The Limits of User-Level Code Transformation and How to Push Them

Eli Tilevich, Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
{tilevich, yannis}@cc.gatech.edu

ABSTRACT

User-level indirection is the automatic rewriting of an application to interpose code that gets executed upon program actions such as object field access, method call, object construction, etc. The recent research literature describes a number of user-level indirection techniques for enhancing the capabilities of applications without modifying their runtime system. Nevertheless, the applicability of all such techniques is constrained due to the presence of unmodifiable (native) code that cannot be indirected and can invalidate the assumptions of any indirection transformation. These problems are real: the native behavior of Java system classes, for instance, invalidates the transparency of recently proposed user-level indirection techniques. In this paper, we demonstrate the problem of employing user-level indirection when native code exists. We then suggest reasonable assumptions on the behavior of native code and a simple analysis to compute the constraints they entail for the applicability of user-level indirection. We show that the type information at the native code interface is a remarkably sufficient approximation of native behavior for inferring when user-level indirection can be applied safely. Finally, we discuss annotations for native code that will enable the accurate inference of constraints for user-level indirection.

1 INTRODUCTION

Much recent research has concentrated on *user-level indirection* techniques for transparently enhancing program capabilities. User-level indirection is the automatic transformation (instrumentation) of application and system code so that its execution characteristics are modified, without changing the underlying runtime system. Standard applications include transparent distributed execution [4][7][15][16][18][19], persistence [2][11][14], profiling [8], and logging [12]. Thus, user-level indirection is a language-level technique for achieving systems-level extensibility. The approach has become even more prevalent with the widespread use of virtual machines, such as the Java VM or Microsoft's CLR, as runtime systems for high-level languages. Compared to the straightforward approach of modifying the runtime system, user-level indirection has the crucial advantage of portability and ease of deployment on unmodified runtime systems. Running applications on modified versions of a platform-specific runtime system is hard and in some cases (e.g., embedded systems) even impossible. Yet if we achieve the same effect through code transformation, the resulting code can run on a variety of platforms on standard-issue runtime systems.

Despite the many user-level indirection techniques proposed, few researchers have clearly identified the limitations of the approach (and some have perhaps overstated its generality[1]). User-level indirection has to be transparent relative to the behavior of the original code. For instance, if we transform an application to log its method calling actions, the resulting application should behave identically, except for the logging. Nevertheless, all user-level indirection techniques have transparency limitations relating to the presence of native code that an application can access. Native code is opaque: it cannot be analyzed or modified without negating the platform-independence advantages of user-level indirection. Yet, native code has its own state, can hold references to user objects, can remember (alias) these references across invocations, and can use them for destructive updates of user-level state. This renders the code transformation incorrect (i.e., non-semantics-preserving) for all user-level indirection techniques in the literature and for most purposes of user-level indirection.

In this paper, we clearly demonstrate why different user-level indirection techniques are not semantics-preserving when native code exists. The problems occur in practice with common native code patterns, e.g., in the Java system classes. In a sense, this is an old problem of semantics-preserving transformations in the presence of opaque code. The same problem could be studied in the context of any language and runtime system. Nevertheless, modern high-level runtime systems are a natural platform for user-level indirection and introduce unique parameters (e.g., well-typed interfaces to system code). We discuss the issue from the perspective of runtimes for OO languages, such as the JVM and the CLR. We then examine what weak assumptions we can make regarding native code and what constraints we can enforce so that disciplined use of user-level indirection is correct. These weak assumptions are highly practical: for instance, they hold widely in existing native code in the Java system. The assumptions are sufficient to enable a simple type-based analysis to guarantee the safety of user-level indirection for the majority of Java system classes.

This work generalizes our previous research on J-Orchestra [19][20]: a system that automatically rewrites Java applications for distributed execution, yet enforces simple constraints that guarantee the correctness of the transformation under general assumptions on the native code behavior (e.g., no down-casting, no aliasing-at-construction, etc.). Additionally, this paper extends the

1. For instance, the authors of the TCH technique [5] argue that "TCH can be used automatically by any general instrumentation"; "[TCH has] the ability [...] to instrument all system classes"; "TCH allows even system classes with native dependencies to be rewritten for distributed execution".

basic J-Orchestra user-level indirection technique. We show that we can remove some of the main J-Orchestra constraints and obtain the freedom to use user-level indirection for many more system classes, at the expense of using a more complex indirection scheme.

