Changing Java's Semantics for Handling Null Pointer Exceptions

Sajjade Daraqshan Arshiya Zill-E-Ilahi Shaheen   Gulfishan Tamkanat   Dr.Abhijeet Kawade

Department of computer science, JJT University, Rajasthan

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. Errorhandling 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%.

1.Introduction

This paper introduces 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. Checking for null pointers manually is tedious and error-prone, especially when pointer values are created by external components or are part of a chain of object references. We analyze programs to locate possible null pointer dereferences and then insert null checks and error handling code. The error-handling code is specified at compile-time via compassable, context-sensitive recovery policies. 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 especially desirable in web services or dynamic web content, where users interpret the final results with respect to an acceptability envelope [23] and high availability is of paramount importance. Because program behavior is preserved when no null pointers are dereferenced, our approach can be applied to any Java program. Instead of raising null pointer exceptions, we change Java's semantics for pointer dereferences to a total mapping for all possible pointer values. Rather than having non-exceptional behavior defined only for valid pointer dereferences, we generate recovery code for the null values as well. We aim to transform programs so that null pointer exceptions are avoided and programs can continue executing without incurring a high run-time cost in space or speed. Null pointer exceptions, while conceptually simple, remain prevalent in practice. Null pointer dereferences are frequent [31], and have been reported as "a very serious threat to the safety of programs" and are the most common error in Java programs [7]. Many classes of null pointer exceptions can be found automatically by

156

static analyses [15]. 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 [25]. 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., [2]). 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. In addition, not all defect reports from static analysis tools are addressed [33]. Programs ship with known bugs [18], and resources may not be available to fix null pointer errors. We propose a program transformation that automatically inserts null checks and error handling code. No program annotations are required, and developers need not wade through defect reports. Programs are modified according to compassable 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 return true. 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., [24]); we present and measure a concrete, low-level, annotation-free version of such a system, and extend it to allow for user-specified actions. We choose to work at the application rather than modifying the existing null checking behavior of a Java Virtual Machine. This has the advantages of retaining portability between different virtual machines and of conceptual simplicity, and the disadvantages of requiring that all relevant source code be processed in advance. Our 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 paper are a presentation of our technique (Section 3, including our definition of soundness in Section 3.3), our notion of recovery policies (Section 4), and experimental evidence (Section 5) 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 5.3). We begin with a motivating example.

2 Motivating Examples

In this section we walk through the application of our technique to a simple 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 external systems (e.g., databases) can significantly reduce the effectiveness of testing in finding potential null pointer dereferences [21, 28].

```
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 defensive 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 a 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 and the research devoted to preventing them is a testament to the inconsistency of null pointer prevention in practice [15]. At the same time, manually placing checks in the code is not only time-consuming and error-prone, but can also make the code more complex and difficult to read.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.1 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 Node table = doc.selectNode("/html/body");
```

A more precise inter procedural 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., [11]), but we chose a flow-sensitive intra procedural analysis for performance and for predictability. Java programmers are already used to simple and predictable analyses, such as Java's definite assignment rules, and understanding the transformation simplifies reasoning about and debugging the transformed code.

3.2 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. 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. In Section 4 we categorize and describe possible recovery policies in more generality.

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 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.

3.3 Soundness

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. We explicitly assume that programs do not rely on NPEs for uses beyond signaling errors (e.g., using try and catch with NPEs as nonlocal gotos). In practice, this assumption is reasonable: for example, in the 3.5 million lines of code of Eclipse version 3.3.1, there were only 23 source code locations that caught NPEs or their super types. We further assume that the user-specified error-handling code will result in acceptable behavior. 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 [1, 23, 25], but, admittedly, may result in unexpected or unintended consequences. Although we cannot offer a solution for all such situations, we believe that careful policy construction,

combined with logging functionality, will minimize the risk of unwanted situations and allow for directed debugging efforts in the rare instances when any APPEND inserted recovery code is called.

