Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2009) 47–64
www.elsevier.com/locate/entcs

Pervasive Load-Time Transformation for Transparently Distributed Java

Phil McGachey[a] Antony L. Hosking[a] J. Eliot B. Moss[b]

[a] *Department of Computer Sciences, Purdue University, West Lafayette, IN, USA*

[b] *Department of Computer Science, University of Massachusetts Amherst, Amherst, MA, USA*

Abstract

The transformation of large, off-the-shelf Java applications to support complex new functionality essentially requires generation of an entirely new application that retains the execution semantics of the original. We describe such a whole-program modification in the context of RuggedJ, a dynamic transparent Java distribution system.

We discuss the proxy-based object model that allows remote Java objects to be referenced in the same way as those residing on the current virtual machine, the optimizations that allow us to bypass proxies in the case of purely local or remote object, and the mechanisms needed to guarantee that static data remain unique in a distributed system. We then detail some of the more interesting features involved when implementing this object model in rewritten bytecode, including transformations required within method bodies and coordination between bytecode and the run-time system that distributes an application across the network.

Keywords: Java, Bytecode Transformation, Load-Time Rewriting, Transparent Distribution, Object Model

1 Introduction

Automatic code modification for Java applications is a widely-used technique that adds functionality to existing software. Aspect-oriented programming or bytecode rewriting make it trivial for programmers to implement cross-cutting concerns such as logging, error handling, or profiling without modification to original applications. More complex is the comprehensive transformation of an application; generating an entirely new program that retains the execution semantics of the original, while adding substantive new functionality.

In this paper we describe the process of pervasive transformation in our transparently distributed Java system, RuggedJ. We use load-time dynamic bytecode transformation to generate an entirely new class hierarchy that mirrors the structure of an off-the-shelf Java application, adding the necessary functionality to execute the application across a network of Java virtual machines. While we discuss

doi:10.1016/j.entcs.2009.11.014

our program transformation process in the context of RuggedJ, many of our techniques would be equally useful to other large-scale modification applications such as persistence.

We discuss a proxy-based object model that abstracts object implementation, hiding whether they are local or remote to a given virtual machine. This model allows for objects to be distributed and migrated across the RuggedJ network while still preserving the execution semantics and class hierarchy of the original application. Additionally, by referring to transformed classes in rewritten bytecode only using interfaces we allow the elimination of proxies in the common case where an object is known to be always local or remote.

Performing our transformations at the bytecode level affords us several major advantages: We do not require access to the original application source; we can perform our transformation at class-load time, taking advantage of dynamic knowledge of the execution environment; the relative simplicity of bytecode when compared to source allowing our rewrites to be more general; and we can perform incremental changes on-the-fly, referring to generated classes that may or may not be created on demand without having to perform a whole-program compilation.

2 Application Distribution with RuggedJ

The current trend in microprocessor technology is for increases in the number of cores on a processor to replace the once-familiar increases in processor speed. The immediate implication for application developers is that we can no longer rely on the next generation of processors to make our systems run faster; we must instead take advantage of parallelism on individual machines and, increasingly, distribution across clusters of commodity machines.

Unfortunately, the implementation of complex distributed systems demands a great deal of additional effort from programmers and is liable to introduce obscure bugs. Objects must be allocated and tracked across nodes, method calls and field accesses must take into account the location of their targets, objects may need to be migrated from node to node in order to gain acceptable performance, and so forth.

RuggedJ is an automatic transparent distribution system that aims to eliminate these concerns by transforming a Java application to run across a cluster of machines. We achieve this through a combination of a run-time distribution library and a transformation process that creates a new, distribution-aware, application from a set of standard Java class files.

Our implementation of RuggedJ is mostly complete. The bytecode transformation process is in place and tested on realistic applications running on a single node. We have distributed simple applications, but are currently working on the complete distribution of complex systems.

2.1 The RuggedJ Network

A RuggedJ network consists of a set of Java virtual machines (VMs) that distribute and run an off-the-shelf Java application. Each virtual machine (a *node*, in RuggedJ

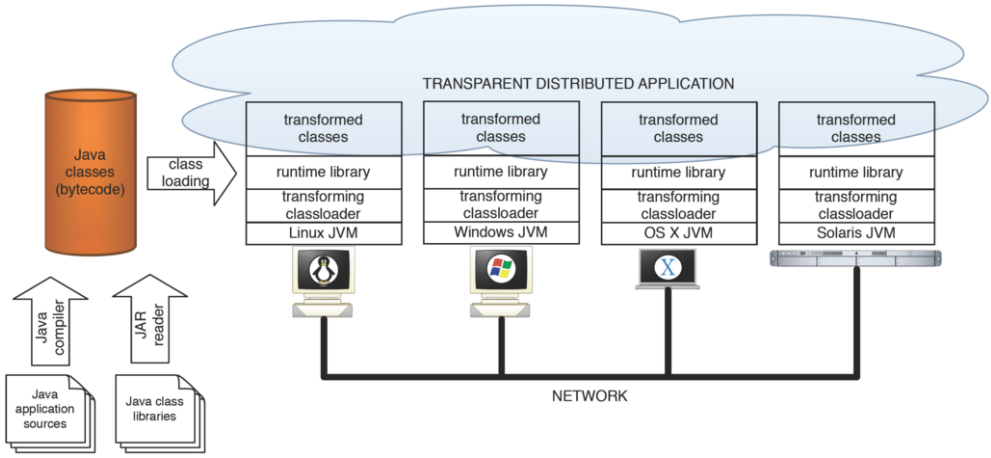

Fig. 1. The RuggedJ System Architecture

terminology) contains an instance of the RuggedJ run-time library that interacts with the run-times on other nodes to coordinate the execution of an application. Figure 1 shows the construction of a RuggedJ network:

We refer to a single physical machine available to RuggedJ as a *host*. This is distinct from a node; a single host can run multiple nodes. RuggedJ is designed to be platform-agnostic, in that a network can consist of heterogeneous hosts. We require only that each host is capable of running a fully-functional Java VM, and that all VMs run the same version of Java, including the standard class libraries.

