns on the left shown times for unmodified DaCapo and SpecJVM benchmarks run against a transformed standard library. The three dark columns on the right are transformed applications run against a transformed library. The error bars represent standard deviations from twenty trials.

unmodified library and against our transformed library. We used the April 30, 2007 build of Apache Harmony JRE, an independent implementation of the Java SE 5 JDK [2].

We used benchmark programs from the DaCapo [7] project, a benchmark suite intended for Java that uses open source, real-world applications with non-trivial memory loads, as well as programs from SPEC JVM98. Figure 6.1 summarizes the results, reporting the average of twenty trials (the eight lighter bars on the left). Each program is separately normalized so that 1.0 is the runtime with the unmodified library; higher numbers indicate slowdowns. In these experiments the average slowdown was less than 1%.

We also measured the overhead of our technique when both the program and the library are transformed. We selected three popular open source applications: JAVASCRIPTZIP version 1.0.3, a web application optimizer; HTMLPARSER version 1.1, an HTML front-end; and SKARINGA version `r3p7`, a Java-XML binding API. All three were run out-of-the-box using the standard library, and those running times were compared to versions where both the applications and the library had

Benchmark Program	Null Checks		Increase
	Normal	With APPEND	
JAVASCRIPTZIP	9	9932	1100x
HTMLPARSER	170499	623361	3.66x
SKARINGA	371	1732	4.66x

Figure 6.2: Increase in the number of null checks in the final code by three benchmarks on their indicative workloads. The null check columns give counts obtained by instrumenting both the original program and the APPEND-modified program at the bytecode level to record null checks before they are made.

been converted by APPEND. Figure 6.1 shows the average execution time for twenty trials of each benchmark in rightmost dark gray bars, with an average slowdown for the three applications-plus-libraries of less than 1%.

Though the average slowdown for our benchmarks was less than 1%, the number of null checks inserted by APPEND and applied at runtime proved to be a substantial increase over the base amount of checking performed by the unmodified programs. Figure 6.2 summarizes the number of null checks that were inserted for three benchmarks at runtime. For the two larger benchmarks, there was an average increase of three times the base amount in the number of null checks the code after being transformed with APPEND, which did not contribute significantly to the runtime slowdown. JAVASCRIPTZIP, the benchmark that showed the greatest runtime slowdown, performed over a thousand times more null-checks when transformed with APPEND.

To ensure that the inserted null checks were actually being executed at run-time, we further instrumented the programs to count the number of times our null checks were executed, versus the number of times user-provided null checks were executed. Figure 6.3 shows the number of times a null check was called by the program for both APPEND-inserted and user-inserted guards. In each case the APPEND-transformed program executed at least five times as many null checks as the original.

Despite the static and dynamic increase in null checks, the average runtime overhead for APPEND-transformed code was only 1.3%. From these three experiments we can conclude both that our transformation is actually affecting the program, in that many additional null-checks are

Benchmark Program	Executed Null Checks		Normal as % of Total
	Normal	With APPEND	
JAVASCRIPTZIP	0	19848	0%
HTMLPARSER	190384	1146002	14%
SKARINGA	296	1360	18%

Figure 6.3: Increase in the number of null checks executed at *run-time by* three benchmarks on their indicative workloads. The null check columns give dynamic counts obtained by instrumenting both the original program and the APPEND-modified program at the bytecode level to record null checks executed during runtime.

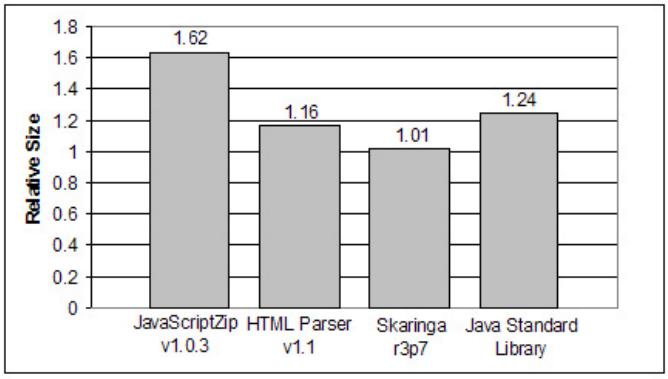

Figure 6.4: Bytecode size changes for transformed programs and libraries. Each column is separately normalized so that the unmodified bytecode size is 1.0. Larger values indicate code size increases. The "Java Standard Library" column indicates the `java`, `util`, `lang` and `io` components of the Harmony Java 1.5 standard library.

performed, and also that the run-time cost of this checking is minimal.

On the other hand, class files subject to our transformation grew moderately. Figure 6.4 summarizes the changes in bytecode size with each entry separately normalized to 1.0. The three programs and the standard library comprised 582 class files totaling 1663k before the transformation and 2036k worth of class files after, for a total increase of 22%.