4 Error-Handling and Recovery Policies

Section 3 discussed how APPEND locates potential null pointer dereferences. In this section, we describe a framework for user-specified, compassable recovery policies that are applied at compile-time to instrument the code with context-specific recovery actions. 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. 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. A recovery policy can be global or it can be associated with a particular class, both as a subject and as a context. 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 [22]. In interactive theorem proving, proof obligations in the object language can be manipulated and simplified by tactics (see e.g., [13, 14]), programs written in a meta language. 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." Our recovery policy notion is also similar to aspect-oriented programming [16], in that rule- and context- based program transformations are applied at compile-time, although aspects typically do not call or direct other aspects [17]. One example of a recovery policy is the default constructor insertion described in Section 3.2. That policy is applicable() when the type under consideration has a default constructor with no arguments. A logging policy is another example: 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.

4.1 Policy Granularity

We require the user to provide a global recovery policy or use one of the default ones we provide. Individual classes and contexts can be annotated with specific recovery policies if desired. Example pseudocode for the apply function of a global recover policy is given in Figure 1. At compile-time, during the program analysis and transformation, we invoke global.apply() on each potential nullpointer 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. The example global policy in Figure 1 gives priority to policies associated with the potentially-null object and with the surrounding class. As an example of the former, a particular application might require that all NPEs associated Input: The program context C and an error location L.

```
1: if the dereferenced object at L has a policy P1
^ P1.applicable(C,L) then
2: return P1.apply(C,L)
3: else if the context class at L in C has a policy P2
^ P2.applicable(C,L) then
4: return P2.apply(C,L)
5: else if the context method at L in C has a policy P3
^ P3.applicable(C,L) then
6: return P3.apply(C,L)
7: else
8: if logging.applicable(C,L) then
9: C,L   logging.apply(C,L)
10: end if
11: if constructor.applicable(C,L) then
12: C,L   constructor.apply(C,L)
13: end if
14: return (C,L)
15: end if
```

Figure 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. 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. For example, in a User Level Transaction 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; 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 [27]. In general, attempts that stop the execution of a block or function when an NPE is prevented are variations of fail-stop computing. 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.

4.2 Data Structure Consistency

While 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 unchanged.

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

Figure 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 Section 4. safe 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. Many proposals exist for using user-defined or computer generated constraints (e.g., [10]) on data structures in the program or database to enforce consistency. A simple recovery tactic to prevent cascading errors in such a case would be to prevent APPEND from persisting any recovery 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 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 related to object consistency. Context at the class level, as opposed to task blocks as described by Rinard [23], 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.

5 Experimental Results

Although source code complexity need not increase with our transformation, byte code size, running time and utility must be considered. To address these issues, we have conducted several experiments to evaluate 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 3.2 is applicable (i.e., if the dereferenced object has a default constructor), we apply it. Otherwise, if the skip policy from Section 4 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.

5.1 Examples from Application Programs

In this section we show how APPEND can be applied to real-world examples of NPEs. We searched various bug repositories and forums for examples of code that raised NPEs, and after verifying that the NPE could be reliably reproduced, we applied our transformation. We then executed the resulting code, making sure that the NPE was no longer raised. Returning to the JTIDY example described in Section 2, 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 select Node 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 report2 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; new Doc 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.

