Classifying Java Class Transformations for Pervasive Virtualized Access

Phil McGachey
Department of Computer Science
Purdue University
West Lafayette, IN
phil@cs.purdue.edu

Antony L. Hosking [*]
Department of Computer Science
Purdue University
West Lafayette, IN
hosking@cs.purdue.edu

J. Eliot B. Moss
Department of Computer Science
University of Massachusetts
Amherst, MA
moss@cs.umass.edu

Abstract

The indirection of object accesses is a common theme for target domains as diverse as transparent distribution, persistence, and program instrumentation. Virtualizing accesses to fields and methods (by redirecting calls through accessor and indirection methods) allows interposition of arbitrary code, extending the functionality of an application beyond that intended by the original developer.

We present class modifications performed by our RuggedJ transparent distribution platform for standard Java virtual machines. RuggedJ abstracts over the location of objects by implementing a single object model for local and remote objects. However the implementation of this model is complicated by the presence of native and system code; classes loaded by Java's bootstrap class loader can be rewritten only in a limited manner, and so cannot be modified to conform to RuggedJ's complex object model. We observe that system code comprises the majority of a given Java application: an average of 76% in the applications we study. We consider the constraints imposed upon pervasive class transformation within Java, and present a framework for systematically rewriting arbitrary applications. Our system accommodates all system classes, allowing both user and system classes alike to be referenced using a single object model.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming

General Terms Design, Languages, Measurement

Keywords Program transformation, Java, Object model

1. Introduction

Rewriting whole applications to augment them for transparent distribution or orthogonal persistence often relies on having all objects implement a single uniform object model. For example, orthogonal persistence relies on all instances having the capability to survive from one execution of the program to another, meaning that they must all have the capability of being stabilized for persistent storage. Similarly, transparent distribution relies on all instances having the capability of remote reference and invocation. Such rewrites most easily apply when extraneous barriers to transformation can be ignored, such as system dependencies that constrain what code can be rewritten (e.g., Java system classes or native code).

Here, we consider how to transform Java applications such that the vast majority of object instances can be manipulated via a unified object model that *virtualizes* every direct field access or method invocation in the original program. We convert those accesses and invocations into interface invocations in the transformed program. Virtualized manipulation permits straightforward interposition of desired functionality to implement extensions such as transparent distribution or orthogonal persistence. The only exceptions to pervasive virtualization in our scheme are those instances whose classes we determine need not be rewritten, either for optimization purposes or due to domain-specific constraints. In the absence of such constraints, we are able to handle all of the classes that comprise typical Java applications, including classes that are imported from the standard Java run-time environment (JRE) libraries. Handling these is particularly critical since the majority of classes that make up typical Java applications (76% on average for the standard benchmarks we consider) belong to the JRE. Yet, it is non-trivial to encompass these classes because Java's class loading restrictions and the presence of native code limit what parts of the JRE can be rewritten. When direct rewriting is not possible we rely instead on a series of transformation templates, tailored to the different characteristics of source classes, which allow us to implement the uniform object model throughout an application without directly modifying constrained classes.

We transform the classes of an application at class-load time, using a specialized rewriting class loader. Deferring transformations until load time permits flexibility in rewriting; we can transform classes differently depending on the circumstances. We additionally perform our transformations in a VM-agnostic manner. Thus, when implementing transparent distribution, we readily support a heterogeneous collection of hosts, so long as their class libraries offer the same APIs. We do not modify the VM in any way, so our implementation is portable and easily-maintained.

1.1 Contributions

Our contributions toward pervasive transformation of large applications include:

- An object model that allows virtualized access to the objects throughout a system, enabling the developer to interpose arbitrary code that tailors the functionality of the application executing on that system.

[*] This research partially done while on leave as Visiting Fellow, Department of Computer Science, Australian National University

GPCE'09 October 4–5, 2009, Denver, Colorado, USA.
Copyright © 2009 ACM 978-1-60558-494-2/09/10...$10.00

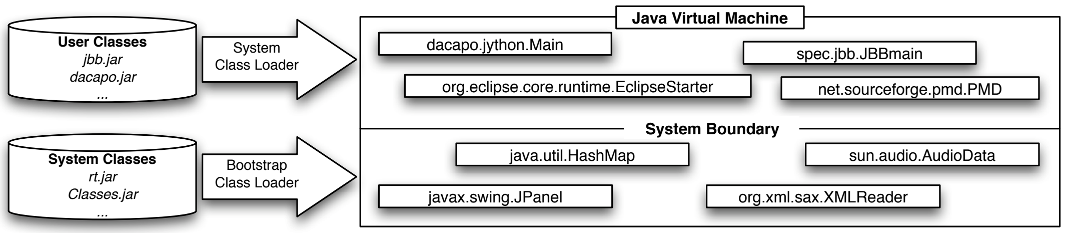
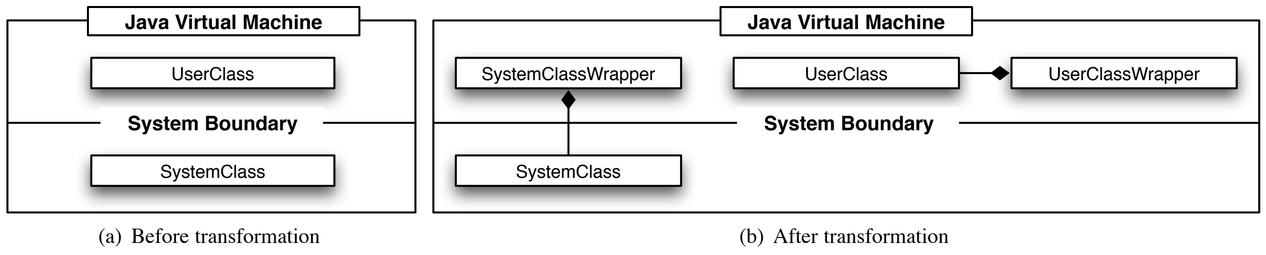


Figure 1. User and System Classes



(a) Before transformation

(b) After transformation

Figure 2. Transforming classes

- Enumerating major barriers to transforming applications, specifically the presence of native code and system library code that cannot be rewritten.
- Transformation templates that allow such classes to conform to our object model, and discussion of how transformed classes can inter-operate with unmodified code.
- The classification algorithm that determines which source classes should be transformed in which ways.
- Finally, an evaluation of the results of the classification process, with insights into the sources of overhead that accrue due to virtualized access. For SPECjbb2005 [6], we find that the fully-transformed benchmark produces 64% of the throughput of the unmodified benchmark, showing that our approach is feasible in practice.

1.2 Terminology

We use (and extend) various terms to characterize Java classes, which we now briefly define.

1.2.1 System versus user classes

Figure 1 gives a simplified overview of class loading within our system. We split classes into two sets, *system* and *user* classes, depending on the class loader that defines them. System classes can be thought of as those in the Java standard libraries, and so are loaded by the virtual machine's *bootstrap* class loader [3]. User classes, produced by the application developer, form the remainder of the application and are loaded by the user-defined *system* class loader. This distinction is vital when considering load-time transformation, as a user-level class loader can modify only user classes.