Each node consists of two parts: the transforming class loader and the run-time library. The presence of a transforming class loader on each node allows a given class to be rewritten differently on different nodes, taking advantage of the capabilities of the host or knowledge of the execution environment. We designate the node upon which the application starts to be the *head node*. As well as functioning as a standard RuggedJ node, the head node acts as a central location for the application, handling I/O requests and other operations that require native access to a particular host. It also acts as an overseer for coordination between nodes: the head node maintains full information on the location of objects and the condition of the network, providing other nodes with a definitive source for this information. While the head node can present a bottleneck, it is necessary not only as a co-ordinator but also as a location for classes that cannot be distributed (see Section 3.4)

2.2 Application Partitioning

The partitioning strategy for a given application is defined by the application developer. While substantial research exists in the literature concerning automatic application partitioning [9, 13, 19], we feel that one is more likely to arrive at an optimal partitioning when developing it using the domain-specific application knowledge available to the programmer. Additionally, many automatic partitioning schemes rely on an advance knowledge of the network configuration under which an application will run. This runs counter to our aim of environmental flexibility,

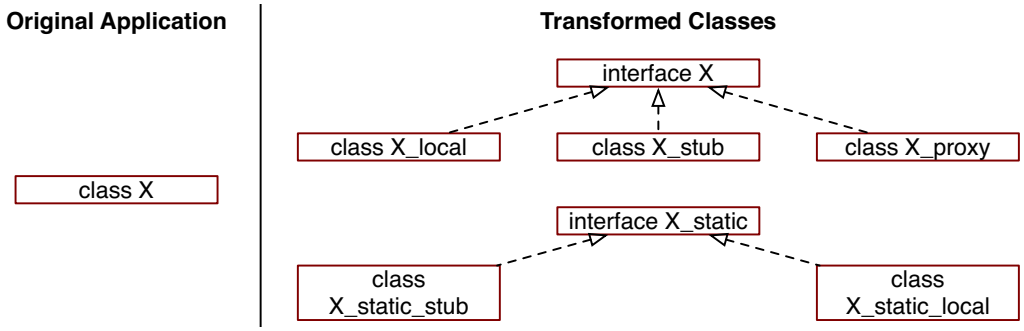

Fig. 2. Classes generated by the RuggedJ class loader

where one can use an application partitioning on a RuggedJ network made up of arbitrary nodes.

To allow the partitioning developer to take full advantage of the RuggedJ network available, we provide a plug-in interface to which one can attach a partitioning strategy, allowing access to the RuggedJ run-time's dynamic internal state (network conditions, node load levels, etc.). RuggedJ consults the strategy for the running application both at class-loading time, guiding the rewrites that the system applies to a class, and at run time, allowing it to base its dynamic decisions upon the current state of the network. Some of the options available to the partitioning designer are discussed in Section 4.4

2.3 Run-Time Support

The RuggedJ run-time library manages the interaction between rewritten bytecode on remote nodes, allowing separate processes to interact and execute a single application. Parts of the library are called by rewritten code, while others coordinate to provide networking services and to generate an accurate picture of the network as a whole.

Library Functionality: The run-time library provides functionality to rewritten bytecode. Many of the operations required to support distribution require complex code sequences. Specifying such operations within a rewritten method would very quickly lead to unreadable bytecode, which is difficult to debug. Instead, we delegate to the run-time library all operations that require more than a very simple bytecode sequence.

Run-time Co-ordination: There are several key tasks performed by the run-time to co-ordinate execution between nodes. These include monitoring the network and host configurations and status, tracking the location of objects for remote method calls and message passing between nodes.

3 The RuggedJ Object Model

We accomplish distribution in RuggedJ by abstraction of object locations. We achieve this through use of proxies for objects. Proxies allow the implementation of an object to vary depending on whether it is local or remote, while presenting a

single interface to external code. For every class in the original application, RuggedJ generates a series of classes and interfaces, as shown in Figure 2.

We split classes into two parts: the fields and methods that make up per-object state (the instance parts), and those that are specific to the class (the static parts). This is necessary in order to ensure that static data exists exactly once in the RuggedJ network; Section 3.3 discusses this issue further.

When the RuggedJ class loader has rewritten a class, it presents only the transformed version for loading into the Java VM. The VM never sees the original class, which removes the possibility of conflicts between modified and unmodified classes. The only exception to this is in the case of unmodifiable classes, which we describe in Section 3.4.