5.2 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
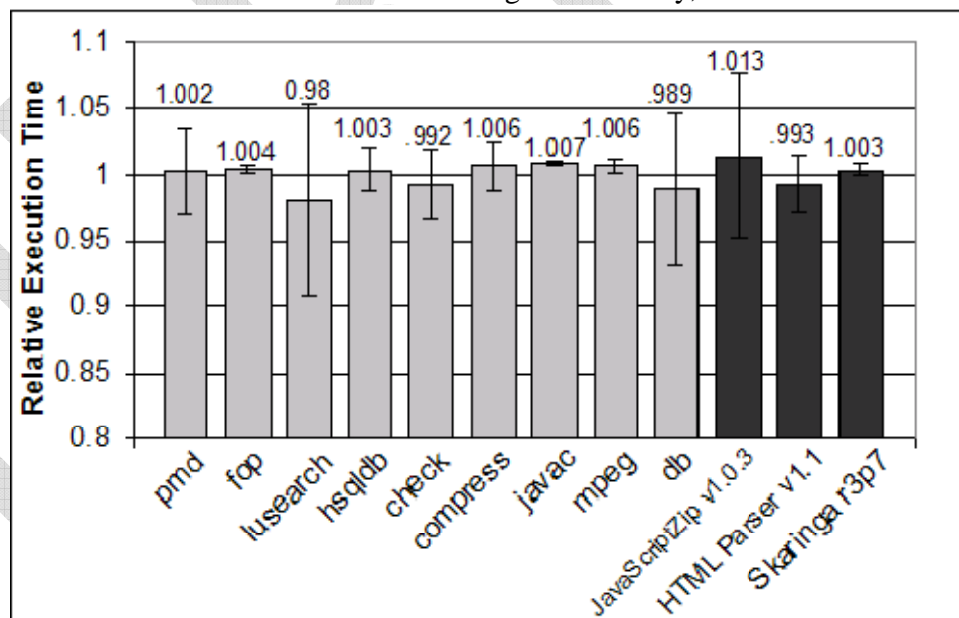
primary codebase. We demonstrate this approach on a defect in the Java Standard Library, version 1.1.6 (Sun Developer Network bug ID 4191214). The defect itself lies in the library's URL class. The bug report included sample code to elicit the NPE by accessing a Vector v1 of five URLs:

```
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 host Equal method of the URL class in the library, which was called form the equals method of URL, which was itself called by the index Of method of the Vector library class. After transforming the library with our technique, the host Equal function no longer raises an uncaught exception, and the overall output is a correct printout of the indices of the URLs 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 5.1 and Section 5.2 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 available. 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.

5.3 Performance and Overhead

Because APPEND inserts code into class files for null checking and recovery, to be usable it must have

only a

Figure 3. 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
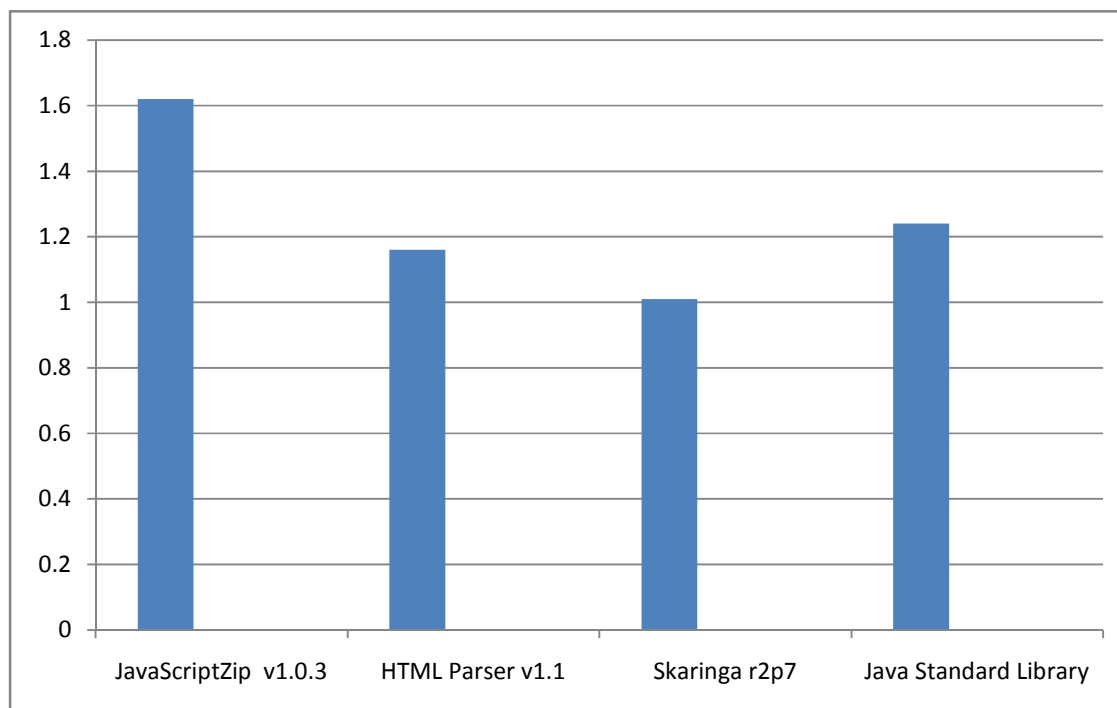
standard library. The three dark columns on the right are transformed applications run against a transformed library. The error bars represent standard deviations from twenty trials. minor impact on on code size and execution time. Using two separate benchmark suites we compared the running time and 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 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. We used benchmark programs from the the DaCapo [4] 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 3 summarizes the results, reporting the average of twenty trials (the nine 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 been converted by APPEND. Figure 3 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 is a substantial increase over the base amount of checking performed by the unmodified programs. Figure 4 summarizes the number of null checks that were inserted for three benchmarks at runtime. For the two larger benchmarks, the number of executed null-checks increased by an average factor of three without a significant runtime slowdown. JAVASCRIPTZIP, the benchmark that showed the greatest runtime slowdown, performed over a thousand times more null-checks when instrumented with APPEND. To be sure that the inserted null checks were actually being called during program execution, we counted the number of times our null checks are called, versus the number of times user provided null checks are called, for our three benchmarks. Figure 4 also shows the number of times a null check was called by the program for both APPEND and user-inserted guards. From these three experiments we can conclude both that our transformation is actually affecting the program, in that many additional null-checks are performed, and also that the run-time cost of this checking is low. On the other hand, class files subject to our transformation grew moderately. Figure 5 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%.

