CHAPTER VI

GENERALIZING THE J-ORCHESTRA INDIRECTION MACHINERY

This chapter discusses how one of the technical contributions of this dissertation can be generalized to domains other than distributed computing. Specifically, we take a closer and broader look at the J-Orchestra approach to enabling indirection in the presence of unmodifiable code (e.g., Java system classes), which is one of its sources of scalability. Chapter IV discussed the J-Orchestra analysis heuristic that determines which application objects get passed to which parts of native (i.e., platform-specific) code and a technique for injecting code that will convert objects to the right representation so that they can be accessed correctly inside both application and native code. Here we discuss the broader ramifications and limitations of user-level indirection and show how the approach taken by J-Orchestra can be fine-tuned so that user-level indirection can be applied to more system classes.

6.1 Introduction

In this chapter, we take a more general look at user-level indirection techniques and show that all different versions of the idea converge into using the same general approaches. Then we discuss why the presence of native code always results in correctness limitations. Some of these limitations are straightforward (e.g., native code can have its own state) while some others are more subtle (e.g., native code can change user-level state

directly). Despite the fact that we generally use Java (i.e., Java language syntax, Java terminology, and JNI conventions) as our reference system, our observations apply to most other runtime systems for platform-independent binary code applications such as the CLR and .NET technologies.

6.2 User-Level Indirection Techniques

We use the name "user-level indirection" to describe any general technique that transparently interposes extra functionality to the execution of existing applications by using code transformation techniques, instead of modifying the underlying implementation of the runtime system. Applications of user-level indirection include transparent distributed execution [21][66][74][75][84], persistence [12][46][59], profiling [33], and logging [48]. In general, user-level indirection aims at capturing specific events and performing actions whenever they occur. Such events typically are:

- Access to a field of an object or a static field (reading or modifying the field).
- Calls to a method of an object of a specific type, or calls to a static method.
- Object construction.

For instance, we may want to add indirection to all changes to the fields of an object for logging: we may want a permanent log of all state updates in a running system. This is possible by finding all field access instructions in the application and modifying them to log their action before taking it. The logging code is either included inline at the field access site, or a separate method can be called.

What complicates user-level indirection is the existence of reusable core functionality in the form of *system classes* (a.k.a. *standard library classes*). User-level indirection

cannot afford to ignore system classes, *even if the intended use is not concerned with system-level events*. For instance, consider a user-level indirection system that performs actions every time a user-level method gets called. User-level methods, however, often get called by system-level code. For instance, system libraries often accept a callback object and invoke its methods in response to asynchronous events, or in response to system code actions initiated by a user-level call. Thus, the user-level indirection technique needs to ensure that it allows and correctly handles all calls, regardless of whether they occur inside user-level or system-level code.

In popular modern runtime systems, the majority of system class code is not special. Most of the Java system classes, for instance, are distributed in Java bytecode format. Thus, one can apply the same user-level indirection techniques to both user-level code and byte-code-only system classes. Indeed, several systems [22][84] follow this approach. The standard technique in this case is to create a separate, instrumented version of the system classes. The instrumented version co-exists with the standard system classes in the same application. In this way, an application can access both the user-level indirected versions of system classes and the original versions without any conflict. This is necessary, since the system classes are often used inside the instrumentation code itself. In original application code, however, all uses of system classes are replaced with uses of their instrumented counterparts. Reference [22] calls this the "Twin Class Hierarchy" approach (TCH). As an example, imagine that the original Java application contains code such as:

```
class A {
  public java.lang.String meth(int i, B b) {...}
}
```

The rewritten class would use the instrumented class types:

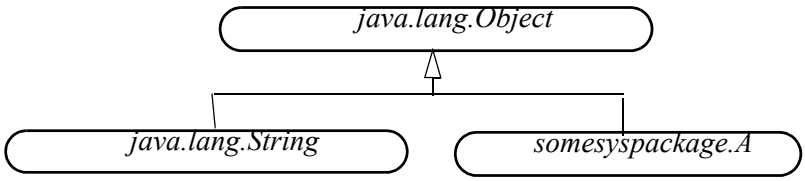

Figure 6-1: (a): Original system classes hierarchy

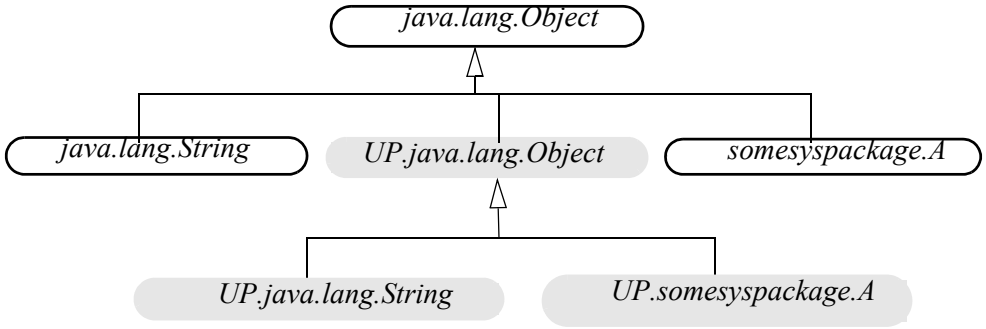

Figure 6-1: (b): Replicating system classes in a user package ("UP")

```
class UP.A {
  public UP.java.lang.String meth(int i, UP.B b)
  {...}
}
```

(`UP` in the above code stands for "user package".) Figure 6-1 shows the effects on the class hierarchies pictorially.

6.3 Transparency Limitations

The problems with any user-level indirection technique begin when a system class with native code needs to be instrumented. Native code (a.k.a. *platform-specific binary code*) is often used to implement system-level functionality. Some of the most fundamental system classes (e.g., the ones dealing with threading, file and network access, GUI, and so forth) rely on native code, mainly for reasons of low-level resource access, such as context-

switching or fast graphical operations. System classes with native code are, thus, a way to export runtime system functionality as language-level facilities.

Native code cannot be instrumented without invalidating all the advantages of the user-level indirection approach. Changing native code requires platform-specific changes and the creation of special versions of the runtime system (either the executable program or its dynamic libraries). Similarly, analyzing native code and relying on its implementation properties is a platform-specific task. Thus, dealing with native code is incompatible with the main motivation for user-level indirection: that of portability and platform independence. Therefore, native code is *opaque* for the purpose of user-level indirection: it can be neither modified nor analyzed.

Having an application access opaque code immediately introduces limitations in user-level indirection approaches. Even if opaque code is a small percentage of the total system code,[1] it is likely to be used by every application and needs to be handled correctly. (In fact, because `java.lang.Object` and `System.Object`, the root classes in Java and C#, respectively, use native code in their implementation, one could argue that every program written in these languages contains opaque code.) Clearly, one limitation is that user-level indirection cannot be used to intercept actions occurring entirely inside native code. For instance, we cannot observe and log updates to program state kept inside native code: such state is invisible to the user-level. That is, changes to internal system state (e.g., the contents of a low-level window, the scheduling structure of threads, and so forth) cannot be intercepted using user-level indirection. Although it may seem that such state is low-level

1. Only about 3% of the Java system classes have native methods. (All numbers were measured on Sun JDK 1.4.2.) Nevertheless, as we show later, these are some of the most commonly used classes in Java and are likely to constitute a much larger percentage of the loaded system code in a Java application.

and is outside the scope of user-level indirection, the restriction nevertheless places boundaries on what is achievable with user-level indirection alone. For instance, without reliance on implementation specifics of the Java system libraries, a distributed execution system that relies on user-level indirection (similarly to J-Orchestra: Pangaea [74], Addistant [84], and JavaSplit [21]) cannot hope to transparently migrate window or thread objects from one machine to another. This task can still be achieved by special-purpose emulation of the semantics of a thread or window at the user level, but not by employing general-purpose user-level indirection techniques on the Java system classes.

Often, however, the interactions of native code with user-level indirection are more subtle. In the Java system, native code can directly read or modify the state of object fields declared in bytecode. This allows for tight integration of native code and Java code. Essentially, the Java Native Interface (JNI) is a way to program using the full object model of the JVM with C or C++ as the host language. Direct access to fields inside native code complicates matters for user-level indirection. Consider the TCH user-level indirection approach for instrumenting standard Java libraries [22]. (This approach is representative of other user-level indirection techniques, including the one in J-Orchestra.) In this approach, if a class `A` has a native method, an instrumented version of `A` delegates calls to the native method of an internal `A` object. This technique is used because a native method implementation in Java is bound to a particular class name and cannot be reused for a different class. For instance, consider original code as follows: (This code does not reflect the Java `File` class but the structure is representative of several system classes with native methods.)

```
class File {
  ...
  public native void write(byte b);
}
```

The instrumented version of this class would be:

```
class UP.File {
  private File origImpl_;
  ...
  // delegate to native method
  public void write(byte b) {origImpl_.write(b);}
}
```