Within the Java VM itself we define the *system boundary* as a logical distinction between the two sets of classes; user classes exist above the system boundary, while system classes exist below. This abstraction is convenient when considering interaction between rewritten user and non-rewritten system code. We can enumerate the ways in which references can cross the boundary, and so ensure that rewritten references are never passed to system code.

1.2.2 Transformation

Figure 2 shows the implementation of *wrapping*; a common approach to handling system classes within a transformed application. In Figure 2(a) we see one system and one user class before applying any transformations. Figure 2(b) shows the result of wrapping each object. Class `SystemClassWrapper` contains a reference to the unmodified `SystemClass`. Since the wrapper was not generated by the bootstrap class loader it exists above the system boundary, with the reference crossing the boundary. In both cases, we refer to the original classes `SystemClass` and `UserClass` as the *base* class, while the two generated classes are *wrappers*.

Additionally, within our system we refer to `SystemClass Wrapper` and `UserClassWrapper` as *new* types. They are generated at load-time by our rewriting class loader, and thus can implement our object model. On the contrary, `SystemClass` and `UserClass` are *old* types, as they come from the original application. Both sets of types are necessary; new types implement the uniform object model that allows all classes to be referenced in the same manner, while old types can be passed safely to system or native code that has not been rewritten to be aware of the presence of generated code. We maintain a strict separation of the two sets of types. User code refers exclusively to new types, while system code refers exclusively to old.

1.3 Target Domain: Transparent Distribution in RuggedJ

We apply the pervasive virtualization transformations described in this paper to RuggedJ, our prototype transparent distribution framework for Java [4]. RuggedJ rewrites and distributes standard Java applications to run across a cluster of machines: we allow developers to deploy their applications onto heterogeneous and dynamically-changing computing platforms, enabling those applications to be re-targeted seamlessly for different distribution topologies.

A RuggedJ network consists of a number of *nodes*, each comprising a single instance of some Java virtual machine running on a hardware host in the cluster. Each node contains a bytecode rewriting class loader and a run-time library; the class loader supplies classes rewritten on-demand, while the run-time system interacts

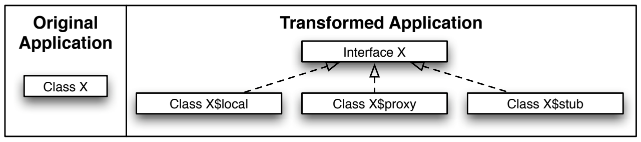


Figure 3. The RuggedJ object model

with other nodes to co-ordinate application distribution, as well as providing library functionality to the rewritten bytecode.

Distributing an application requires transforming all of its classes; not only do we add, remove, and modify code, we transform fields and method descriptors and generate new classes. Program transformations of this scale require not only modification of user code but also manipulation of system code. We distribute applications by abstracting object location: transformed application code manipulates local and remote objects transparently, executing the same code against those objects regardless of their location. Our object virtualization transformations allow the same flexibility between user and system classes; system and user objects can be manipulated in a uniform manner both by local and remote code.

2. The RuggedJ Object Model

The ability to distribute an application in RuggedJ stems from the uniform object model that we apply to all objects. Figure 3 shows the transformation of a single user class `X` to conform to the RuggedJ object model. While we discuss this object model in the context of distributed Java, the theory behind it is equally applicable to other domains that use virtualization of object accesses.

2.1 Generated Classes

For each class within the original application we generate three classes and one interface. Further, for distribution within RuggedJ we generate an additional three classes and an interface for the static parts of the class; this is necessary for distribution, but may not be for other applications of the object model. We discuss static data in Section 2.4.

The generated interface, `X`, contains the signatures of all the original instance methods, along with new accessor methods for all the original instance fields. It uses the same name as the original class—this simplifies later rewriting of classes that refer to the original class `X`, since we do not need to update type names in method signatures, field definitions, or casts. Interface `X` is implemented by three different representations of the members of the original class. The first, `X$local`, contains rewritten implementations of the instance methods of the original class, plus implementations of the new field accessor methods. In the rewritten application, an instance of `X$local` corresponds to an instance of class `X` from the original application: an `X$local` object holds all the data present in an old instance of `X`.

The second implementing class is used to refer to remote instances on other nodes: `X$stub` contains forwarding implementations of all the methods of the new interface `X`, which simply call the corresponding method on a remote `X$local` instance. Within a distributed application, the local and stub instances have a 1:*n* relation: any local object can be remotely referred to and invoked by stubs from the *n* nodes in the cluster.

The third (and final) new class is `X$proxy`. A proxy encapsulates a reference to either a local or stub (remote) instance, and its methods simply forward all calls to the target local/stub. Proxy indirection simplifies dynamic migration of instances to different nodes: a migratable instance is referred to by proxy, so upon migration only the reference in the proxy need be updated. Rewritten application code types all references to the three implementing classes using interface `X`. However we can bypass the proxy instance for objects that are known not to migrate. As all three classes implement interface `X` we can use them interchangeably without modification to any calling code. Determining which objects may require proxies is beyond the scope of this paper. In RuggedJ we use programmer input to determine how to partition an application across the network; other systems may use a whole-program analysis to find where proxies are required.

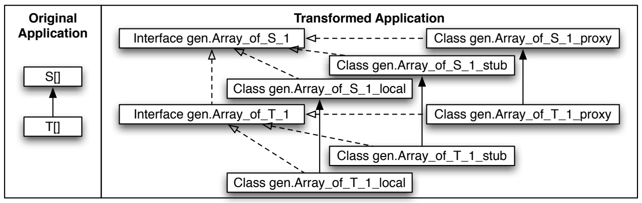

Original Application
S[]
T[]
Transformed Application
Interface gen.Array_of_S_1
Class gen.Array_of_S_1_proxy
Class gen.Array_of_S_1_stub
Class gen.Array_of_S_1_local
Interface gen.Array_of_T_1
Class gen.Array_of_T_1_proxy
Class gen.Array_of_T_1_stub
Class gen.Array_of_T_1_local

Figure 4. Generated array types

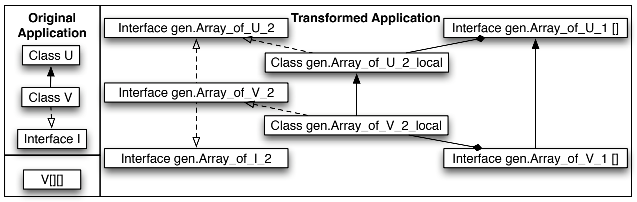

Original Application
Class U
Class V
Interface I
V[][]
Transformed Application
Interface gen.Array_of_U_2
Interface gen.Array_of_U_1 []
Class gen.Array_of_U_2_local
Interface gen.Array_of_V_2
Class gen.Array_of_V_2_local
Interface gen.Array_of_I_2
Interface gen.Array_of_V_1 []

Figure 5. Multi-dimensional arrays with interfaces

All of the classes in an application can be adapted to implement the RuggedJ object model. As we shall see, we use several techniques to generate local classes. However each implementation strategy produces a class that implements the corresponding interface, allowing proxy and stub classes to interact with any style of local class in the same manner. As the designs of stubs and proxies do not vary between implementation techniques, they are so straightforward as to be uninteresting. We therefore focus only on the local classes.