6 Related Work

Our approach falls somewhere between error prevention and fault isolation. In this section we contrast it to similar efforts to improve software quality. Static analyses to find program defects have been the focus of much recent research [3, 6, 8, 9]. Many static analysis tools are able to detect possible null pointer dereferences, as well as and other defects, typically at the cost of false positives and false negatives. False positive rates for null pointer analyses are often high for the reasons discussed in Section 1, and our transformation approach entirely

avoids false positives at the cost of program overhead. False negatives do not arise (with some assumptions and restrictions stated in Section 3) since each potential null pointer dereference is guarded by a check.

Figure 5.

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. Checkpointing and transactions are common approaches to dealing with run-time errors. Borg et al. [5] describe a checkpointing system that allows unmodified programs to survive hardware failures. Essentially, every system call is intercepted and logged. Others (e.g., [20, 26]) provide similar services. Our approach deals only with null pointer exceptions, not with all system faults. In addition, such 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. [19] formalize this point by noting that the desire to log all events actually conflicts with the ability to recover from all errors. Such 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. Rinard also proposes to use a meta language to partition computation into tasks [24]. If a software 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. 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 [32]. APPEND can also be used in situations where

Bench Mark Program	Static Null Check			Dynamic Null Check		
	Normal Increase	With	Append	Normal Increase		With Append
JavaScript Zip	**9**	**9932**	**1100x**	**0**	**19848**	∞
HTML Parser	**170499**	**623361**	**3.66x**	**190384**	**1146002**	**6.02x**
Skaringa	**371**	**1732**	**4.66x**	**296**	**1360**	**4.60x**

Figure 4.

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. The "Static" column counts the number of checks in the bytecode; the "Dynamic" column measures checks actually performed at run-time. the source code is not available directly, and in Section 5 we presented experimental results for a library-protection usage model that is similar to the XEPT approach. Exception handling and error recovery have been studied by Fu et al. [12]. Because it is difficult to generate exceptional situations, their approach focuses on white box testing error of handling code by injecting faults. Their technique applies to checked exceptions, where it achieves high coverage. By contrast, the null pointer exceptions addressed by our approach are usually unchecked exceptions. Inasmuch as our notion of recovery policies involves program transformations that operate on code at compiletime according to rules and contexts, it is tempting to phrase them in terms of aspect-oriented programming (e.g., [16]). Transformations of the form