3.1 Instance Classes

Focusing on the instance parts of Figure 2, we see the three classes and one interface that RuggedJ generates from the instance parts of each application class **X**:

Interface X: The interface contains an abstract version of each instance method present in the original application class, as well as get and set methods for each field (see Section 4.2). The name of the interface is significant. By using the name of the original class, the Java type system will recognize an object that implements this interface as the original class. This property simplifies the rewriting of certain bytecode structures, such as `instanceof` checks and exception handling, and removes the need to transform every reference to the original class. The local, stub and proxy classes each implement the interface, and rewritten code refers to a class primarily via its rewritten interface.

Class X_local: The local class contains the fields and the implementation of each instance method from the original class. One can thus think of it as the "actual" object. Any methods invoked upon the object must ultimately execute on an instance of the local class.

Class X_stub: The stub class represents a remote object (i.e., one for which a local version exists on a different node). The stub contains a globally-unique identifier of the remote object, and it implements each method of the interface as a remote method invocation.

Class X_proxy: The proxy class provides a level of indirection between calling code and the local or stub implementation of an object. It contains a single field that holds a reference to either a local or stub object, and implementations of every method in the interface that invoke the relevant method on the referenced object.

Where the original application allocates an object of type `X`, the transformed version creates a pair of objects. One is either an `X_local` or `X_stub`, depending on the node upon which the allocation occurs. The other is an `X_proxy` object that references the local or stub object. By referring to proxies rather than local or stub object in rewritten code, RuggedJ ensures that only a single pointer exists to a local or stub object on a given node. This allows objects to migrate easily from node to node: should an object move from the local node, it is necessary only to

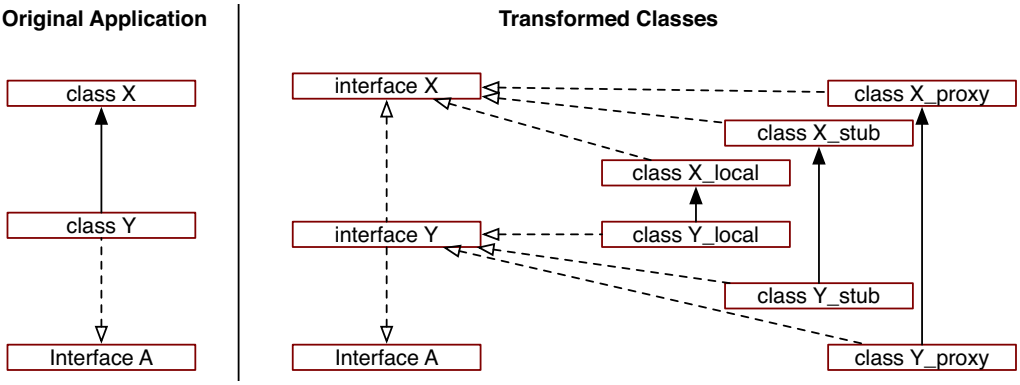

Fig. 3. Inheritance between transformed classes

update the reference in the proxy from the local class to a stub. Migration without proxies would require updating all references in objects or on stacks, which would be prohibitively expensive.

The design of the object model, however, does allow for the direct allocation of local or stub objects, bypassing the proxy. This is desirable for objects that are known never to migrate, such as those objects directly tied to the local virtual machine (such as file handles, class objects, and so forth), or objects known by the author of the partitioning strategy to exist on only one machine (such as temporary objects or local data structures). Allocating proxies for such objects would be unnecessary, adding the overhead of indirection when the referenced object is never going to change. In these cases, RuggedJ instead simply allocates either the local or stub object.

We can use `X_proxy`, `X_local`, and `X_stub` objects interchangeably in this manner because each implements the generated interface `X`. We make all method calls within rewritten code in terms of the interface, and field accesses go through the generated get and set methods. By calling methods through interfaces, we minimize the transformation necessary on calling code, while maximizing flexibility in the types of objects used.

3.2 Inheritance

As well as providing a mechanism by which we can reference different versions of a class uniformly, RuggedJ's generated interfaces maintain the inheritance relationships between original classes. Figure 3 shows the relationship between transformed classes (omitting static parts).

The original application's inheritance relationship between subclass `Y` of class `X` appears as the transformed interface `Y` extending interface `X`. Since rewritten code refers to objects exclusively by interface, this allows one to use any object that implements `Y` when the original code required an instance of `X`. Similarly, `checkcast` or `instanceof` operations operate over interfaces, and produce the same results in transformed code as in the original application.

Each transformed class `Y_local`, `Y_stub` and `Y_proxy` extends the equivalent

part of class `X`. This is not necessary to preserve the inheritance relationships of the original application. Other than when allocating objects, rewritten code never refers to these individual classes. Rather, this subclassing works to simplify the implementation of these classes. Without it, each class would have to contain the fields and implementations for every method of the superclasses of its unmodified version, which would lead to duplication of code and overly-complex classes.

3.3 Static Classes

Turning to the static parts in Figure 2, we see that RuggedJ generates an additional interface and two classes:

Interface X_static: This interface contains each static method and get/set methods for each static field in the original class. It functions similarly to the instance interface `X`. Both `X_static_local` and `X_static_stub` classes implement `X_static`, allowing rewritten code to use them interchangeably.

Class X_static_local: This class contains the static fields and implementations of each static method from the original class. RuggedJ modifies both fields and methods to be *instance* members of `X_static_local` rather than static members. This allows class `X_static_local` to fulfill the requirements of interface `X_static`, and decouples the implementation of static members from the implementation of the VM.