Chapter 7

Related Work

Our approach falls somewhere between error detection and fault recovery. In this chapter we contrast it to similar efforts to improve software quality.

Static analyses to find program defects have been the focus of much recent research [6, 10, 13, 16, 22, 30]. Tools such as FINDBUGS are able to detect possible null pointer dereferences, as well as other defects, typically at the cost of false positives and false negatives. FINDBUGS matches user code against a variety of *patterns* for potential bugs [24]. Creating a syntactic pattern for generic null pointer dereference remains a challenge, and is not the focus of such work. Even with specific tuning of the static analysis in FINDBUGS' null pointer checking component [25], false positive rates for null pointer analyses are often high for the reasons discussed in Chapter 1. Our transformation approach avoids false alarms at the cost of program overhead; instead of attempting to decide which pointer dereferences are likely to be errors, we treat all dereferences as potential NPE sites, and always insert recovery code unless we know the dereference will definitely not be null. The programmer is relieved of manually inspecting the code or defect warnings. Modulo the assumptions in Chapter 4, our technique does not suffer from false negatives because each potential null pointer dereference is guarded by a check. Our transformation is orthogonal to tools such as FINDBUGS; instead of attempting to flag the logic errors in the code, we implement a safeguard for all potential errors of one specific type. Our approach can instrument the program with logging code to reveal where bugs have occurred after-the-fact, without allowing the NPE to ever manifest.

Checkpointing and transactions are common approaches to dealing with run-time errors. Borg

et al. [8] describe a checkpointing system that allows unmodified programs to survive hardware failures. Essentially, every system call is intercepted and logged. Others [34,42,43] provide similar services. Lowell et al. [34] describe a checkpointing service for recovery from failure for general applications with low overhead, on the order of less than 0.6% application slowdown. In Chapter 6, we demonstrate that our checking has a comparably low overhead, although our approach deals only with null pointer exceptions, not with all system faults. Shapiro et al. [43] also describe a checkpointing based system for managing persistence, called EROS. Consistency is preserved across memory writes through explicit checking during a snapshot operation. APPEND, like EROS, can use a runtime mechanism transparent to the user to check for null pointer dereferences, and can be combined with a recovery policy, as described in Section 5.2, to maintain database consistency as well. APPEND can prevent the persistence of objects containing unexpected null pointers, or default objects created by the tool, as the policy specifies.

In addition, such checkpointing and transaction oriented techniques address an orthogonal error handling issue. In Borg et al.'s system, a buggy process that reads a null value from a database on initialization will continue to fail no matter how often it is recovered unless something else changes. Lowell et al. [33] formalize this point by noting that the desire to log and replay all events actually conflicts with the ability to recover from all errors. Checkpointing systems are very good at preventing hardware failures and quite poor at preventing software failures; Lowell et al. suggest that 85–95% of application bugs cause crashes that would not be prevented by a failure-transparent systems. Our technique addresses an important subset of such application bugs.

Demsky and Rinard [14] repair defects in key data structures at run-time. Their technique works at the level of data structures and not at the level of program instructions, and it may be viewed as addressing the orthogonal problem of restoring complex program invariants. For example, their technique is well-suited to repairing the links in an inconsistent doubly-linked list, while ours might create a default list or default object for a method that was expecting one but received a null instead. In Section 5.2 we presented an example of a recovery policy in our system that maintains data structure consistency and prevents invalid objects from being written to a database.

Rinard also proposes to use a metalanguage to partition computation into tasks [39]. If a soft-

ware error or hardware fault is encountered, the task is discarded and execution continues. The system allows users to bound the distortion of the output when tasks area discarded, which may allow users to confidently accept results of computations that have encountered failures. Our work provides no formal bound but also requires no task-division annotations.

Rinard explores *acceptability-oriented* and *failure-oblivious* computing [38, 40]. In the former, systems are built to satisfy key properties rather than to be completely free of errors. Our approach can be viewed in that framework as a way of applying resilient computing at the low level of individual instructions with automatically-generated recovery actions and no explicit specifications.

Li and Yeung classify computations that have qualitative user-level interpreted results as *soft computations*, where data corruptions may change the result of such computations but user's interpretation of the results need not change [31]. Rather than being numerically oriented, such applications have a higher-level, subjective user interpretation. They report that 62% of dynamic instructions in their benchmarks are part of soft computations. APPEND is well-suited to tackle situations where availability, rather than data precision, is fundamental to usability.

Vo et al. describe XEPT, an instrumentation language that can be used to help detect, mask, recover, and propagate exceptions from library functions when source code is not available [50]. Such functionality is especially useful when using software under proprietary control when original developers are unable to fix bugs in the code. A list of functions to be intercepted, along with C-like code to handle exceptions, is provided by the user and then instrumented through the tool. APPEND can also be used in situations where the source code is not available directly, and in Chapter 6 we presented experimental results for a library-protection usage model that is similar to the XEPT approach. APPEND provides protection from faulty library object code, and can also extend this protection to any functions without the need to specify which functions to intercept.