It may at first seem that the `UP.File` class can use arbitrary user-level indirection for its non-native methods. Nevertheless, this is not the case. Imagine that the `File` class also has a non-native method `newLine`:

```
class File {
  ...
  public native void write(byte b);
  public void newLine() { ... }
}
```

It is not safe to indirect method `newLine` (e.g., to track its changes to fields of a `File` object) yet simply delegate method `write`. To see this, consider the re-written code:

```
class UP.File {
  private File origImpl_;
  ...
  // delegate to native method
  public void write(byte b) {origImpl_.write(b);}
  public void newLine() {...} // instrumented body
}
```

The problem is that any call to method `write` affects the `origImpl_` object, while any call to method `newLine` affects the current object of type `UP.File`. Separating these two objects (when they were one in the original application) destroys the transparency of

user-level indirection. Therefore, we see that the TCH user-level indirection approach is all-or-nothing: any class that has even a single native method is impossible to instrument transparently. This limitation is not specific to the TCH approach: following the same reasoning one can see that once a class has native methods, it is not possible to transparently replace it with an instrumented copy of the class such that it implements any kind of user-level indirection.

The ability of Java native system code to directly access user-level state hinders many more user-level indirection tasks. For instance, consider user-level indirection approaches that capture all updates to fields of an object (e.g., to implement transparent persistence or distributed execution). In this case, all objects that can ever be referenced by native code cannot be fully indirected using user-level indirection techniques. That is, even if an object's class has no native methods, if the object is ever referenced by some other class's native code, then we cannot indirect all access to the object's fields.

Furthermore, often constraints on the use of user-level indirection have to do with restrictions derived from the structure of the user-level indirection scheme itself. For instance, consider again the above TCH rewrite. Without any special provisions, the limitations on the use of indirection propagate to all subclasses. A subclass `ROFile` of the original `File` class may have no native methods, yet its methods cannot be instrumented. If the instrumentation were performed, the `UP.ROFile` class would be a subclass of `UP.File` and not of `File`. Thus, `UP.ROFile` would not be able to access non-public members of `File`. We later discuss how to remove this limitation.

6.3.1 Beyond Java Conventions: Native Code in .NET

For the purposes of our discussion, the .NET and Java technologies are almost equivalent, with .NET being slightly more restrictive due to the unstructured nature of interfacing between managed and unmanaged code. Just like in the Java case, managed and unmanaged code in the CLR can operate on the same objects. Just like in Java, .NET unmanaged code, usually written in C++, provides many system services that are impossible to implement in a managed environment because they require such low-level programming techniques as direct memory access. Unlike the Java platform, however, which clearly distinguishes between bytecode and native libraries and provides a clean interfacing mechanism between the two in the form of the JNI, the C# core classes implementation consists of managed and unmanaged code that are binary compatible with each other.

At the language level, the annotation `[MethodImplAttribute(MethodImplOptions.InternalCall)]` specifies external methods that are implemented natively in the runtime itself. These methods use standard Microsoft C language calling conventions (such as `__stdcall` and `__cdecl`). In addition, the internal member methods in C# take this as the first argument, which in C++ becomes just a regular pointer that can be used to access and modify the memory of the underlying C# class directly. For example, a brief look at the Microsoft Shared Source CLI Implementation reveals that the C++ native code of the runtime relies on a very concrete object memory layout. For example, comparing whether two C# references point to objects of the same type includes comparing the pointers to their method tables, located at a predefined memory offset from the base references. Therefore, unmanaged code in the CLR not only accesses fields of objects, but is allowed to make assumptions about how these fields are laid out in memory. Such tight coupling between

managed and unmanaged code enables an efficient implementation for the runtime but also makes introducing any indirection into the managed code almost impossible. Therefore, introducing indirection by simply moving code of a Core Library C# class with native dependencies to a different package is even more unrealistic and error-prone than it is in Java. In the remainder of this chapter, all our qualitative observations should apply equally well to the CLR, unless we explicitly note otherwise.

6.4 Weak Assumptions of J-Orchestra Classification

To determine which program actions can be safely indirected, we would need to analyze the implementation of native methods. Since source code for the VM and its dynamic libraries will typically not be available, one important question is whether one can use the type information at the native code interface as a "poor-man's native code annotations." We discuss how some well-founded assumptions on the behavior of native code enable J-Orchestra to employ a conservative type-based analysis of what objects can be accessed by native code. It turns out that type information is often remarkably sufficient for determining the safety of user-level indirection.