Finally, we discuss the use of annotations that describe the behavior of native code so that constraints on user-level indirection can be inferred. This discussion is meant as a suggestion to designers of language runtime systems and proposes appropriate annotations for expressing analysis properties of binary code.

2 USER-LEVEL INDIRECTION AND ITS LIMITATIONS

We first describe user-level indirection to make clear why all different versions of the idea converge into using the same general approaches. Then we discuss why there are correctness limitations when native code is involved. 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). We generally use Java (i.e., Java language syntax, Java terminology, and JNI conventions) as our reference system. Even though we demonstrate program transformations in source code for readability, these transformations are generally performed at the bytecode level. In Section 2.3 we discuss the differences relative to the CLR and .NET technologies.

2.1 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 [4][15][16][17][18][19], persistence [2][11][14], profiling [8], and logging [12]. 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 bytecode-only system classes. Indeed, several systems [5][18][19] 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 [5] 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:

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

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

2.2 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, etc.) 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,[2] it is likely to

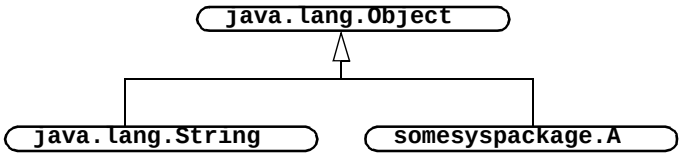

Figure 1(a): Original system classes hierarchy

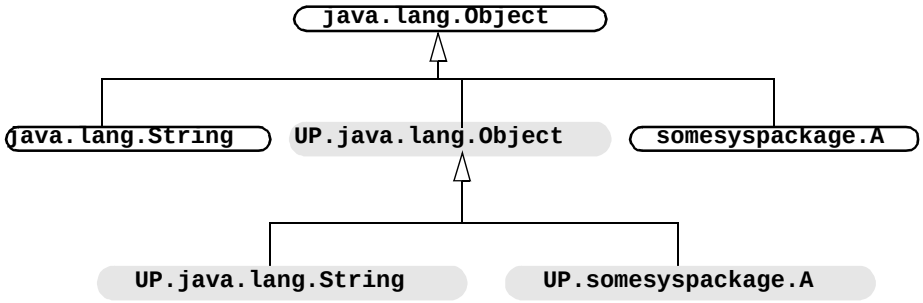

Figure 1(b): Replicating system classes in a user package ("UP")