Exception handling and error recovery have been studied by Fu et al. [20]. Because it is difficult to generate exceptional situations on demand, their approach focuses on white box testing error of handling code by injecting faults. They use an analysis that "allows compiler-generated instrumentation to guide the fault injection and to record the recovery code exercised" to create a test coverage metric. Their technique applies to *checked* exceptions, where it achieves high cov-

erage. By contrast, the null pointer exceptions addressed by our approach are usually *unchecked* exceptions. As mentioned earlier, null pointer exceptions have been reported as the most common error [11] in Java programs. APPEND is able to detect and recover from such exceptions without the need for the user to directly manipulate source code.

Inasmuch as our notion of recovery policies involves program transformations that operate on code at compile-time according to rules and contexts, it is tempting to phrase them in terms of aspect-oriented programming (e.g., [27]). Transformations of the form `foo(x);` $\Longrightarrow$ `if (x == null) { x = new Bar(); } foo(x);` could be reasonably phrased using *around* advice in popular AOP systems, although it might require separate advice for each class `Bar`. However, transformations such as `x = a.b.c;` $\Longrightarrow$ `if (a && a.b && a.b.c) { x = a.b.c; }` cannot always be conveniently phrased in commonly-available AOP systems. In addition, composing aspect mechanisms and understanding the semantics when multiple pieces of advice apply to the same bit of code is still an active area of research (e.g., [28]). Our system is much more specialized than AOP, but we claim it is more convenient for composing context-sensitive transformations that apply after null checks fail.

Cristian constructs a unified view of programmed and default exception handling based on automatic backward recovery [12]. Programmed exception handling is an example of a fault avoidance technique, while default exception handling falls under the category of fault tolerance. Both programmed and default exception handling can be designed to solve the same problems of masking, recovery, and signaling [12]. Recovery blocks [4] are one example of default exception handling [12], and are a way of organizing programs to include tests for potential errors and recovery actions if those errors are detected. The error detection takes the form of an acceptability check that is explicitly inserted into the code. As long as the acceptability check fails, correct state is restored and alternative code is tried. Recovery blocks are quite expressive, and many error-handling techniques can be phrased in terms of them. The code transformation portion of our approach could be simulated using recovery blocks by inlining the entire policy in to the program at each potential null-pointer dereference. Instead, we evaluate the policy at compile-time with respect to the context of the error and use the result to transform the code. This allows users to gain the advantages of

composable and reusable policies without paying time and space overhead for inapplicable recovery policies at run-time. Follow-up work (e.g., [46]) applies recovery blocks to algorithm-based fault tolerance, providing additional examples of efficient ways of detecting and responding to errors with the recovery block scheme.

Chapter 8

Conclusions

We aim to create a world where exceptions are not raised: instead, operations become total functions where both valid and invalid inputs are mapped to specific and tailored actions. As a first step, we presented APPEND, a technique for handling null pointer exceptions in Java programs. Checking for null pointers by hand is tedious and error-prone. We analyze programs to locate possible null pointer dereferences and then insert null checks and error-handling code. The handling code is determined by composable recovery policies that are queried at compile-time and transform the program to add context-sensitive error handling. Such prevention and handling of null pointer exceptions is a first step towards changing Java's semantics with respect to exceptional behavior.

In our experiments we were able to take externally reported null pointer exceptions and transform programs, showing that our technique can do useful work at preventing and recovering from null pointer exceptions. We also measured the overhead our transformation induces when applied to programs and to standard libraries. Our approach supports incremental adoption, allowing files and components to be transformed as desired, both at the bytecode level (e.g., for each of development and code readability) and at the source code level (e.g., for debugging). Although many more null checks were executed at run-time in programs subjected to our transformation, the average execution time slowdown was less than 1% and the average class file size increase was 22%. We believe that this technique can improve availability by allowing the program to continue to execute, especially in scenarios where finding and fixing an entire class of bugs manually is not practical.

Bibliography

[1] Sunk by windows NT. *WIRED*, July 1998.

[2] Apache Harmony – open source Java SE, 2007. `http://harmony.apache.org/`.

[3] Advisor. Beware: 10 common web application security risks. Technical Report Doc 11756, Security Advisor Portal, January 2003.

[4] T. Anderson and R. Kerr. Recovery blocks in action: A system supporting high reliability. In *International Conference on Software Engineering*, pages 447–457, 1976.

[5] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–402, 1995.

[6] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, May 2001.

[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 *Object-Oriented Programing, Systems, Languages, and Applications*, October 2006.

[8] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1), February 1989.

[9] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *Object-oriented programing, systems, languages, and applications*, pages 403–417, 2003.