2.2 Referencing transformed objects

Within rewritten code, we exclusively refer to values with generated interfaces using that interface. This allows us to vary the implementations of these interfaces among several alternatives (local, proxy, and stub classes) without impacting code elsewhere in the system.

Additionally, we use interfaces as a means of maintaining the class hierarchy from the original application. While some of the transformations we present in Section 3.2 do not maintain the original relationship between their local classes, we ensure that their generated interfaces do. Thus, since we refer to such classes exclusively by interface, we can perform subtype and instance checks correctly.

2.3 Arrays

We convert array types to new array classes, which allow us to refer to them as we do any other transformed class. The new array classes conform to the RuggedJ object model; we generate an interface, local class, stub class, and proxy for each, as shown in Figure 4. A one-dimensional array type `T[]` is represented by an interface `Array_of_T_1`, while a two-dimensional array type `T[][]` is represented by `Array_of_T_2`. An array type comprises both an element type and the number of dimensions of the array, so we encode both of these properties in the name of the new array types. Java defines subtyping among array types having the same

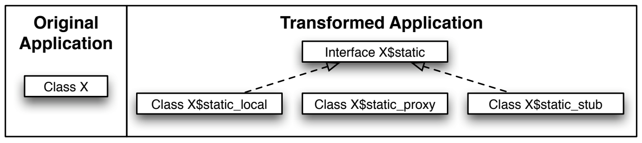


Figure 6. Handling static data for distribution

dimensions only if the element types are subtypes. We capture this by making any generated array class for a subtype directly extend the generated array class for its supertype (both having the same dimensions).

We implement arrays using wrapping: the generated array class wraps a regular Java array having the same component type as the wrapping array class. The implementation also provides methods to obtain the array `length` and to perform the standard operations that arrays inherit from `Object`, such as `clone`.

Figure 5 expands on the handling of arrays, showing the classes generated for a two-dimensional array type `V[][]` whose element type `V` extends `U` and also implements an interface `I`. We omit the new stub and proxy classes for clarity. This example highlights some interesting features of our generated classes.

Looking at the wrapped array within the local class, we see that the component type of the wrapped array is the same as that of the wrapper, with one less dimension. This mirrors the Java definition of arrays as a single dimension of components, where each component can be a sub-array. A useful consequence of this approach is that we do not place restrictions on the implementations of the components of the wrapped array, so long as they implement the appropriate interface. Thus, in RuggedJ, sub-arrays can be distributed across different nodes, regardless of the location of their enclosing array.

Figure 5 also illustrates that the old subtyping relationships between array elements and interfaces must also be represented in the new types. When passing array instances as arguments it is necessary for `Array_of_V_2` to implement `Array_of_I_2`. If an original method signature expects an array argument whose elements implement a given interface `I`, then in the rewritten new method we will expect an argument that implements some interface `Array_of_I_`*n* (for some dimension *n*), so capturing the proper type constraint. Within that new method all `aaload` operations are rewritten as `get` invocations on the argument. The type constraint ensures that any argument passed to the new method will have an appropriate `get` method to return a value implementing `I`.

2.4 Static Data

A class's static state presents a complication in a distributed setting, since an application must see just one version of the static state. Simply rewriting class fields as static fields in the transformed application will result in each node having a separate loaded class with that field, whose states will not be coherent across the nodes. We approach this issue through the use of *static singletons*. We extract the static parts of each class to form a single instance, which we handle as any other object within the system. The state of this singleton object represents the static state of the original class, and can be accessed from any node.

Since static singletons are required only to maintain a canonical version of static data, we do not need to create a singleton for a class that has no static fields. Our analysis shows that static singletons are required in only 18% of classes in the applications we studied.

Static singletons implement the RuggedJ object model as shown in Figure 6. Interface `X$static` complements the instance interface `X`; it contains the static members of original class `X`. We transform the static members of the original class into instance members of `X$static_local`, and use the RuggedJ run-time library to ensure that only one instance of that class is ever created. Thus, simply rewriting all static invocations to use the static singleton ensures that the static data is indeed unique.

The stub class `X$static_stub` performs the same remote access function as its instance counterpart. The final class in Figure 6, `X$static_proxy`, acts as a per-node cache for the appropriate static local/stub object. We never instantiate `X$static_proxy`; it simply contains a static field, providing a means to obtain a pointer to the appropriate static singleton.

RuggedJ's static singletons are an orthogonal concern to the main rewriting process that we have described. While the separation of statics is necessary within a distributed setting, it may not be when working in other contexts. The program transformations that we describe in this paper are equally valid with or without static singletons.

3. System Classes

The presence of system code within an application complicates the implementation of the RuggedJ object model. In this section we examine the issues involved when handling system code, and present the transformations that allow us to integrate system code into our object model.

We examine the restrictions imposed upon our system before we consider the (presumably) more interesting user code, because we find that system code is generally subject to more constraints than user code. Thus, as we will see in Section 5, the majority of constraints on user code are caused by dependencies on system classes.

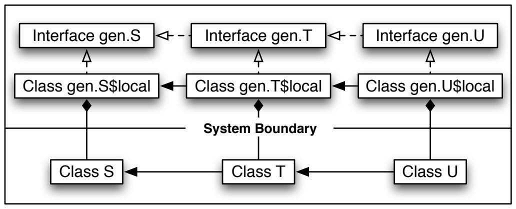
3.1 Barriers to transformation

The fundamental problem concerning the transformation of system classes relates to class loading. System classes are loaded by the virtual machine's bootstrap class loader, which we cannot modify without changing the VM. This would tie us to a particular VM implementation, making our system less portable.

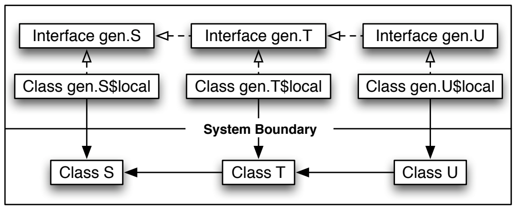
While we cannot implement a custom class loader for system code, we are able to perform limited rewrites on the majority of system classes. By implementing a Java VM Tool Interface agent [8] we can intercept classes before they are loaded. However we cannot perform the full range of transformations on these classes. For example, we can only modify existing classes rather than generating multiple new classes. Additionally, and more crucially, we cannot insert references to non-system classes. Due to the hierarchial nature of Java class loading (as discussed by Liang and Bracha in [3]) a class can reference only those classes loaded by its own, or a parent of its own, class loader. Thus a system class loaded by the bootstrap class loader can reference only other system classes.