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, etc.) cannot be intercepted using user-level indirection. Although it may seem that such state is low-level 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 (such as J-Orchestra [19], Pangaea [16], Addistant [18] or JavaSplit [4]) 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 [5]. (This approach is representative of other user-level indirection techniques, such as the one in J-Orchestra [19].) 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() { ... }
}
```

2. 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.

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.

2.3 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 paper, all our qualitative observations should apply equally well to the CLR, unless we explicitly note otherwise.

3 PUSHING THE LIMITS WITH WEAK ASSUMPTIONS

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, the results of the analysis can be exported by VM implementors together with the VM in a reusable form. In Section 5 we discuss an annotation language that is capable of expressing the results of such an analysis. However, before we attempt to convince all VM vendors to export information on their native implementations to allow for safe user-level indirection, we first examine a less ambitious idea. Instead of analyzing and annotating the native implementations, can we 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 can enable 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.

3.1 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 2.1 we distinguished several different kinds of events typically captured by user-level indirection: access to fields, method calls, constructor calls, etc. 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.

Our previous work on the J-Orchestra system used some weak assumptions on the behavior of native code and a simple type-based analysis to distinguish code that is likely to safely employ user-level indirection. In this way, we can exploit the rich type information of the Java system classes API. J-Orchestra uses user-level indirection in order to execute monolithic Java applications over a network of machines. Typically, the application is split in parts (consisting of user code and system classes) so that each machine handles a different hardware or software resource—e.g., the graphical input/output code may run on one machine, while the processing is done on a second and database access on a third.

Here we abstract away the specifics of the J-Orchestra approach so that it can be generalized to different domains. The approach 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. (See Section 6 for a discussion of reflection.) In Section 4 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. The base J-Orchestra rules for inferring the classes that have user-level indirection limitations are 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.

(The above rules represent the essence of the analysis but are not complete. For instance, we 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. 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. The numbers we later report are for the full version of the rules, however. Note that interface access does not impose restrictions since an interface cannot be used to directly access state.)

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.

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 2.2: 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 [13][19].

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.

3.2 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 2.2, 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 2. The example class `File` of Section 2.2 is now shown with a non-public field `field1`. `File` has a subclass `TXFile` with no native dependencies. Figure 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 3.

Note that in the case of the CLR we may need to use a slightly more complex transformation to get the same effect. As pointed out in Section 2.3, native code in the .NET framework can make assumptions about the memory layout of objects. Thus, in some cases it may not be possible to introduce new fields in existing classes. Instead, in our example, a new system class, `FileBridge`, can be added as a subclass of `File`. This class will just serve to export through public methods the non-public members of its superclass, `File`, to instrumented classes.

4 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 implementation 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.

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

4.1 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 1. The table shows how many of

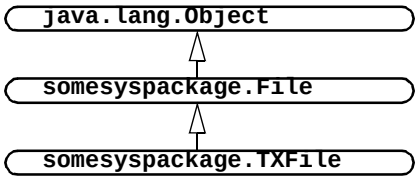

Figure 3(a): A File class hierarchy

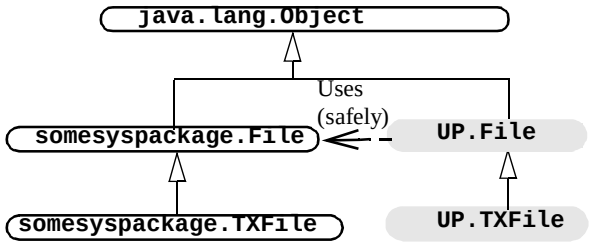

Figure 3(b): Removing subclassing restrictions

the system classes actually used by different Java applications are classified as NUI under our analysis of Section 3.2. 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. (The DaCapo suite has 3 more applications that we did not manage to run by paper submission time due to setup issues, such as library dependencies or unclear input files.) 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 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 assump-

tions, 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 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
Swing Set2	1887	120	6	303	16
JBits	1442	124	9	306	21
RMI Server	415	37	9	109	26

4.2 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, as we later discuss in Section 6.
- 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.

4.3 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 in Section 4.1 to confirm that the results of our type-based analysis of Section 3.2 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 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 discovery (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) there is a reasonable expectation 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.