```
foo(x);
=)
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;
=)
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., [17]). 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. Recovery blocks [1] 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, correction code is executed and the original code is tried again. 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. More recent work (e.g., [29]) 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. Rinard explores acceptability-oriented and failure oblivious computing [23, 25]. In the former, systems are built to satisfy key properties rather than to be completely free of errors. Our work can be viewed in that framework as an application of resilient computing at the low level of individual instructions with automatically-generated recovery actions and no developer-provided specifications.

7 Conclusions

We presented APPEND, a technique for handling null pointer exceptions in Java programs. Checking for null pointers by hand can be 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 exceptional behavior semantics. We desire 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. In our experiments we were able to take externally reported null pointer exceptions and transform programs, showing that our technique can do useful work. We also measured the overhead it 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, 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 programs to continue to execute, especially in scenarios where finding and fixing an entire class of bugs manually is not practical.

Acknowledgments

References

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

2   M. Atkinson and R. Morrison. Orthogonally persistent object systems. The VLDB Journal, 4(3):319–402, 1995.

3   T. Ball and S. 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.

4   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. Stefanovi´c, T. VanDrunen, D. von Dincklage, and B.Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In Object-Oriented Programing, Systems, Languages, and Applications, Oct. 2006.

5   A. Borg,W. Blau,W. Graetsch, F. Herrmann, andW. Oberle. Fault tolerance under UNIX. ACM Transactions on Computer Systems, 7(1), Feb. 1989.

6   H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2004.

7   M. Cielecki, J. Fulara, K. Jakubczyk, and L. Jancewicz. Propagation of JML non-null annotations in java programs. In Principles and practice of programming in Java, pages 135–140, 2006.

8   M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. SIGPLAN Notices, 37(5):57–68, 2002.

9   D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-
written compiler extensions. In Operating Systems Design and Implementation, 2000.

10 M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In International Conference on Software Engineering, pages 449–458, 2000.

11 S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In International Symposium on Software Testing and Analysis, pages 133–144, 2006.

12 C. Fu, A. Milanova, B. G. Ryder, and D. Wonnacott. Robustness testing of java server applications. IEEE Trans. Software Eng., 31(4):292–311, 2005.

13 F. Giunchiglia and P. Traverso. Program tactics and logic tactics. Ann. Math. Artif. Intell., 17(3-4):235–259, 1996.

14 J. Hickey and A. Nogin. Extensible hierarchical tactic construction in a logical framework. In Theorem Proving in Higher Order Logics, pages 136–151, 2004.

15 D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. SIGSOFT Softw. Eng. Notes, 31(1):13–19, 2006.

16 G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In International Conference on Software Engineering, pages 49–58, 2005.

17 S. Kojarski and D. H. Lorenz. Awesome: an aspect coweaving system for composing multiple aspect-oriented extensions. In Object-Oriented Programming, Systems, Languages, and Applications, pages 515–534, 2007.

18 B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In

Programming Language Design and Implementation, June 9–11 2003.

19 D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery.   In Operating System Design and Implementation, Oct. 2000.

20 D. E. Lowell and P. M. Chen. Discount checking: transparent, low-overhead recovery for general applications. Technical Report CSE-TR-410-99, University of Michigan, 1998.

21 D. Malayeri and J. Aldrich. Practical exception specifications. In Advanced Topics in Exception Handling Techniques, pages 200–220, 2006.

22 G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst., 1(2):245–257, 1979.

23 M. Rinard. Acceptability-oriented computing. In Objectoriented programming, systems, languages, and applications,pages 221–239, 2003.

24 M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In International Conference on Supercomputing, pages 324–334, 2006.

25 M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In Opearting Systems Design & Implementation, pages 21–21, 2004.

26 J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In Symposium on Operating Systems Principles,pages 170–185, 1999.

27 S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis.Building a reactive immune system for software services. In USENIX Annual Technical Conference, 2005.

28 S. Sinha and M. J. Harrold. Criteria for testing exceptionhandling constructs in java programs. In Internal Conference on Software Maintenance, pages 265–, 1999.

29 A. M. Tyrrell. Recovery blocks and algorithm-based fault tolerance. In EUROMICRO, pages 292–, 1996.

30 R. Vall´ee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In CASCON 1999, pages 125–135, 1999.

31 A. van Hoff. The case for java as a programming language.IEEE Internet Computing, 1(1):51–56, 1997.

32 P. Vo and Y. Huang. Xept: a software instrumentation method for exception handling. In Symposium on Software

   Reliability Engineering, pages 60–69, Nov. 1997.

33 W. Weimer. Patches as better bug reports. In Generative Programming and Component Engineering, pages 181–190,2006.