We do, however, make use of a JVMTI agent to perform some minor modification to certain system classes within the application. The implementation of some transformations, for example, is complicated by Java's access control mechanism; if we change the package to which a class belongs, we can no longer access other classes with default access in the original package. We use a JVMTI agent to bypass these restrictions. Such a modification does not require reference to any additional classes, and does not alter program semantics, because the access control was checked statically at compile time.

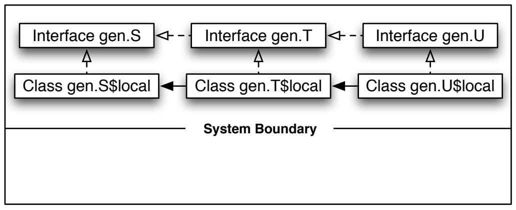
A second barrier to rewriting system classes is that some of these classes are effectively hard-wired into the VM. The bytecode that represents classes contains direct references to `java.lang.String` and `java.lang.Class`; both appear in the constant pool of a class file, and can be directly accessed using the `LDC` bytecode (that directly loads a constant to the stack). Again, changing the representation of these classes would require modi-



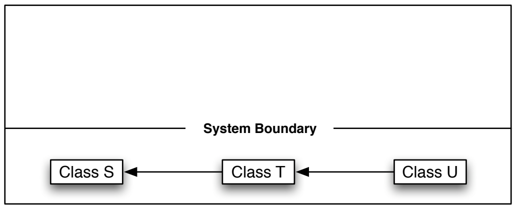
(a) Wrapping class hierarchy



(b) Extending class hierarchy



(c) Promotable class hierarchy



(d) Direct class hierarchy

Figure 7. Class hierarchies

fying the VM to understand the modified versions, which violates our goal for being able to run on any (unmodified) Java VM.

Interestingly, native and reflective code do not present any difficulty at the system level. Both types of code could break a system that transforms classes (and, indeed, must be accounted for within user code). However, since we do not rewrite system code, native and reflective operations perform as they would in an unmodified system.

3.2 Rewriting Templates

Our class transformations use four basic techniques to obtain new types, shown in Figure 7:

- The local instance of a *Wrapping* class holds a reference to a paired instance of the old type.
- *Extending* classes implement the object model through subtyping, with the generated local class extending the original system class.
- *Promotable* classes are not referenced by native code or by any other system classes, and so can be turned into user classes.
- *Direct* classes are not transformed, and so do not conform to the object model, solely because it does not make sense to for the target domain (the other transformations can be applied to such classes, but would result in unnecessary overhead for the target domain). In a distributed system, immutable objects such as `Integer` are an example; in a persistent system, open network connections would likely be Direct.

3.2.1 System Wrapping

Wrapping is the most straightforward of the transformation templates and is shown in Figure 7(a). In this approach, a set of classes are generated above the system boundary, in a special user-level package chosen to prevent name conflicts. For conciseness we refer to this package as `gen`. The base class is loaded by the bootstrap class loader, and is not modified. The local class contains new-type implementations of all the methods of the base class, each of which translates the arguments from new to old, invokes the method on the wrapped base object, then performs an old-to-new translation on the return value if necessary. In this way a given object can be referred to by new type above the system boundary, and by old type below.

Unwrapping objects when passing from user to system code is a trivial operation. However, we must be more careful when performing the inverse; wrapping objects that are passed from system to user code. In this case we need to ensure that a given object that has previously been wrapped is reunited with its original wrapper; to do otherwise would create two wrappers for a single base object, which would not preserve identity. To this end, we maintain a hash table against which we check for existing wrappers before generating a new one.

As with all classes that conform to the RuggedJ object model, wrapping classes maintain the inheritance hierarchy of the original through their generated interface. That the local classes also subclass the relevant local class is merely a convenience—if they did not, every wrapper would have to implement redirect methods for the methods of every superclass, rather than just those in its base.

The System Wrapping template can be considered the "universal solvent" for system classes. We can generate wrappers for any system class, which ensures that all objects in the application can conform to our object model. Unfortunately, the System Wrapping template also carries the highest overhead (as objects must be wrapped and unwrapped, which can be expensive), making the other templates more desirable.

3.2.2 System Extending

The System Extending template is an alternative means of handling system classes that eliminates the overhead of unwrapping. Under this technique, the generated local class extends the original base class, as shown in Figure 7(b). The generated interface and local class conform to our object model, while the base class remains unchanged. Note that in this case there is no inheritance relationship between the local classes; this is not important because the interfaces maintain the class hierarchy above the system boundary, while the base classes maintain it below.

An extending class can be passed to system or native code without any conversion process, since it extends the unmodified base. However we cannot create a new instance of an extending class within system code (as we cannot rewrite the allocation site to refer to `T$local` rather than `T`). This limits the applicability of this template to system classes that are only ever allocated above the system boundary. Further, while we obviously cannot extend `final` classes, we can also not override `final` methods. This may be an issue if a `final` method includes an old type as an argument or return value; the object model requires that such methods be overridden in order to be called by user code, which only uses new types. Thus, while the System Extending template is preferable to System Wrapping, due to its lower overhead it can be used only in limited cases.

3.2.3 Promotable

Promotable classes are a subset of system classes that are not referenced by any other non-Promotable system class or by native code. In this case we know that any reference to a Promotable class will either be in user code or in other Promotable classes. We can therefore move Promotable classes above the system boundary (by renaming their classes to form part of the `gen` package), and treat them as we do any other user class. Promotable classes often exist in cliques within the system libraries, with no external uses from other classes in the libraries. An example that we have encountered is the Java XML processing library. If an application uses XML processing, much of the library is loaded into the VM. However these classes refer only to one another. Thus, we can *promote* these classes en-masse.

The structure of a Promotable class is shown in 7(c). This is the most straightforward implementation of the object model, with each local class implementing its interface. While the inheritance hierarchy is maintained by generated interfaces, the local classes retain the original relationship. In the System Wrapping and Extending templates the actual method implementations were located in the base classes, Promotable local classes contain complete implementations of all their methods. Thus, Promotable classes must extend their parent so as to have their parent's methods available.

The Promotable template is similar to the Twin Class Hierarchy (TCH) approach proposed by Factor et al. [2], in that it loads system classes into the user space in order to perform transformations. However there is one important difference: the TCH system allows both modified and unmodified versions of the code to exist within a VM. We promote only those classes that are not used by other system code, so the promoted version is the only one in the system.

3.2.4 System Direct

The final set of classes, System Direct, do not conform to the RuggedJ object model. This template exists as an optimization; as we have seen, any class can conform to the object model through the System Wrapping template. However there are classes for which it is not necessary to conform to the object model. For example, when distributing an application with RuggedJ, we do not want to transform immutable objects. If we know that an object will never change, we can replicate it on multiple nodes, and eliminate the overhead of remote method calls. Similarly, there are classes that are closely tied to the individual VM (such as `java.lang.Class`) that do not make sense to reference remotely. Similar examples can be imagined for other contexts in which these transformations may be used.