Class X_static_stub: The static stub acts similarly to the instance stub class. It contains implementations of each method in interface `X_static` that perform remote invocations on the appropriate `X_static_local` object.

Transforming static methods of original class `X` into instance members of class `X_static_local` serves two purposes. First, it allows the static part of an object to be treated as any other object in the RuggedJ network. This allows us to take advantage of any object migration or caching performed by the system for static data as well as individual objects. More importantly, however, is the fact that transforming static data to representation as an object allows us to ensure that only one copy of the data exists in the network. Were static fields left unmodified, each VM that loads class `X` would have its own copy of each field, leading to inconsistent data.

We use the concept of *static singletons* to maintain unique static data. The RuggedJ run-time library creates static objects on demand, and coordinates between nodes to guarantee that the network contains at most one instance of `X_static_local`. Once some node has allocated the singleton, all other nodes will create `X_static_stub` objects as required. By managing static singletons through the run-time, we eliminate the need for a `X_static_proxy` class. Since rewritten code does not store references to the static singleton, we ensure that each node has a single reference to a given static singleton. Should the need arise to migrate a static singleton, we must update only this one reference.

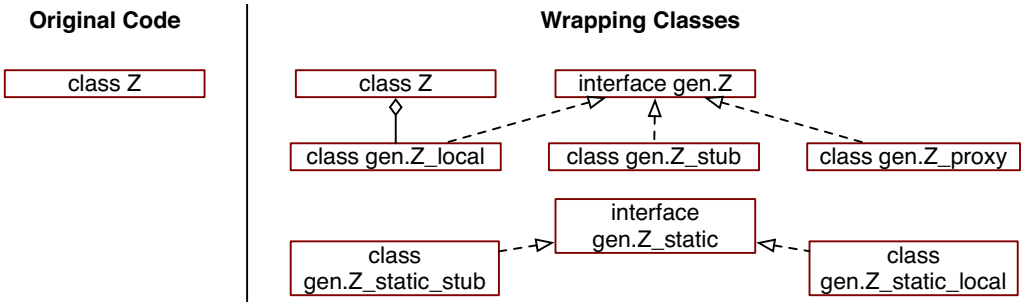

Fig. 4. Wrapping unmodifiable classes

3.4 Unmodifiable Classes

While RuggedJ can rewrite the majority of application classes as described in Section 3.1, there are a subset of application and Java standard library classes that it cannot. This limitation arises from the presence of native code. We cannot rewrite a method implemented using Java's native interface, and such a method will not be aware of the presence of transformed classes. It may attempt to access fields or methods that we have modified or that are not available. We call these classes *unmodifiable*, and do not rewrite them. Tilevich and Smaragdakis define such classes to be those accessible by native code: classes that contain native methods, those passed to or returned from unmodifiable classes, the types of fields in unmodifiable classes, and superclasses of unmodifiable classes [20]. While it is theoretically possible for native code to access other (indeed, any) classes in the system, they found this heuristic to be sufficient for the realistic applications they examined.

Since we do not transform unmodifiable classes, we cannot distribute them. In practice, we find that the majority of unmodifiable classes exist within the Java standard libraries, and are often closely tied to the underlying VM. This does not prove to be a great obstacle to the distribution of an application, since such classes would not move in any case. However, it is necessary that remote nodes be able to reference instances of unmodifiable classes. To this end, we generate *wrappers* for unmodifiable classes.

Figure 4 shows the classes we generate for an unmodifiable class. It is important to note that in this case, class `Z` is the original, unmodified class; we generate the wrapping classes as part of a reserved package to avoid naming conflicts with the original. Class `gen.Z_local` acts as a wrapper around the original class `Z`. It contains a reference to the instance of the unmodifiable class, with implementations of instance methods, each of which calls the appropriate method on the wrapped object. We generate `gen.Z_stub` and `gen.Z_proxy` identically to the stub and proxy classes described in Section 3.1.

Class `gen.Z_static_local` acts as a static singleton for the wrapped class, with one important difference. Since an unmodifiable class may directly access static members, we cannot rewrite such static data to form instance methods of `gen.Z_static_local`. Thus, the methods of the static local class instead simply delegate to the original class. To ensure uniqueness in static data, a given unmodifiable class can thus access static data only on a single node. While this limits the

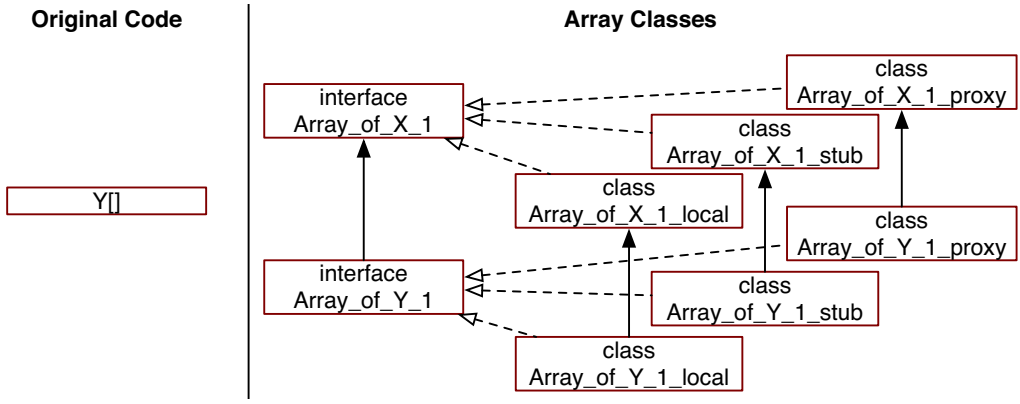

Fig. 5. Generated array classes

potential for distribution, in practice most such classes tend to be allocated only on the head node, with transformable application classes running on all other nodes.