[10] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Network and Distributed System Security Symposium (NDSS)*, pages pages 171–185, San Diego, CA, February 2004.

[11] Maciej Cielecki, Jędrzej Fulara, Krzysztof Jakubczyk, and Jancewicz. Propagation of JML non-null annotations in java programs. In *Principles and practice of programming in Java*, pages 135–140, 2006.

[12] Flaviu Cristian. Exception handling and software fault tolerance. *IEEE Trans. Computers*, 31(6):531–540, 1982.

[13] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Notices*, 37(5):57–68, 2002.

[14] B. Demsky and M. Rinard. Automatic data structure repair for self-healing systems, 2003.

[15] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. Technical Report MIT-LCS-TR-875, MIT, December 2002.

[16] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA*, October 2000.

[17] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *International Conference on Software Engineering*, pages 449–458, 2000.

[18] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen Mccamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[19] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *International Symposium on Software Testing and Analysis*, pages 133–144, 2006.

[20] Chen Fu, Ana Milanova, Barbara G. Ryder, and David Wonnacott. Robustness testing of java server applications. *IEEE Trans. Software Eng.*, 31(4):292–311, 2005.

[21] Fausto Giunchiglia and Paolo Traverso. Program tactics and logic tactics. *Ann. Math. Artif. Intell.*, 17(3-4):235–259, 1996.

[22] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.

[23] Jason Hickey and Aleksey Nogin. Extensible hierarchical tactic construction in a logical framework. In *Theorem Proving in Higher Order Logics*, pages 136–151, 2004.

[24] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM Press.

[25] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. *SIGSOFT Softw. Eng. Notes*, 31(1):13–19, 2006.

[26] W. Keller. Mapping objects to tables – a pattern language, 1997.

[27] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *International Conference on Software Engineering*, pages 49–58, 2005.

[28] Sergei Kojarski and David H. Lorenz. Awesome: an aspect co-weaving system for composing multiple aspect-oriented extensions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 515–534, 2007.

[29] Kesav Kumar. Serialization problem in DocumentFactory, October 2001. `http://sourceforge.net/mailarchive/forum.php?forum_name=dom4j-user&viewmonth=200110`.

[30] K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In Kai Koskimies, editor, *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305, April 1998.

[31] Xuanhua Li and Donald Yeung. Exploiting soft computing for increased fault tolerance. In *Workshop on Architectural Support for Gigascale Integration*, 2006.

[32] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, June 9–11 2003.

[33] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In *Operating System Design and Implementation*, October 2000.

[34] David E. Lowell and Peter M. Chen. Discount checking: transparent, low-overhead recovery for general applications. Technical Report CSE-TR-410-99, University of Michigan, 1998.

[35] Donna Malayeri and Jonathan Aldrich. Practical exception specifications. In Christophe Dony, Jørgen Lindskov Knudsen, Alexander B. Romanovsky, and Anand Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 200–220. Springer, 2006.

[36] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[37] Marco Pöhler. JTidy Dom4j NullPointerException, May 2003. `http://www.mail-archive.com/dom4j-user@lists.sourceforge.net/msg01435.html`.

[38] Martin Rinard. Acceptability-oriented computing. In *Object-oriented programming, systems, languages, and applications*, pages 221–239, 2003.

[39] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *International Conference on Supercomputing*, pages 324–334, 2006.

[40] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Opearting Systems Design & Implementation*, pages 21–21, 2004.

[41] Axel Schairer, Serge Autexier, and Dieter Hutter. A pragmatic approach to reuse in tactical theorem proving. *Electr. Notes Theor. Comput. Sci.*, 58(2), 2001.

[42] Frank Schmuck and Jim Wyllie. Experience with transactions in QuickSilver. In *Symposium on Operating Systems Principles*, pages 239–53, 1991.

[43] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[44] S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis. Building a reactive immune system for software services, 2005.

[45] Saurabh Sinha and Mary Jean Harrold. Criteria for testing exception-handling constructs in java programs. In *Internal Conference on Software Maintenance*, pages 265–274, 1999.

[46] Andrew M. Tyrrell. Recovery blocks and algorithm-based fault tolerance. In *EUROMICRO*, pages 292–299, 1996.

[47] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *CASCON 1999*, pages 125–135, 1999.

[48] Arthur van Hoff. The case for java as a programming language. *IEEE Internet Computing*, 1(1):51–56, 1997.

[49] V Vipindeep and Pankaj Jalote. List of common bugs and programming practices to avoid them. Technical report, Indian Institute of Technology, Kanpur, March 2005.

[50] P. Vo and Y. Huang. Xept: a software instrumentation method for exception handling. In *Symposium on Software Reliability Engineering*, pages 60–69, November 1997.

[51] Westley Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[52] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, New York, NY, USA, 2004. ACM Press.