Those classes we designate to be System Direct are not transformed in any way (as shown in Figure 7(d)). As such they do not incur any overheads, and can be freely passed between system and user code, as well as to native methods. However, since they do not conform to the RuggedJ object model, they cannot be modified to extend the original application's functionality.

3.3 Subtyping

Since all of the transformation templates described above rewrite classes differently, we cannot freely "mix and match" techniques between super- and subclasses. Each rewriting technique therefore imposes restrictions on the classes of its hierarchy.

Since System Direct classes do not conform to the RuggedJ object model, we must ensure that they have only other Direct classes in their hierarchy. To do otherwise would violate our rule that inheritance is maintained through interfaces; a Direct class has no interface, and so cannot fit into this scheme.

Likewise, System Wrapping classes can have only other Wrapping classes in their hierarchies. A Wrapping class cannot extend an Extending or Promotable class in case it is returned to user code from system code. There would be no way to produce a new-type representation of the Extending or Promotable superclass. The argument as to why a Wrapping class can only be extended by other Wrapping classes is similar. An Extending class that extends a Wrapping class removes our ability to translate from an old type to a new. In the case of a Promotable subclass, the local class would have to subclass the Wrapping subclass (since a Promotable object does not have a base class). This relationship would be lost when the base class was unwrapped.

A System Extending class can extend only another Extending class, since the local class must directly extend the base, and we cannot change the superclass hierarchy of the base class. However an Extending class can act as the superclass for a Promotable class; the System Extending template does not require unwrapping, so a Promotable local class can extend a System Extending local class without any loss of information should the object be passed to system or native code. This further indicates the usefulness of the System Extending template over System Wrapping. Promotable classes offer more options when extending an application's functionality, and by increasing the number of Extending classes, we likewise increase the number of potentially Promotable classes.

Our discussion of subtyping must also consider the original interfaces implemented by classes (as opposed to those generated as part of the RuggedJ object model). We rewrite interfaces in much the same way as classes: user-level interfaces contain signatures using new types, while system-level interfaces contain old types. Thus, system-level interfaces must be System Direct (if they contain only primitive or Direct arguments and return values) or System Extending (if they contain Extending, Wrapping, or Unconstrained arguments).

3.4 Classification

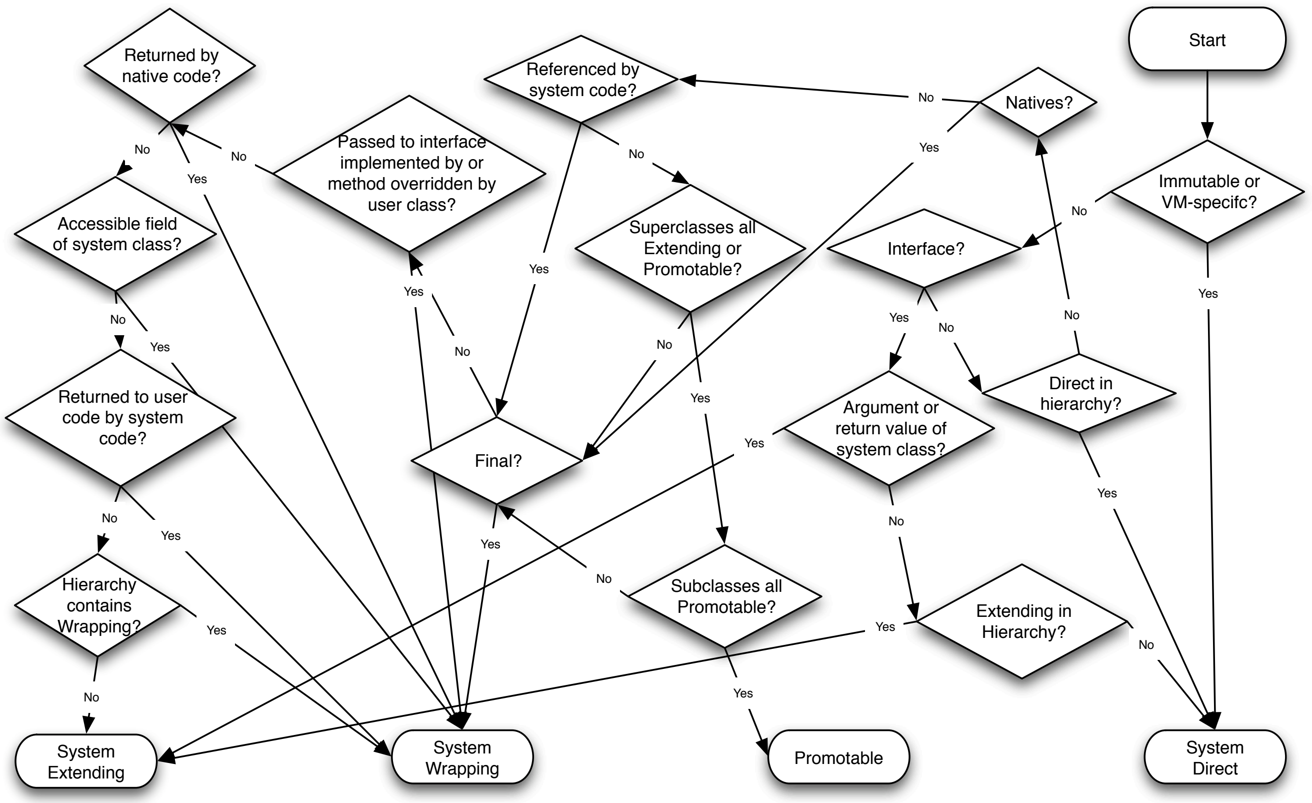
We refer to the process by which templates are chosen for each class as *classification*. A given class's classification may be determined by its subclasses or its references from elsewhere in the system, so we require knowledge of the entire application. We run the classification algorithm only on the classes that make up the application; analyzing the entire Java class libraries would introduce false dependencies, and limit our flexibility in transforming the application. We need to run the classification only once per application for a given set of class libraries.

We arrange the various classification templates using a partial ordering. Direct classes are handled first, as they are an optimization and otherwise fall into at least one other classification. Next we find Promotable classes, which maximize the flexibility of our rewrites, then Extending classes that handle the remaining classes with less overhead. Wrapping classes account for the remainder.

The algorithm is iterative, since changes to the classification of one class may affect others. We present the algorithm as a decision graph, which produces the classification for a given class, assuming that all other classes have already been correctly classified. To generate a full classification, we simply run the algorithm until a fixed point is reached. The decision graph for system classes is shown in Figure 8. Note that this graph was designed for RuggedJ, with Direct classes chosen due to immutability. In other contexts the determining factors for Direct classes may be different.

4. User Classes

The transformation of system classes constrains that of user code. As we discussed in Section 3.3, the classification of a given type can affect the classification of its super- and sub-classes. This requirement extends above the system boundary, meaning that we need to create equivalent versions of the four templates within user classes. Additionally, native code can be present in user as well as system code, which limits our ability to rename and rewrite classes.