Rewritten code interacts with unmodifiable code using the interfaces and wrapping classes in the same way as regular classes. This allows the same transparency with regard to object location for wrapped classes that exists for transformed classes. However, since the unmodifiable code itself is unaware of transformed classes, we must unwrap objects when passing them as arguments to unmodifiable code, and re-wrap returned objects. The unwrapping process is simple: the methods with `Z_local` unwrap any transformed class arguments (recall that, by definition, all classes passed to unmodifiable code are themselves unmodifiable and so can be unwrapped). Wrapping return values is slightly more difficult, since we must ensure that a given object has only one wrapper—an object returned multiple times from unmodifiable code must always be wrapped by the same object. We control this in the RuggedJ run-time library, which tracks generated wrappers and creates a new wrapper only if the object has not been wrapped before.

3.5 Arrays

When distributing an application, we must transform not only objects but also arrays. To this end, we generate classes for array types, as shown in Figure 5.

We generate a set of interfaces and classes for every pair of array content and dimensionality used in the application. The interface contains get and set methods for the array content, as well as methods to perform standard operations such as getting the length or hash value for the array. Class `Array_of_Y_1_local` is a wrapper for a one-dimensional array of `Y` objects (the contents of which are themselves instances of local, stub, or proxy classes that implement interface `Y`). Array classes do not need static singletons, since arrays maintain no static state.

We generate the classes for multi-dimensional arrays in the same way as for single dimension arrays (we represent a two-dimensional array of `Y` by interface `Array_of_Y_2` and so forth), with the local class containing a wrapped array. The wrapped array is always one-dimensional, so `Array_of_Y_2_local` contains an array of `Array_of_Y_1` objects. As well as simplifying the implementation of arrays, this

design allows us to spread large multi-dimensional arrays across multiple nodes.

3.6 Hand-Coded Classes

A final, small, subset of classes within RuggedJ are hand-written and loaded unmodified into the Java VM. These are classes that require specific, customized implementations within the RuggedJ network. For example, `java.lang.System` contains several methods for which we define special semantics: we must redirect all references to `System.out` to the head node, rather than to the local machine. Since performing such one-off transformations would be laborious and would complicate the transformation framework, we prefer instead simply to load a hand-coded version of these classes.

4 Implementation

Beyond generating new classes, implementation of the RuggedJ object model requires widespread modification to application bytecode. In this section we describe some of the more interesting features of the rewriting process.

4.1 Bytecode Rewriting Tools

When implementing RuggedJ, the first decision we needed to make was the level at which to rewrite. High-level tools such as AspectJ [10] and MetaAspectJ [8] would allow us to specify RuggedJ's transformations in Java source code. While this is adequate to add code to a method, more complex transformations would require an additional tool. A more flexible approach is that of Javassist [3, 4], which allows one to specify transformed code in Java syntax, which it compiles with a custom compiler. This offers a lower-level interface to rewriting. However, we found that its on-demand compilation approach made whole-program modification difficult. Ultimately, we found that ASM [1] supports a good balance of direct access to method bytecode while hiding awkward details such as management of constant pools and the selection of instructions with hard-coded local variable slots. These two abstractions vastly simplified the design of transformations and generated bytecode, making ASM more useful to us than the similarly-featured BCEL [5].

4.2 Transforming Method Bodies

Of the classes we generate for a given application class, only the local and static local versions contain copied bytecode. We generate all other classes from scratch. Thus, we apply the following transformations only to the bodies of local and static local classes.

Instance Method Invocation: We must refer to all transformed objects in RuggedJ by interface rather than class type, allowing us to vary the implementation of a class among proxy, local, or stub transparently to the calling code. This clearly requires modification to method call sites, transforming `invokevirtual` bytecodes to `invokeinterface`.

We need a more complicated rewrite in the case of `invokespecial` bytecodes, used to call private methods, constructors, or superclass methods. We can call private methods in the same way as regular methods (for the sake of simplicity, we modify all methods to be public; the original Java compilation enforced the access controls). However, we cannot call constructors or superclass methods through an interface. We must invoke a constructor upon the appropriate class; we describe this process in Section 4.4. Superclass invocations must specify the superclass type upon which to invoke a method (in case a subclass has overridden the method). This does not present a problem since we knot that the code we are modifying is within a local class, the superclass of which we also know.

Instance Field Accesses: We must rewrite accesses to instance fields, since direct access to a field assumes that an object is local. To this end, we replace every instance field access by a call to the appropriate get or set method in the interface.

This policy obviously adds an unnecessary level of indirection when the accessed field belongs to the accessing object. A more subtle problem exists, however, that necessitates special handling of such accesses. Under the Java VM specification, the only operation that may occur in a constructor before the invocation of a superclass constructor is the initialization of a field in the local object. Rewriting such a field invocation to a method call would cause a verification error, since a method call cannot precede the superclass constructor call. We can detect cases where a field access occurs on the accessing object using a simple flow analysis, as we describe in Section 4.5.