5 PUSHING THE BOUNDARIES WITH CODE ANNOTATIONS

Although the analysis of Section 3 is sufficient for many practical purposes, there are three major reasons why we may want to go further. First, the analysis is heuristic: it is based on implicit assumptions that are never expressed by the system class programmers and can be violated without warning. Second, the analysis is purely type-based: it does not distinguish between different instances of the same class or between different fields of the same class. Third, there are domains in which it is important to handle the <8% of system classes that get classified as "unsafe to employ user-level indirection" by the rules of the previous section. For instance, we have already mentioned that J-Orchestra makes further assumptions about how state is shared among system libraries, in order to instrument applications so that file processing is done on a separate machine from user I/O or from sound output—even though these resources are all handled by native code and in theory could be interacting behind the scenes.

If one had information on how exactly native code libraries interact among them and with the bytecode of system classes, this would allow more flexibility in designing the user-level indirection machinery. Unfortunately, the straightforward way to get such information is to analyze the source code of the runtime system. This is complicated at best and unrealistic at worst (source code may not be available). The natural avenue for extending program analysis when source code is not available is to employ programmer-supplied annotations. These annotations form a language for communicating implementation insights. We propose that system classes be annotated to reflect how exactly they interact with native libraries.

A large body of work in compilers and type systems is applicable to this problem. For instance, Guyer and Lin [6] have presented an annotation language that can express alias analysis and abstract interpretation information. Such annotations can express very detailed information—e.g., that a certain method's result points to an object that aliases one of the method's arguments. Nevertheless, there is no universal annotation language: every annotation language is tailored to a specific kind of analysis. In our case, some of the detail is unnecessary. For instance, it is of little concern for user-level indirection how output values relate to input values. Instead, it is important to know whether the method affects any of the fields of a reachable object, or whether some state variable inside native code aliases an input object, etc.

Next, we discuss annotations that we consider capable of providing the most benefit, based on our experience with user-level indirection, and adapt them to this domain. Thus, this section is speculative and serves more as "food-for-thought" and as a good starting point for further discussion.

Fine-Grained annotations

The most straightforward kind of annotation that would alleviate the issues with user-level indirection is annotations that describe native method effects and special class semantics.

- `@aliases`
  `({@obj (source_path =`
  `[ret.]"qname"|hidden|transient,`
  `target_path = [ret.]"qname"), ...})`
  To describe the aliasing effects of a method, we can use annotations that detail how the method internally aliases objects reachable from its arguments or the fields of the current object. There are two kinds of aliases: ones that persist after execution of the method and ones that do not. The latter kind only restricts whether all aliasing of an object is observable through user-level indirection. The former kind, however, is useful in conjunction with any kind of program analysis. Aliasing information includes the aliasing source as a qualified name (*`qname`*): this is the user-level-reachable field that holds the alias (if applicable—`hidden` otherwise); and the aliasing target: the user-level-reachable object that is being aliased. For instance, the annotation
  `@aliases(@obj(source_path="field1",`
  `target_path="arg.fld"))`
  means that on return from the method, `field1` will be aliasing the object referenced by `arg.fld` when the method is called. The `ret` prefix on a qualified name means that the path is relative to the return value of the method. Additionally, the qualified name can be extended with type information to indicate any

downcasting performed by the native method (e.g., "(Der)arg.fld1" can mean that arg gets cast into a Der and has its fld1 aliased).

- @accesses ({@obj(path="qname", write=T|F),...}) Just like in the case of aliasing, a native method should declare what fields it accesses and how. The first goal of accesses annotations is to expose the limitations of user-level indirection: a field accessed inside native code is one that cannot employ user-level indirection transparently. Nevertheless, for some applications the annotation is enough to overcome these limitations. For example, if a piece of data is not aliased and we know it gets changed as a result of a native method, it is often correct to implement a consistency protocol by propagating the update after the method finishes executing.
- @special. This is a class-level annotation that states that a class has special semantics, even though it does not have any native methods.

Coarse-Grained Annotations

Coarse-grained annotations can exist either as a complement of fine-grained annotations or as a complete replacement. Fine-grained annotations (aliases, accesses) are powerful enough for interfacing with program analysis. Program analysis is valuable as it can overcome the limitations of the type-based approach of Section 3. For instance, different instances of the same class or different fields of the same class can be treated differently based on how they are used in the program.