Start
Immutable or VM-specifc?
Natives?
Referenced by system code?
Returned by native code?
Passed to interface implemented by or method overridden by user class?
Superclasses all Extending or Promotable?
Interface?
Accessible field of system class?
Direct in hierarchy?
Argument or return value of system class?
Returned to user code by system code?
Final?
Subclasses all Promotable?
Extending in Hierarchy?
Hierarchy contains Wrapping?
System Extending
System Wrapping
Promotable
System Direct
Yes
No

Figure 8. Classification for system classes

The four templates for rewriting user code closely mirror those for system. Classes can be User Wrapping, User Extending, User Unconstrained (the user-level equivalent of Promotable), or User Direct. As might be expected, occurrences of user-level native code or the subclassing of system classes are rare. As we show in Section 5, the vast majority of user classes are either User Direct or User Unconstrained.

4.1 Rewriting

User code differs from system code in one important manner: the classes are loaded by our user-level class loader, and so can be rewritten. This has implications for User Direct classes, as well as the base classes for User Extending, and Wrapping.

When rewriting user code, we define two invariants:

1. Values with generated interfaces (User Wrapping, Extending or Unconstrained) are always typed using that interface. This allows us to vary the implementations of these interfaces among several alternatives (as discussed in Section 2). If we know that these instances will always be manipulated through the interface methods then any implementation of those interfaces is safely encapsulated and we can freely decide on that implementation without worrying if that decision impacts other code.
2. User code exclusively refers to new types. By strictly ensuring that all rewritten code uses new types, we define a clear separation between old and new types. We can maintain this invariant because instances cross the system boundary in well-defined places (passed as arguments, returned from methods, etc.). Thus, we never need to check dynamically if an instance is of an old or new type; the context from which the instance is referenced (system or user) decides statically if the instance has an old or new type.

We occasionally break the second invariant to optimize base classes. However, these violations are always localized transformations (an old-type reference never escapes the method in which it is used), and so do not impact the system as a whole.

4.2 Native and reflective code

When transforming user code, we must make allowances for native code (for which we do not assume that we have source code) and for reflective code. We observe that either native or reflective code can break any large-scale series of transformations by introspecting on any class in the system. Should a class, field, or method be renamed or removed, hard-wired assumptions in native or reflective code may fail. We accept that an adversarial programmer, or one that makes extensive use of such code, can disrupt our system. We focus instead on permitting the widest possible range of common usages of both native and reflective code.

In the case of reflection, we do this at run time by intercepting reflective methods that refer to rewritten code and converting the results to the appropriate new types. In the case of native code, we exploit the heuristics laid down in J-Orchestra [11] that determine which classes are most likely to be accessed by native code, and ensure that they conform to the User Direct, Wrapping, or Extending templates. This way they retain a base object upon which native code can operate. They define classes with native methods to be *unmodifiable*, as well as the types of their fields and superclasses (dynamic dispatch can result in calling an overridden method indirectly from native code). These heuristics are adequate for the applications we consider.

4.3 Base Classes

User Wrapping and Extending classes are largely similar to their System equivalents, with the difference that their base classes are above the system boundary and so can be rewritten. Following the second invariant, we rewrite the method signatures and bodies of the base class to use new types rather than old. This simplifies the local classes that wrap or extend the base, since they do not have to translate between old and new types.

However, since user-level base classes may be passed to natives or system code (typed as interfaces or system-level superclasses), a base class must retain the *signature* of its unmodified original. New fields and methods may be added and the bodies of methods may be rewritten, but the class cannot be renamed, and its fields and methods must retain their original names and types. This violates our second invariant, that user code exclusively refer to new types.

We overcome this for methods by providing old-type implementations that simply redirect to their new-type equivalents. For fields this is more difficult. We ensure that any field that may be accessed by native code is not classified as User Unconstrained by the definition of unmodifiable classes above; a field of an unmodifiable class is itself considered unmodifiable, and so can not be classified as User Unconstrained. We observe that system code cannot directly access the fields of user classes, since they are loaded by different class loaders. Of the remaining templates, Direct and Extending classes are trivially compatible with system and native code (although we must type Extending classes as their base, and then cast upon use in user code). User Wrapping classes are also typed by their base, but since the wrapper is a separate object, we maintain a cached copy of the wrapper as an additional field. System or native code use the base class, while user code uses the new wrapper field. Note that the casting and wrapping of fields is required only in the base class itself; all other user classes refer to the object by interface and so can never access the field directly.

Another violation of our invariant occurs when a method accesses its `this` pointer. The type of the `this` pointer in a base class is an old type. We must therefore convert the reference to a new type, either by casting if it is a User Extending class or by wrapping if it is User Wrapping. This way the invariant is maintained. There are, however, some situations in which this is not desirable and some in which it is not allowed. If the `this` reference is loaded to the stack in order to execute a field access, for example, we would rather perform the access directly rather than going through the `get` method of the interface. More importantly, if the pointer is loaded in preparation for a superclass constructor call (as required in every constructor) it would be incorrect to wrap the reference. Doing so would lead to the constructor being called on the wrapper rather than the base, which would cause a run-time error.

We determine which `this` references to convert using a def-use analysis. If the reference escapes the current method (by being passed as an argument or stored as a field) we convert it, otherwise we do not. While this violates our invariant that rewritten code exclusively refers to new types, it does so only in a localized manner. Note that we can also use this optimization when accessing local fields within Unconstrained classes.

4.4 Classification

The classification of user code follows a similar approach to that of system classes. Figure 9 shows the decision graph for user classes. The user classification process uses the same ordering as the system; Direct classes are handled first, then Unconstrained, Extending, and Wrapping.

5. Evaluation

We evaluate our classification system using experimental results obtained from RuggedJ. We examine the output of our classification algorithm on a variety of benchmark applications, and provide some insight into the sources of overhead introduced by our system.

5.1 Configuration

Since the focus of this paper is the rewriting process within RuggedJ rather than the distribution process, we limit ourselves to describing performance on a single-node network. This allows us to analyze the overheads of the rewriting process without the additional complication of network interaction. We do not, however, optimize our implementation based on this network configuration. While a single-node network will never need to reference an object remotely or to perform migration, we do not disable the generation of stubs and proxies, and perform all rewrites as we would in a distributed system, referring to objects via our new interfaces, etc.

All results were generated on an Apple computer, using Mac OS X 10.5.6, and version 1.6.0_07 of Apple's Hotspot-based Java VM. This affects the results of the classification; different implementations of the standard class libraries may produce slightly different classifications. The test machine was configured with a 2.16GHz Intel Core 2 Duo processor and 2 GB of RAM.

5.2 Classification

We ran the classification algorithm on applications from different benchmark suites, producing the results shown in Table 1. The first block consists of ten benchmarks from the the DaCapo suite (version 2006-10-MR2 [1]). The remaining benchmarks come from the Standard Performance Evaluation Corporation: nine from SPECjvm2008 [7], plus SPECjbb2005 [6]. To obtain an accurate count of the classes referred to by the DaCapo applications, we analyzed them without the DaCapo harness. This way we classified only those classes referred to by the application, not by the harness.