Static Method Bodies: As we discussed in Section 3.3, we transform static fields and methods within the static_local class to be members of the static singleton object. While transforming fields is straightforward, we must rewrite static method bodies to function as instance methods. The first local variable slot in an instance method is reserved for the `this` pointer, referring to the object upon which the method is invoked. Static methods are not invoked on any object, and so do not have a `this` pointer. Thus, when converting a static method to an instance method we must to update all local variable references to allow for the new reference.

This transformation can cause major changes to the bytecode sequence of a method. Not only does it change the parameters to local variable bytecodes, but the bytecodes themselves may change. For example, the `aload_3` bytecode operates as an `aload` with a parameter of 3. Incrementing the local variable slot upon which this bytecode operates would require replacing the `aload_3` bytecode (a one-byte instruction) by an `aload` with an argument of 4 (a two-byte instruction). This will affect the offsets of future bytecodes, and will require updates to jump instructions, exception handling blocks, and so on. Fortunately, a bytecode rewriting toolkit such as ASM abstracts away most of these details.

4.3 Accessing Static Singletons

As in the case of instance field accesses and method invocations, we must rewrite static accesses. However, the presence of static singletons makes the process somewhat more complex. First, the RuggedJ run-time library must locate the appropri-

ate static singleton by looking it up in a hash table of static objects. If the required singleton is unavailable, the run-time library must first determine whether a singleton exists on another node and, failing that, create one. This involves coordination with the other nodes in the network to find an existing singleton, or synchronization with the head node to avoid two nodes simultaneously creating singletons. When the code has found the singleton, the modified bytecode can then invoke the necessary method on it. In the case of field accesses, this consists of a call to the appropriate get or set method.

This is clearly a costly operation, particularly in the case where the static singleton is a stub, and a method invocation requires access to a remote object. As such, we minimize access to static singletons as far as possible. We observe that the static singleton exists only to ensure that there is only one copy of static data. Therefore, we need to call the singleton only when we may access that state: calls to static methods that do not read or write the singleton's fields do not go through the singleton. Rather, they call a local version of the static method. Indeed, for classes with no static state, it is not necessary to create a static singleton at all.

Initialization of static data is performed by the run-time system when a static singleton is created. Any `static{}` code block is transformed into an instance method, allowing it to be called at the appropriate time during singleton creation. Static final fields (constants) are treated in the same way as all other static fields; constants are initialized by the static initializer, and can have different values on different nodes (consider a constant initialized to a host's IP address). Forcing static final fields to go through a static singleton is conservative, and can be optimized in many cases.

4.4 Allocation

The object allocation process involves interaction between the rewritten bytecode in a method and the partitioning strategy defined by the application author. It is the primary means by which one distributes an application. By strategically allocating objects on remote nodes and remotely invoking methods, one can perform large computations across a collection of nodes.

We define an *allocation site* as an instance of a `new` bytecode. When rewriting an allocation site, the RuggedJ rewriting class loader first queries the partitioning plug-in with static site information to request a load-time allocation strategy. The allocation site information includes the class and method in which the allocation site occurs and the type it allocates. Based on this, the partitioning can return one of three options:

Allocate Locally: If the policy knows that the code uses the particular type of object principally on the local node, we can streamline the allocation process to create the local version of the class. This is a fairly common case: some classes rely on local resources, many objects are temporary and of purely local interest, and domain-specific knowledge may determine that an object will rarely be used by another node. The partitioning plug-in may also determine whether to allocate a proxy to allow for later migration, or simply to allocate the local version directly,

allowing it to be remotely referenced but not migrated.

Allocate Remotely: On the other hand, a policy may sometimes know that we should always allocate an object on a different node. This may be the case if the partitioning strategy dictates to spread objects of a certain across the network for load balancing purposes, or that a particular class would benefit from a resource that is not available on the local host. This option allocates both a proxy and a stub object, and determines at run-time on which node to allocate the object.

Allocate Dynamically: Finally, there are cases where we will not know the best allocation node for an object until run time. This may be the case if we should evenly distribute the objects of a class over the network: the location of the object will depend on the run-time distribution pattern. This option defers the decision of whether to allocate a local or stub class until run time.

Each of these options causes the rewriting class loader to replace the allocation site with a different bytecode sequence. In the case of a local allocation, the bytecode sequence simply creates a new local object, with or without a proxy. The remote allocation sequence involves a call to the run-time library to determine the node upon which to create the object, then a remote creation request and creation of a stub object and proxy. Finally, the dynamic allocation option generates both sets of bytecode, with a call to the run-time library to determine which to execute.

A final complication when rewriting an allocation site is that of calling the appropriate constructor. The constructor call for an object can be an arbitrary distance from the `new` bytecode that creates the object to pass to the constructor, since there may be an arbitrary number of operations to compute the arguments to the constructor. There may even be other constructor calls between the two bytecodes, since the arguments to the constructor may require creation of new objects. We take advantage of the fact that every `new` bytecode has exactly one constructor call, and so we can match a `new` bytecode with its constructor call using a simple stack-based scanning technique. We scan forward through the bytecode stream pushing any `new` bytecodes, and popping them when we encounter constructor calls. The final constructor we encounter therefore belongs to the original `new` bytecode.