Nevertheless, the problem with fine-grained annotations is that their initial cost of adoption is very high. First, a very expressive language needs to be standardized and the exact expressiveness of the language will always be a trade-off. Second, the annotations would be hard to produce by hand and the potential for error would be great.

Overall, given that our type-based analysis of Section 3.2 identifies just 8% of Java system classes as NUI, fine-grained annotations may be overkill—we may get most of the practical benefit with simpler techniques.

- @type_exact. This annotation can encode our second assumption of Section 3.1: the method does not use dynamically discovered type information to access extra fields of objects it manipulates. This information can also be supplied by fine-grained annotations (e.g., aliases) but the property is so useful and likely to hold, that it is probably best to have an explicit annotation for it. The annotation could be at the method level or even at the class level.
- @aliases ({@types (name = "typename"), ...})
  @accesses ({@types (name = "typename",
  write = T|F), ...})
  The aliases and accesses annotations can also be used on a per-type basis. If a type is aliased or accessed, all its instances and fields are considered aliased or accessed.

Separation of State

With either fine-grained or coarse-grained annotations, it is important for several user-level indirection applications to have information on how hidden state interacts with different pieces of native code. For instance, a type-based analysis can determine that classes Thread and Window both have native code dependencies, but it cannot determine that they likely access completely different state: everything behind the native boundary could in theory be interconnected. We do not believe that a fine-grained modeling of hidden state is necessary to get the benefits of state separation. Instead, a propositional approach, in which different kinds of state are labelled with different symbols, should be sufficient. Of course, ideally we would have type system support for distinguishing different kinds of state and ensuring they do not get accidentally mixed. The structuring of stateful libraries using monad types in the Haskell programming language is probably the ideal trade-off between expressiveness and power in this respect. Nevertheless, monads cannot be (easily) supported in object-oriented programming languages, so a manual solution is more appropriate.

- @state_kind(state = "label")
  We can label native methods or their classes with a @state_kind annotation. Then native code pieces of different state kinds can be assumed to not share state. In this way, objects that the analysis has determined to be aliased by one kind of native code are not automatically assumed to be accessible from other kinds.
- @stateless. A native method that does not result in any side effects and has a purely functional semantics can be annotated as @stateless. A stateless native method introduces no limitations with respect to user-level indirection. The only correctness obligation of the user-level indirection mechanism is to pack-unpack the input and output data of the stateless method. This means converting the input data from the instrumented form to the one in the original application before the method is called and converting it back on return. An example of a stateless native method would be hashCode in java.lang.Object.

Usability

From the implementation perspective, it seems that it is the developers of native libraries who are in the best position to provide annotations (either by hand or as a result of automatic analysis). Furthermore, these native code interaction annotations are unlikely to be platform independent. It seems natural that different virtual machine vendors should have significant leeway in which approach they choose to employ in implementing the interactions between their native libraries and the bytecode of system classes. Thus, even if the public API code of systems classes is standardized across different platforms, this requirement does not have to extend toward the annotations. Provided that the majority of indirection-enabling tools are generative in their nature, they should be fairly easy to re-rerun for each target platform.

6 CAVEATS AND APPLICABILITY

In this section we discuss miscellaneous topics relating to the applicability of our work.

Correctness under Reflection

For full transparency, user-level indirection techniques may need to replace all uses of the standard reflection mechanisms with uses of a special-purpose reflection library. The special-purpose library will emulate on the instrumented program the behavior of the standard reflection mechanisms on the original program. This is, however, just an engineering complication. In practice, user-level indirection techniques typically choose to ignore reflection because full transparency matters only in obscure cases. Common uses of reflection, such as instanceof checks, member type

recovery, field access and method calling, are safe with standard user-level indirection techniques. The unsafe parts usually consist of operations that depend on hard-coded class names, which may be replaced by names in an isomorphic class hierarchy under user-level indirection.

Special Purpose Functionality
The discussion of reflection is only one example of special-purpose handling of system functionality in classes that cannot safely employ user-level indirection. As mentioned in Section 2.2, user-level indirection techniques can be combined with special purpose replacements for the system code that cannot transparently employ user-level indirection. For instance, the thread creation and management (i.e., synchronization) functionality is low-level and employs native code. Systems that employ user-level indirection can have a special-purpose replacement of threading and synchronization code that works correctly with the indirection [4][20]. The same approach can be used with other low-level classes, such as those managing files, database access, network communication, GUI, etc. Thus, this paper can be viewed as an exploration of how far we can push general-purpose user-level indirection before we have to resort to ad hoc solutions.