6.4.1 Type-Based Analysis + Weak Assumptions

Recall that the majority (~97%) of Java system classes have no native methods. Such classes encode useful reusable libraries and not system-level functionality. It is, thus, crucial to automatically recognize system classes that do not interact with native code and to support correct user-level indirection for them. In general, this task is impossible without making assumptions regarding native code behavior. For instance, all classes in Java are subclasses of the `java.lang.Object` class, which has native code. In theory, any native

method can be receiving an `Object`-typed argument, discovering its actual type using reflection and performing on the object some action (e.g., reading fields) that would be undetected by any user-level indirection mechanism. Next we discuss practical assumptions that let us classify different parts of system functionality for safe user-level indirection.

In Section 6.2 we distinguished several different kinds of events typically captured by user-level indirection: access to fields, method calls, constructor calls, and so forth. Clearly none of these events can be captured if they occur entirely within opaque code. For instance, it is impossible to capture updates to state (i.e., variables) that is defined inside native code. The interesting case, however, is that of events concerning user-level (i.e., non-opaque) entities and the question of whether these can occur inside opaque code. For instance, we may want to capture all updates to an object field that is declared in a Java system class implemented in bytecode. We need to ask if this field is ever accessed inside native code. In this section we assume the full gamut of user-level indirection events, including access and modification of fields. If a certain application is only interested in capturing method and constructor calls, the restrictions are typically far less severe. Nevertheless, most interesting applications of user-level indirection (esp. distributed execution and persistence) need to capture field accesses.

Here we abstract away the specifics of the J-Orchestra approach. It makes two main heuristic assumptions regarding system classes:

- Classes without native methods have no special semantics.
- Native methods do not use dynamic type discovery (reflection, downcasting, or any low-level type information recovery) on objects supplied through method arguments.

These assumptions generally hold true with few exceptions. The first assumption does not hold, for instance, for classes in the `java.lang.ref` package. The second assumption does not hold in the implementation of reflection classes themselves. In Section 6.5 we discuss a study of the Sun implementation of Java system classes and how it supports our assumptions.

The first assumption essentially states that the JVM is not allowed to handle different types of objects specially when the objects just use plain bytecode instructions. For instance, the JVM is not allowed to detect the construction of an object of a "special" type and keep a reference to this object that native code can later use for destructive state updates. This is a reasonable assumption, conforming to good software design practices. The second assumption states that native code is strongly typed: if a reference is declared to be of type `T`, it can never be used to access fields (method calls are fine) of a subclass of `T`. For instance, the assumption prohibits native methods from taking an `Object`-typed argument, checking if it is actually of a more specific type (e.g., `Thread` or `Window`), casting the object to that type and directly accessing fields or methods defined by the more specific type. This assumption also encodes a good design practice: code exploits the static type system as much as possible for correctness checking.

With the above two assumptions, we can perform a classification of Java system classes with respect to whether they can employ user-level indirection transparently or not, based on their usage of native code. We will use the term *NUI* (for *non-user-indirectible*) to describe classes that cannot employ user-level indirection transparently. We can generalize the J-Orchestra rules from Chapter IV to infer all classes that have user-level indirection limitations, as follows:

1) A system class with native methods is NUI.

2) A system class used as a parameter or return type for a method or static method in a NUI class is NUI.

3) If a system class is NUI, then all class types of its fields or static fields are NUI.

4) If a system class, other than `java.lang.Object`*, is NUI, then its subclasses and superclasses are NUI.*

(Just like in Chapter IV, the above rules represent the essence of the analysis but are not complete. For instance, they do not discuss arrays or exceptions—these are handled similarly to regular classes holding references to the array element type and method return types, respectively. Note that interface access does not impose restrictions since an interface cannot be used to directly access state. We prefer the abbreviated form of the rules for readability, especially since the analysis is based on heuristic assumptions, and therefore we do not make an argument of strict correctness.[2] The numbers we later report are for the full version of the rules, however.)

Rule 1 above is justified because no user-indirection technique can guarantee to capture all field updates of an instance of a class with a native method. The native method can always perform updates without any indirection.

2. In addition, one possible native dependency that might not be determined correctly through our type base heuristic is "intrinsics," a class loading mode in which a JVM ignores the bytecode file of a class defined in the platform specification, providing instead a native implementation for the class's functionality. The problem arises when the vendor of a JVM that uses "intrinsics" fails to mark intrinsic methods as native. In that case, the only way to find out if some methods are intrinsic is by trial-and-error, and our static type based heuristic is no longer sufficient. Empirically, this has not been an issue for Sun's Hot Spot JVM.