4.5 Flow Analysis

The vast majority of bytecode modifications in RuggedJ are context-independent; their implementations do not require knowledge of the method as a whole. Aside from the method scanning required to locate constructors mentioned above, there exist two cases for which we need to analyze the method body.

The first concerns operations on arrays. As discussed in Section 3.5, we replace all arrays in a RuggedJ network with wrapping objects. This presents problems during the rewriting phase since, unlike most bytecodes that operate over references, array operations (`aaload`, `aastore`, `arraylength`, etc) do not encode type information. One can determine the type of the array reference and return value only by modeling the run-time stack. Since we rewrite these bytecodes to standard `invokeinterface` method calls, we need to know both the type and dimensionality of the array upon which to invoke the method. We find this information using a a

standard bytecode flow analysis of types of objects that tags each array bytecode with the type of array currently on top of the stack.

The second flow analysis we require is to track the `this` pointer in instance methods. As mentioned in Section 4.2, the RuggedJ rewriting class loader must differentiate between field accesses on the current object and those on others. Since we know that the `this` pointer exists in local array slot 0, we can track any references that start life with an `aload_0` bytecode, determining them to be references to the current object.

This analysis is, by its nature, conservative. It can produce a false negative when, for example, code passes a reference to a method that then returns something of the same type. The return value could be the original reference or a different object. This conservatism is not a problem since we employ the analysis mostly for optimization, so missing a reference does not violate correctness. The only occasion where we rely on this analysis is where a field initialization occurs prior to the super-constructor call in a constructor. However, since the only field initializations that may occur before that call are to the local object, the analysis will always be accurate in this case.

4.6 Uncooperative Code

As with most large-scale automatic application transformation systems, RuggedJ cannot guarantee correctness in all cases. There are certain corner cases where an adversarial programmer can foil the rewriting system into producing incorrect results. However, we are confident that such cases are rare under normal circumstances.

The most apparent area in which our rewrites might lead to errors is reflection. An application developer generally has more knowledge of the run-time properties of objects in an application, and could use Java's reflection system to perform operations on a class that may not be possible in the rewritten system. With that said, we do take measures to avoid this by intercepting reflective calls and updating arguments or types to fit within the RuggedJ system, allowing most common usages of reflection to operate within our system.

We also do not support applications that define their own custom class loaders. Since RuggedJ uses a rewriting class loader, we cannot integrate the operations that may be performed by an application's own class loading system.

Finally, we are aware of several ways in which native code could produce incorrect results within RuggedJ. The heuristics discussed in Section 3.4 allow our system to accommodate most native code, but the Java Native Interface allows native code virtually limitless access to the VM. By allocating or invoking methods on arbitrary objects a native method can perform operations that are incompatible with RuggedJ's transformations. This problem will most likely arise in a non-adversarial application though use of static singletons. The Java Native Interface `CallStatic<type>Method` methods allow native code to invoke static methods of arbitrary classes. Reflectively invoking a static method of a class that requires a static singleton will result in the call failing in RuggedJ. However, allowing for

arbitrary static method calls would mean that no class could have a static singleton and so could only be accessed from a single node, making distribution impossible.

5 Related Work

The system that most closely resembles RuggedJ is J-Orchestra [18]. Indeed, J-Orchestra influenced many of RuggedJ's original design decisions. However, J-Orchestra's fundamental goal is different from RuggedJ's. J-Orchestra aims for "resource-driven distribution," where one shares an application between a small set of machines with specific capabilities. For example, a transformed system may perform calculations on a back-end server, while displaying its user interface on a PDA. This differs from RuggedJ's goal of distributing an application across a cluster of machines, taking advantage of additional hardware to exploit parallelism. The design of each system reflects these differing objectives.

The major difference between the two systems is that RuggedJ performs dynamically many functions that J-Orchestra performs statically. J-Orchestra determines a partitioning ahead of time for a given network configuration. Guided by a whole-program analysis, a user determines which classes should have their instances allocated on each network location. This approach works well for J-Orchestra's usage, since it targets small clusters with clear roles for each machine. However, RuggedJ performs this partitioning at run time using an application-specific partitioning plug-in to decide dynamically upon the location of remote objects. Similarly, one can see the static/dynamic difference in the way in which J-Orchestra rewrites application code. It transforms classes ahead of time, generating proxies and remote representations as Java source that one then compiles, producing a `jar` file for each network location. This is in contrast to our approach of rewriting at class load-time, which gives us the ability to generate bytecode tuned to the RuggedJ network upon which the application is running, and removes the system's dependence on an external compiler.

Another consequence of J-Orchestra's ahead-of-time partitioning strategy is that it makes all partitioning decisions on a per-class basis. In contrast, RuggedJ's dynamic partitioning system allows per-instance decisions, allowing us to allocate instances of a given class on arbitrary nodes within the network. Not only does this let us take advantage of current network conditions that cannot be predicted ahead of time, but it also allows us to perform load-balancing by distributing key objects of a given class across the network.