Aspect-Oriented Programming
A natural application domain for user-level indirection is that of Aspect-Oriented Programming (AOP) [9]. AOP advocates separating distinct aspects of an application's functionality and composing them at the implementation level. Thus, the kinds of events captured by user-level indirection approaches are of particular interest from an AOP standpoint as *join points* for interposing extra functionality. AspectJ [10], the flagship AOP tool, concentrates exactly on capturing events such as field accesses, method calls, object construction, etc. Although newer versions of AspectJ compile directly to bytecode, current AspectJ compilers do not allow manipulating pre-compiled bytecode. Accordingly, join points in system libraries or inside native methods cannot be instrumented with AspectJ. Nevertheless, it is a stated goal in the AspectJ specification to allow the AspectJ semantics to apply transparently to all classes, yet allow implementations to diverge for practical reasons. We believe that our work clarifies what an AspectJ implementation can do while using general-purpose, non-VM-intrusive techniques, and thus paves the way for allowing AspectJ to interact with pre-compiled and system code.

7 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. In this paper, we studied ways to identify these limitations, in order to enable user-level indirection to be applicable as widely as possible. In the greater scheme, our paper offers several interesting elements:

- we show how 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.
- we show how a simple type-based analysis together with fairly general assumptions can help distinguish classes that can be safely indirected from those that cannot. It is interesting that the type information at the user/system boundary would be sufficient for this purpose. The type system of modern OO languages like Java is directly responsible for enabling this analysis. The analysis would be, for instance, impossible at the user/system boundary between C and Unix or Windows, where most of the arguments to system library calls are unstructured pointers and byte buffers.

We hope that our findings will be of value not just in the domain of user-indirection-based software systems but also in the design of future runtime systems and environments, making the code running on top of them easier to indirect.

References

[1] Anasua Bhowmik and William Pugh, "A Secure Implementation of Java Inner Classes", *PLDI 99 poster session*.

[2] Kumar Brahnmath, Nathaniel Nystrom, Antony Hosking and Quintin Cutts, "Swizzle Barrier Optimizations for Orthogonal Persistence in Java", proc. *8th International Workshop on Persistent Object Systems (POS8) and 3rd International Workshop on Persistence and Java (PJW3)*, 1998.

[3] Markus Dahm, "Byte Code Engineering", *JIT* 1999.

[4] Michael Factor, Assaf Schuster and Konstantin Shagin, "JavaSplit: A Runtime for Execution of Monolithic Java Programs on Heterogeneous Collections of Commodity Workstations", *2003 International Conference on Cluster Computing (CLUSTER'03)*.

[5] Michael Factor, Assaf Schuster and Konstantin Shagin, "Instrumentation of Standard Libraries in Object-Oriented Languages: the Twin Class Hierarchy Approach", *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2004.

[6] Samuel Z. Guyer and Calvin Lin, "An Annotation Language for Optimizing Software Libraries", *2nd Conference on Domain-Specific Languages (DSL)*, 1999.

[7] Bernhard Haumacher, Jürgen Reuter, Michael Philippsen, "JavaParty: A distributed companion to Java", `http://wwwipd.ira.uka.de/JavaParty/`

[8] Jarle Hulaas and Walter Binder, "Program Transformations for Portable CPU Accounting and Control in Java", Partial Evaluation and Semantics-Based Program Manipulation (*PEPM*), 2004.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, "Aspect-Oriented Programming", *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, "An Overview of AspectJ", *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[11] Gordon Landis, Charles Lamb, Tim Blackman, Sam Haradhvala, Mark Noyes, and Dan Weinreb, "ObjectStore/PSE: a Persistent Storage Engine for Java", proc. *2nd International Workshop on Persistence and Java (PJW2)*, p. 129-137, 1997.

[12] Han B. Lee and Benjamin G. Zorn, "Bytecode Instrumentation as an Aid in Understanding the Behavior of Java Persistent Stores", *OOPSLA 1997 Workshop on Garbage Collection and Memory Management*.

[13] Nikitas Liogkas, Blair MacIntyre, Elizabeth Mynatt, Yannis Smaragdakis, Eli Tilevich, and Stephen Voida, "Automatic Partitioning: A Promising Approach to Prototyping Ubiquitous Computing Applications", *IEEE Pervasive Computing*, 3(3): 40-47, July-September 2004.

[14] ObjectDesign Inc., *ObjectStore PSE/PSE Pro for Java API User Guide*, 1999.

[15] Michael Philippsen and Matthias Zenger, "JavaParty - Transparent Remote Objects in Java", *Concurrency: Practice and Experience*, 9(11):1125-1242, 1997.

[16] Andre Spiegel, "Pangaea: An Automatic Distribution Front-End for Java", 4th *IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '99)*, San Juan, Puerto Rico, April 1999.

[17] Andre Spiegel, "Automatic Distribution in Pangaea", *CBS 2000*, Berlin, April 2000. See also `http://www.inf.fu-berlin.de/~spiegel/pangaea/`

[18] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano, "A Bytecode Translator for Distributed Execution of 'Legacy' Java Software", *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[19] Eli Tilevich and Yannis Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning", *European Conference on Object-Oriented Programming (ECOOP)*, 2002.

[20] Eli Tilevich and Yannis Smaragdakis, "Portable and Efficient Distributed Threads for Java", *5th International Middleware Conference (Middleware'04)*.