Rule 2 is justified with a similar argument: if an object can be passed to native code, native code can alias it and (either during the native method execution or during a later invocation) change its state. Furthermore, the rule can be applied transitively: if a class is NUI then we cannot replace all its uses with uses of an instrumented version in a user package `UP`. Then all objects used as arguments of any method (even non-native) may have their fields accessed directly.

Rule 3 is analogous to Rule 2 but for fields: native code can access any object transitively reachable from an object that leaks to native code.

Rule 4 is justified by the specifics of the J-Orchestra user-level indirection scheme. We saw an instance of this restriction in Section 6.3: if a class cannot be indirected, its uses in the application cannot instead employ a modified copy of the class in a user-level package. Thus, all subclasses and superclasses also cannot be copied to a user level package, as they may need to access non-public fields of their superclass.

These rules enable user-level indirection to be used safely for many Java system classes. Specifically, 37% of the Java system classes are classified as having no dependencies to native code and, thus, being able to employ user-level indirection safely.

Still, however, these rules are too conservative, as 63% of the system classes are deemed non-indirectible. Nevertheless, the rules are a good starting point and can be weakened to be made practical for specific applications of user-level indirection. For instance, in the context of J-Orchestra one more assumption is made relating to the way native code in different libraries can share state. The extra assumption allows placing different pieces of native code on separate machines and placing the instances of opaque classes in the same machine as the relevant code [51][87].

Next, we show one important general-purpose weakening of the rules. Rules 2 and 4 can be weakened significantly if we are allowed to modify system packages (still without touching native code) and we employ a more sophisticated user-level indirection scheme than that of J-Orchestra or TCH.

6.4.2 More Sophisticated Type-Based Analysis

The rules of the previous section are conservative because they assume that all code in system packages (be it native or not) is opaque. See, for instance, Rule 2: although any object that is used as a parameter of a native method can have its fields accessed with no indirection, there is no need to recursively propagate this constraint to the non-native methods of this object as well. If the object class is in pure bytecode, we can edit it and introduce indirection for accesses to its parameters. This, however, relies on a low-level assumption: we assume that the user-level indirection technique can modify system packages in order to edit the bytecode of existing system classes or add a new class in a system package. This is not desirable in some user-level indirection settings because it requires control over the startup environment of the JVM. Such control is not always possible, e.g., for deploying applets that random users will download and use inside a browser, or in systems in which the user cannot modify or extend the system package for security. Nevertheless, many applications of user-level indirection are allowed to set the parameters of the runtime system, and this can include a modified system package.

Under this assumption, we can use a weaker version of Rules 2 and 4.

1) A system class with native methods is NUI.

2') A system class used as a parameter or return type for a native method is NUI.

3) If a system class is NUI, then all class types of its fields or static fields are NUI.

4') If a system class is NUI, then its superclasses are NUI.

The weaker rules push the limits of user-level indirection much further: fewer than 8% of the Java system classes are classified as unable to employ user-level indirection (i.e., NUI). This means that a general-purpose user-level indirection technique can apply to more than 92% of the Java system classes with no special handling.

We already discussed how the new version of Rule 2 is a result of instrumenting the bytecode of bytecode-only NUI classes. The weakening of Rule 4 is more interesting. In the new Rule 4, a class does not impose any restrictions on its subclasses. This also eliminates any special handling of the `java.lang.Object` class, which is a common singularity in user-level indirection schemes.

To use the weaker version of Rule 4, we need to make sure that every system class `C` that cannot employ user-level indirection transparently is replicated in a user-level package. The replica class will just delegate all method calls to the original. Subclasses of `C` that have no native dependencies will employ full user-level indirection: an instrumented copy will be created in a user package and all references to the original class will become references to the instrumented version. As discussed in Section 6.3, the problem is that the instrumented class will not be able to access non-public members of `C`, as it is not in the same package as `C`. One solution is to make public all non-public members of class `C` by editing the class bytecode. (Or, equivalently, to create a subclass of `C` that exports the non-public members of `C`—see later.) A safer approach would be to emulate the Java access control at run-time using a technique such as that proposed by Bhowmik and Pugh [1] for

the Java inner classes rewrite. At load time, class `C` creates a secret key and passes it to the instrumented version of its subclass. When objects of the instrumented class need to access `C` members, they call a public method that also receives and checks the secret key. This is a safe emulation of the Java access protection, yet it avoids the requirement of placing classes in the same package.