As Table 1 shows, the majority of classes (74% on average) in an application belong to the standard libraries. This is due to the degree of interaction between system classes: a single reference can cause a large closure of classes to load. This strongly demonstrates the need to handle system classes within a rewriting system.

The user classes are split between User Direct (8% of the total application) and User Unconstrained (9% of the application). Very few classes are User Extending or User Wrapping. There was no user-level native code in the applications we studied, so these two classifications were used only for user classes that extended system classes. We see that only four classes in any of the benchmarks were classified as User Extending. While the number of User Extending classes seems insignificant, we must retain the classification template for these classes. Recall that an Extending class cannot extend a Wrapping class, so eliminating the User Extending template causes more system classes to be Wrapping rather than Extending, which we wish to avoid due to the wrapping overhead.

Below the system boundary, System Wrapping classes are the most common, representing 42% of the applications on average. This can be attributed to the need to wrap objects that are passed or returned to user code. System Extending classes are less common, representing 14% of classes, while 14% are System Direct. Finally, 1% of classes on average can be promoted.

5.3 Performance

When considering the performance of our system, we focus upon steady-state behavior. While start-up time (including class loading and hence rewriting time) is a major factor for small applications, the usefulness of rewriting such applications is limited. With RuggedJ, for example, there is little to be gained by distributing an application whose running time is dominated by start-up costs.

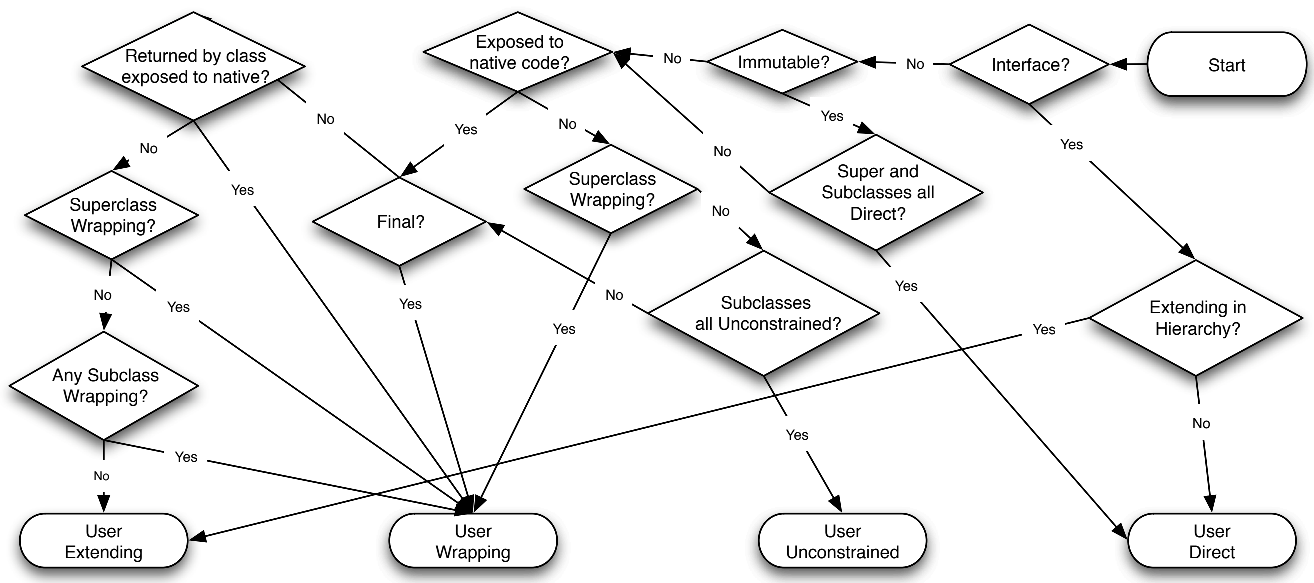


Figure 9. Classification of user classes

Benchmark	User	UD	UU	UE	UW	System	SD	P	SE	SW
antlr	140 (16%)	47 (5%)	90 (10%)	0 (0%)	3 (0%)	721 (83%)	147 (17%)	10 (1%)	132 (15%)	432 (50%)
bloat	270 (26%)	74 (7%)	193 (18%)	0 (0%)	3 (0%)	755 (73%)	152 (14%)	4 (0%)	136 (13%)	463 (45%)
chart	302 (18%)	130 (7%)	153 (9%)	0 (0%)	19 (1%)	1365 (81%)	222 (13%)	10 (0%)	305 (18%)	828 (49%)
eclipse	1847 (70%)	770 (29%)	1019 (38%)	0 (0%)	58 (2%)	781 (29%)	209 (7%)	24 (0%)	138 (5%)	410 (15%)
fop	800 (48%)	286 (17%)	499 (30%)	0 (0%)	15 (0%)	849 (51%)	176 (10%)	19 (1%)	165 (10%)	489 (29%)
hsqldb	145 (15%)	54 (5%)	84 (9%)	0 (0%)	7 (0%)	783 (84%)	164 (17%)	4 (0%)	153 (16%)	462 (49%)
jython	655 (45%)	75 (5%)	571 (39%)	0 (0%)	9 (0%)	792 (54%)	162 (11%)	5 (0%)	144 (9%)	481 (33%)
luindex	112 (12%)	43 (4%)	66 (7%)	0 (0%)	3 (0%)	750 (87%)	149 (17%)	4 (0%)	136 (15%)	461 (53%)
lusearch	149 (17%)	51 (5%)	94 (10%)	0 (0%)	4 (0%)	727 (82%)	150 (17%)	4 (0%)	133 (15%)	440 (50%)
pmd	557 (41%)	260 (19%)	291 (21%)	0 (0%)	6 (0%)	796 (58%)	199 (14%)	11 (0%)	139 (10%)	447 (33%)
xalan	480 (36%)	189 (14%)	276 (21%)	0 (0%)	15 (1%)	818 (63%)	192 (14%)	34 (2%)	148 (11%)	444 (34%)
compiler	297 (12%)	151 (6%)	122 (5%)	0 (0%)	24 (1%)	2037 (87%)	339 (14%)	103 (4%)	442 (18%)	1153 (49%)
compress	299 (15%)	152 (7%)	126 (6%)	0 (0%)	21 (1%)	1618 (84%)	284 (14%)	98 (5%)	340 (17%)	896 (46%)
crypto	288 (14%)	147 (7%)	120 (6%)	0 (0%)	21 (1%)	1672 (85%)	297 (15%)	106 (5%)	342 (17%)	927 (47%)
derby	1146 (39%)	483 (16%)	605 (21%)	4 (0%)	54 (1%)	1729 (60%)	321 (11%)	103 (3%)	371 (12%)	934 (32%)
mpegaudio	317 (16%)	164 (8%)	132 (6%)	0 (0%)	21 (1%)	1638 (83%)	290 (14%)	98 (5%)	348 (17%)	902 (46%)
scimark	288 (15%)	146 (7%)	120 (6%)	0 (0%)	22 (1%)	1615 (84%)	283 (14%)	98 (5%)	339 (17%)	895 (47%)
serial	309 (15%)	161 (8%)	126 (6%)	0 (0%)	22 (1%)	1668 (84%)	290 (14%)	98 (4%)	360 (18%)	920 (46%)
sunflow	396 (19%)	201 (9%)	173 (8%)	0 (0%)	22 (1%)	1633 (80%)	284 (13%)	98 (4%)	345 (17%)	906 (44%)
xml	290 (12%)	147 (6%)	121 (5%)	0 (0%)	22 (0%)	1944 (87%)	338 (15%)	31 (1%)	403 (18%)	1172 (52%)
SPECjbb	69 (4%)	32 (2%)	33 (2%)	0 (0%)	4 (0%)	1466 (95%)	262 (17%)	157 (10%)	277 (18%)	770 (50%)
Geo. Mean	19%	8%	9%	0%	0%	74%	14%	1%	14%	42%