Finally, there are differences in the object model implemented by each system that we feel allow RuggedJ more flexibility when executing large applications. In J-Orchestra, the fundamental class for objects that code may reference remotely is the proxy, while in RuggedJ it is the interface. Rewritten bytecode in J-Orchestra refers to proxies rather than interfaces, removing the ability to elide proxies for objects that are known to be either local or remote. Additionally, J-Orchestra's approach to arrays differs in that it considers arrays of a given type but of different dimensionality to be related, while RuggedJ considers an array type to consist of

both a base type and dimension, allowing for a more flexible partitioning scheme.

There exist several other projects that seek to simplify the distribution of Java. Space limitations prevent us from discussing these systems in detail, but none follow the same approach as RuggedJ.

Terracotta [17] is an open-source JVM-level clustering framework that uses bytecode rewriting techniques to generate a distributed Java application without the requirement to code to a specific API. The Terracotta approach is superficially similar to that taken by RuggedJ, but there are several fundamental differences: Terracotta requires that the application developer label "root" references with altered semantics through which one can reach shared objects, while RuggedJ considers all objects as potentially reachable from remote nodes. Additionally, Terracotta is heavily based upon a central server node, which manages the canonical versions of all shared objects. We maintain canonical versions of objects throughout the cluster.

Addistant [16] uses bytecode transformation to distribute legacy code, but does not aim to distribute large parts of the application. AIDE [14] uses a modified JVM to offload execution from portable devices to servers, whereas RuggedJ runs on unmodified VMs. JavaParty[7, 15], Javanaise [6], Do! [11, 12] and Java// [2] each provide language-level features to Java that simplify distributed programming, while RuggedJ performs its transformation at the bytecode level without modification to the original source.

6 Conclusion

Whole-program transformation is a powerful technique that allows one to add substantive new functionality to an existing off-the-shelf application. In this paper we have presented the object model implemented by a transformed application running under the RuggedJ transparent distribution system. We have outlined the classes and interfaces required for a flexible, dynamic distributed system and described how such an object model maintains the semantics of the original application. We then discussed the process of transforming an application to implement this object model, including the classes generated, modification to method bodies, and the dynamic distribution of an application though object allocation.

We believe that the techniques described in this paper offer insight into some of the issues involved in large-scale transformation of Java applications, and may serve to guide future implementations not only of distributed Java but of any system that uses indirection to achieve object transparency.

Acknowledgement

This work is supported by the National Science Foundation under grants 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] Bruneton, E., R. Lenglet and T. Coupaye, *ASM: a code manipulation tool to implement adaptable systems*, in: *Adaptable and Extensible Component Systems*, 2002.

[2] Caromel, D., W. Klauser and J. Vayssière, *Towards seamless computing and metacomputing in Java*, Concurrency—Practice and Experience **10** (1998), pp. 1043–1061.

[3] Chiba, S., *Load-time structural reflection in Java*, Lecture Notes in Computer Science **1850** (2000), p. 313.

[4] Chiba, S. and M. Nishizawa, *An easy-to-use toolkit for efficient Java bytecode translators*, in: *Proceedings of the International Conference on Generative Programming and Component Engineering*, Lecture Notes in Computer Science, 2003, pp. 364–376.

[5] Dahm, M., *Byte code engineering with the BCEL API*, Technical Report B-17-98, Freie Universität Berlin (2001).

[6] Hagimont, D. and D. Louvegnies, *Javanaise: distributed shared objects for Internet cooperative applications*, in: *Middleware'98*, The Lake District, England, 1998.

[7] Haumacher, B., J. Reuter and M. Philippsen, *JavaParty: A distributed companion to Java*, http://www.ipd.uka.de/JavaParty/.

[8] Huang, S. S. and Y. Smaragdakis, *Easy language extension with Meta-AspectJ*, in: *ICSE '06: Proceedings of the 28th international conference on Software engineering* (2006), pp. 865–868.

[9] Hunt, G. C. and M. L. Scott, *The Coign automatic distributed partitioning system*, in: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 1999, pp. 187–200.

[10] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of AspectJ*, Lecture Notes in Computer Science **2072** (2001), pp. 327–355.

[11] Launay, P. and J.-L. Pazat, *A framework for parallel programming in Java*, in: *HPCN Europe*, 1998, pp. 628–637.

[12] Launay, P. and J.-L. Pazat, *Generation of distributed parallel Java programs*, Technical Report PI-1171, Institut de Recherche en Informatique et Systemes Aleatoires (1998).

[13] Liogkas, N., B. MacIntyre, E. D. Mynatt, Y. Smaragdakis, E. Tilevich and S. Voida, *Automatic partitioning: A promising approach to prototyping ubiquitous computing applications* (2004).

[14] Messer, A., I. Greeberg, P. Bernadat and D. Milojicic, *Towards a distributed platform for resource-constrained devices* (2002).

[15] Philippsen, M. and M. Zenger, *JavaParty — transparent remote objects in Java*, Concurrency—Practice and Experience **9** (1997), pp. 1225–1242.

[16] Tatsubori, M., T. Sasaki, S. Chiba and K. Itano, *A bytecode translator for distributed execution of "legacy" Java software*, in: J. L. Knudsen, editor, *Proceedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science **2072** (2001), pp. 236–255.

[17] Terracotta Inc., *Terracotta*, http://terracotta.org.

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

[19] Tilevich, E. and Y. Smaragdakis, *J-Orchestra: Automatic Java application partitioning*, in: B. Magnusson, editor, *Proceedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science **2374** (2002), pp. 178–204.

[20] Tilevich, E. and Y. 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, 2006.