An example application of this technique is shown in Figure 6-2(a). The example class `File` of Section 6.3 is now shown with a non-public field `field1`. `File` has a subclass `TXFile` with no native dependencies. Figure 6-2(b) shows the transformed classes so that `UP.File` and `UP.TXFile` can correctly replace all uses of `File` and `TXFile`, respectively, yet `UP.TXFile` can employ fully transparent user-level indirection. (As a low-level note, this transformation means that the instrumented system package, `UP`, needs to be loaded by the bootstrap class loader, since there is a call to method `UP.File.setKey` inside the `File` system class. The easiest way to effect this is to put the `UP` package in the `rt.jar` file.)

The effects of the transformation on the example class hierarchy are shown pictorially in Figure 6-3.

6.5 Validating The Assumptions and Analysis

We validate the assumptions and analysis of the previous section in three ways: first we measure the impact of our type classification for real applications: can we indeed use user-level indirection, without any special-case handling, for a large number of the system classes used by realistic applications? Next we examine an actual native code implementa-

```
class File {
 SomeT field1;
 ...
 public native void write(byte b);
 public void newLine() {...}
}

class TXFile extends File {
 ...
 public void writeString(String s) {
... foo(field1) ... }
}
```

Figure 6-2: (a): Original system class File (with a native method) and subclass TXFile (without native dependencies).

```
class File {
 SomeT field1;
 // Allow free access to field1 only to
 // class UP.File (and children)
 private static final Object key_ = new Object();
 static { UP.File.setKey (key_); }
 public SomeT get_field1(Object key) {
   if (key != key_)
     throw new IllegalAccessException();
   return field1;
 }
 ...
 public native void write(byte b);
 public void newLine() {...}
}

// Just delegates to File. Only used for correct
// subtype hierarchy.
class UP.File {
 protected File origImpl_;
 protected static Object  key_;
 public static void setKey(Object key)
 { key_ = key; }
 ...
 // delegate to native method
 public void write(byte b) { origImpl_.write(b); }
 public void newLine() { origImpl_.newLine(); }
}

class UP.TXFile extends UP.File {
 ...
 // methods of this class can employ any
 // user-level indirection scheme
 public void writeString(String s) {
   ...foo(origImpl_.get_field1(key_))...
 }
}
```

Figure 6-2: (b): Result of the user-level indirection transformation, with safe access to non-public fields of class File.

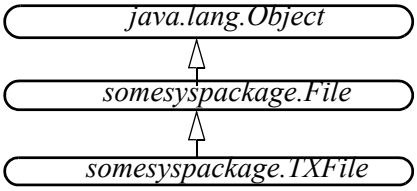

Figure 6-3: (a): A File class hierarchy

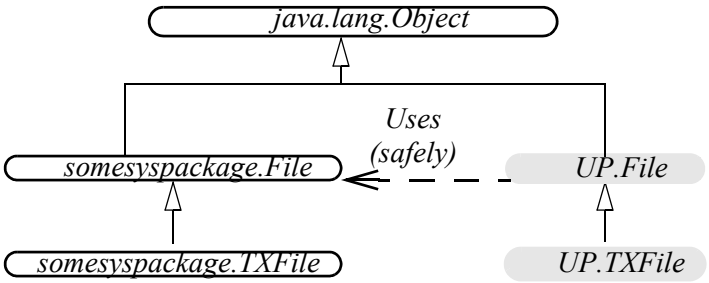

Figure 6-3: (b): Removing subclassing restrictions

tion of system methods and check whether it satisfies our assumptions. Finally, we perform a dynamic analysis of several Java applications and show that they do not violate the results of our type-based analysis during their execution.

6.5.1 Impact on Real Applications

An interesting question is to quantify the impact of the type-based analysis for real applications, as opposed to the set of all Java system classes. Although the more sophisticated version of our analysis allows to use indirection in 92% of the system classes, the remaining 8% are some of the most heavily used classes in practice. We demonstrate this in Table 6-1.. The table shows how many of the system classes actually used by different Java applications are classified as NUI under our analysis of Section 6.4.2. The table also shows how many of the used system classes have native methods themselves—this is a

lower bound on the number of NUI classes under any analysis. (We find the used classes by dynamically observing the loaded classes, minus JVM bootstrap classes. We then run our type-based analysis with the set of used classes as a universe set—any NUI dependencies introduced by classes that were not loaded are ignored.)