Table 1. Classification by benchmark

Therefore, we will concentrate here on SPECjbb2005. As well as being one of the more complex benchmarks that we evaluated, it also exhibits measurable steady-state behavior.

We ran an unmodified version of SPECjbb2005 in single-user mode ten times, using the server version of the Hotspot VM. The benchmark was configured to use four-minute timing runs for eight warehouses. We averaged the benchmark score, representing the steady-state throughput. We then ran the benchmark under RuggedJ using the same configuration, again averaging the scores. Overall our rewritten system produces 64% of the throughput of unmodified SPECjbb2005. By running the system under the YourKit Java profiler [13], we found that the overheads of the rewritten system come from two sources: Wrapping classes and proxies.

By far the largest overhead came from wrapping those classes that were passed or returned from system to application code. 19% of the execution time within the timing periods was spent wrapping instances of system classes for use by application code. Of that time, 67% was spent in hash-table lookups determining whether an object had been wrapped previously. Additionally, 10% of the wrapping time was spent reflectively creating wrappers for objects that had not previously been wrapped.

Another 10% of the timing period was spent executing proxies, particularly the proxy objects for one-dimensional `int` arrays. SPECjbb2005 contains several methods that iterate over large arrays, making the overhead for indirection particularly obvious for these objects. However the majority of the performance overhead came from the proxy, rather than the local class. Methods of local array classes represented less than 1% of the execution time. Therefore this overhead could be substantially reduced by eliminating proxies for arrays that are known to be local.

6. Related Work

The issue of rewriting system code has been considered in the past. The Twin Class Hierarchy (TCH) approach of Factor et al. [2] copies relevant system classes into a user-level package, which can then be rewritten, and is referred to by rewritten user code. Because the original system classes remain unchanged, any instrumentation inserted into the rewritten versions can safely refer to system classes without affecting the statistics gathered or causing an infinite loop. The TCH system does not allow rewritten system classes to interact with the original classes below the system boundary, making it too limited for our needs. Additionally, the TCH approach requires custom wrappers for all native methods. This approach does not scale, and could require that separate wrappers be written for different implementations of the standard class libraries, compromising ease of deployment over heterogeneous Java VMs.

The Automatic Test Factoring system [5] produces “mock” versions of objects which return memoized results from a previous measuring run, allowing developers to speed up the testing of individual application components. Their system uses the same interface technique that allows us to refer to proxy and local stubs transparently; in their case the interfaces allow them to switch real classes with their mock equivalents, determining which parts of an application are to be tested. The Test Factoring system differs in the way it handles system code. Rather than redirecting through wrappers or extending classes, they directly rewrite the system library to include mock objects. This is not feasible in our system, due to the limitations of visibility between class loaders. Such rewrites are possible only if classes are not renamed, and any referenced libraries are stored in the boot class path.

We present our work in the context of the transparent distribution of Java applications. The closest work in that area is J-Orchestra [9, 10, 12], which also performs transparent distribution, targeted at a different domain. J-Orchestra partitions an application across a fixed, small number of hosts while RuggedJ distributes across large clusters. This difference is visible in our rewriting approach: RuggedJ performs all rewriting at class load time, taking advantage of the particular configuration of a given network, while J-Orchestra is able to use its advance knowledge of the network to generate a customized `jar` file ahead of time for each site.

7. Conclusion

We have described a series of virtualizing transformations that can be applied to the various classes that comprise an application to allow them to conform to a unified object model, as well as the classification process to determine which template should be applied to each class. We can transform nearly every class in an application to be distributable or remotely accessible, permitting fine-grained distribution of objects across a cluster of nodes. We have described the implementation of our prototype system, and presented both an analysis of the classification process when applied to several benchmarks and a discussion of the overheads imposed by the transformations. We have demonstrated the feasibility and generality of our transformations, and shown that their performance overheads are low enough for practical use.

Acknowledgments

This work is supported by the National Science Foundation under grants Nos. CNS-0720505/0720242, CNS-0551658/0509186, and CCF-0540866/0540862, and by Microsoft, Intel, and IBM. Any opinions, findings, and conclusions expressed herein are the authors’ and do not necessarily reflect those of the sponsors.

References

[1] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA ’06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

[2] Michael Factor, Assaf Schuster, and Konstantin Shagin. Instrumentation of standard libraries in object-oriented languages: the Twin Class Hierarchy approach. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 288–300, 2004.

[3] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36–44, 1998. doi: 10.1145/286936.286945.

[4] Phil McGachey, Antony L. Hosking, and J. Eliot B. Moss. Pervasive load-time transformation for transparently distributed Java. In *Proceedings of the International Workshop on Bytecode Semantics, Verification, Analysis, and Transformation*, 2009.

[5] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for Java. In *Proceedings of the International Conference on Automated Software Engineering*, pages 114–123, 2005.

[6] Standard Performance Evaluation Corporation. SPEC JBB2005 Benchmark. http://www.spec.org/jbb2005, .

[7] Standard Performance Evaluation Corporation. SPEC JVM2008 Benchmark. http://www.spec.org/jvm2008/, .

[8] Sun Microsystems, Inc. The JVM Tool Interface. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti.

[9] E. Tilevich and Y. Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Transactions on Software Engineering and Methodology*, 2009. To appear.

[10] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java application partitioning. In Boris Magnusson, editor, *Proceedings of the European Conference on Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 178–204. Springer-Verlag, 2002.

[11] Eli Tilevich and Yannis Smaragdakis. Transparent program transformations in the presence of opaque code. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, Lecture Notes in Computer Science, pages 89–94, 2006.

[12] Eli Tilevich, Yannis Smaragdakis, and Marcus Handte. Appletizing: Running legacy Java code remotely from a Web browser. In *Proceedings of the IEEE International Conference on Software Maintanance*, pages 91–100, 2005.

[13] YourKit, LLC. The yourkit java profiler. URL `http://www.yourkit.com/`.