Three of the applications (javac, jess, mpegaudio) are standard benchmarks from SPEC JVM'98. (The rest of the SPEC JVM'98 programs yield practically identical numbers.) Unsurprisingly, these benchmarks are old and exercise few of the Java system classes. Nevertheless, we still see that more than 62% of the system classes used can employ user-level indirection. The next seven applications (antlr, bloat, chart, hsqldb, jython, ps, xalan) are from the more modern DaCapo benchmark suite (version beta050224). These applications are more realistic, yet they still do not exercise a large part of the Java system libraries. We see that our analysis enables 66-85% of the system classes used in the DaCapo benchmark programs to be safely indirected. For applications that exercise more of the Java system classes, we examined the Sun demo application SwingSet2 and the JBits FPGA simulator by Xilinx. The inputs used for these two applications were interactive and consisted of navigating extensively through the application's GUI and performing standard program actions (e.g., loading a simulator and an FPGA configuration and performing simulation steps). Both of these applications exercise over 1400 Java system classes. Only 21 and 16% (for JBits and SwingSet2, respectively) of these classes were found to be NUI under our analysis: the rest can employ user-level indirection without any special treatment. Finally, we include in our suite the RMIServer sample application from Sun, in order to exercise networking system classes.

Thus, Table 6-1. confirms that native code is not a negligible part of real applications. Additionally, although the type analysis assumes the most general native code behavior that respects its assumptions, it is still sufficient for enabling safe indirection for the large majority of Java system classes used in actual applications. (Where safety is always contingent on non-violation of our heuristic assumptions by the native code. We later discuss how we confirm that our approach is indeed safe for these executions.)

Table 6-1. Type-based analysis of used system classes

Application	#classes	#native	%native	#NUI	%NUI
javac	167	21	13	62	37
jess	165	21	13	61	37
Mpeg audio	158	21	13	60	38
Antlr	209	21	10	67	32
Bloat	275	25	9	80	29
Chart	601	69	11	194	32
Hsqldb	295	26	9	83	28
Jython	263	20	8	76	29
Ps	175	18	10	60	34
Xalan	505	21	4	74	15
SwingSet2	1887	120	6	303	16
JBits	1442	124	9	306	21
RMI Server	415	37	9	109	26

6.5.2 Accuracy of Type Information

Recall that one of the heuristic assumptions of our type-based analysis is that the APIs to system functionality offer accurate type information. That is, we assume that native code does not discover type information dynamically: if a native method signature refers to type `A`, then it does not attempt to dynamically discover which particular subtype of `A` is the actual type of the object and to use fields or methods specific to that subtype. It is certainly common to pass instances of subtypes of `A` to the native method, but these should only be

accessed using the general interface defined by the supertype `A`. This assumption is in line with good object-oriented design.

Although the assumption is soundly motivated, there are certainly exceptions in real code. Nevertheless, such exceptions are fairly rare. To validate the assumption, we examined part of the implementation of native methods in Sun's JDK 1.4.2. We searched for the use of specific idioms throughout native method implementations and we examined in detail all native methods (109 of them) accepting as argument or returning as result an object with declared type `java.lang.Object` (the root of the Java inheritance hierarchy). In our study, we observed few violations of our assumptions. The most important ones are:

- reflection functionality routinely circumvents the type system, as expected. Reflection requires special handling in a user-level indirection environment.
- passing primitive arrays to native code is typically invisible to the type system. Several native methods accept an `Object` reference but implicitly assume that they are really passed a Java array of bytes or integers. This does not affect our analysis, as we consider primitive types and their arrays to be non-indirectible by default.
- a handful of methods have poor type information and violate our type accuracy assumptions. For instance, method `socketGetOption` in class `java.net.PlainSocketImpl` takes an `Object` as argument, casts it into a `java.net.InetAddress` and then sets one of its fields. (The `addr` field is set when the method returns the bind address for its socket implementation.) Similarly, native method `getPrivateKey` in class `sun.awt.SunToolkit` assumes that its `Object` argument is really a `java.awt.Component` or a `java.awt.MenuComponent` and dynamically discovers its actual type.

  These exceptions, however, are very rare, in our experience. A quick search of all native code in Java system libraries (for all platforms together) reveals just 69 uses of the JNI function `IsInstanceOf`, which is the main way to do dynamic type discovery in native

code. In contrast, there are about 5900 uses of the Java counterpart, `instanceof`, in plain Java code in the system libraries. (The total size of Java code in system libraries is roughly twice the size of C/C++ native code, so the discrepancy is not justified by the size alone.)

We, thus, feel that our heuristic assumption is well-justified. Even though the native implementation is free to circumvent the type system, we believe that in practice it is reasonable to assume that sufficient type information exists at the user/system boundary of languages like Java to allow a heuristic but fairly good type-based analysis. Clearly the analysis will not offer strict guarantees, but if it determines that a certain system class can employ user-level indirection, it is highly likely to be right. We quantify this likelihood for actual applications next.

6.5.3 Testing Correctness

Our type-based analysis attempts a heuristic solution to an unsolvable problem. Recall that if we treat native code as an adversary, there are no safe assumptions we can make, other than "all native code can directly access and modify all objects". This assumption invalidates every kind of user-level indirection. Nevertheless, in practice our heuristic, type-based approach works well. (Our experience with J-Orchestra was what first suggested to us that a type-based analysis is sufficient for ensuring safe indirection in practice.)

We dynamically analyzed the applications discussed above to confirm that the results of our type-based analysis are rarely, if ever, violated in practice. We instrumented a Java VM to observe all reads and writes to object fields performed inside native code. Then we checked whether fields of a class that we did not consider NUI are ever read or written inside native code. Of course, this experiment is just a test under specific inputs.

Our analysis results could still be violated by different program inputs. Nevertheless, given the amount and variety of tested code and inputs, we have high confidence in our observations.

Almost all applications listed in Table 6-1. exhibit accesses to Java object fields from inside native code. Some applications (especially the more graphics-intensive ones) have native code access the fields of objects of more than 50 different classes. Throughout all executions of the applications, we observed only two instances of access inside native code to objects of types that were not classified as NUI. Both cases represented native code implementation patterns in Sun's JDK 1.4.2 that violated our type-accuracy assumptions.

Specifically, the first case was that of method `populateGlyphVector` in class `sun.awt.font.NativeFontWrapper` (not a directly user-accessible class). The method accepts a `java.awt.font.GlyphVector` parameter but implicitly assumes that the true type of the parameter is `sun.awt.font.StandardGlyphVector` and proceeds to set specific fields of that class. This is a classic case where information is not present in the type signatures of native methods for no apparent good reason. (Upon further inspection, a couple of more methods in the same class also circumvent the type system for `GlyphVector` arguments.)

The second case was that of the constructor of class `sun.java2d.loops.MaskFill`. The constructor accepts a `java.awt.Composite` parameter but assumes its real type is `java.awt.AlphaComposite`. Although this is again a bad practice of obscuring information from the type system, at least in this case there is some code economy benefit from doing so: the constructor is only called in native code using dynamic method discov-

ery (i.e., reflection at the native level). Eliding the specific type information allows the constructor to be called by the same code as some other similar constructors.

In summary, our experience confirms that a type-based analysis is quite safe in practice. Although no guarantees can be offered (as the assumptions can be violated by the implementation of native methods) one can reasonably expect that the type analysis will be safe. In the absence of complete information on the behavior of native code, our analysis is a clear win. The alternatives are to either not support indirection for any system classes, or to leave the user with no assistance in determining the correctness of applying indirection.

6.6 Conclusions

In recent years, the high and growing popularity of high-level languages such as Java and C#, running on top of virtual machine-based runtime systems, has influenced the proliferation of user-level indirection techniques for achieving systems-level extensibility. The ability to transform a piece of software automatically and correctly by enhancing it with useful functionality such as logging, persistence, distribution, and others, relieves the programmer from the necessity of performing tedious and error-prone tasks by hand. However, the applicability of all such user-level indirection techniques is limited by the presence of native code. This chapter has studied ways that identify these limitations, in order to enable user-level indirection to be applicable as widely as possible. In the greater scheme, this chapter has generalized one of the technical contributions of this dissertation to the domain of user-indirection-based software systems, having made the following observations:

- Native code can invalidate any user-level indirection technique in the worst case. Although this is a standard observation for program analysis experts, it is a topic often completely ignored by implementors of user-level indirection mechanisms.
- A simple type-based analysis together with fairly general assumptions can help distinguish classes that are safely indirectible from those that are not. It is interesting that the type information at the user/system boundary would be sufficient for this purpose. It is the type system of modern OO languages such as Java that is directly responsible for enabling this analysis. In other words, the analysis would be impossible at the user/system boundary between C and Unix or Windows, in which most of the arguments to system library calls are unstructured pointers and byte buffers.
- These findings have the potential to be of value in the design of future runtime systems and environments, making the code running on top of them easier to indirect. Specifically, these findings pointed out the need for a runtime specification that would describe how system classes interact with their native platform-specific libraries. Having such a runtime specification, perhaps in the form of annotations of Java system classes, would make user-level indirection techniques safe from the possibility of being invalidated by native code.