Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Class Transformations for Transparent Distribution of Java Applications

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

a. VMware, Cambridge, Massachusetts, USA
b. Purdue University, West Lafayette, Indiana, USA
c. University of Massachusetts at Amherst, USA

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 78% 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.

Keywords distribution, partitioning, program transformation, 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

Phil McGachey, Antony L. Hosking, J. Eliot B. Moss. *Class Transformations for Transparent Distribution of Java Applications*. In Journal of Object Technology, vol. 10, 2011, pages 9:1–35.
doi:10.5381/jot.2011.10.1.a9

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 uniform 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 (78% 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 so as to accomodate any standard Java virtual machine (VM), and we can readily support a heterogeneous collection of host Java virtual machines running a single application or manipulating a single persistent store, 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.

2 Transparent distribution in RuggedJ

We apply the pervasive virtualization transformations described in this paper to RuggedJ, our prototype transparent distribution framework for Java [MHM09b, MHM09a, McG10]. 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. Our focus with RuggedJ is to permit scaling of distributed applications in clusters. In contrast, prior systems such as J-Orchestra [TS06] and Addistant have emphasized *partitioning* of applications, mainly to exploit heterogeneity (such as a graphical user interface running on a client while back-end computations execute on a server), though RuggedJ also supports partitioning.

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 RuggedJ run-time distribution library; the class loader supplies classes rewritten on-demand, while the run-time system interacts with other nodes to co-ordinate application distribution dynamically (e.g., encoding how instances of each class are to be striped across the nodes of a cluster

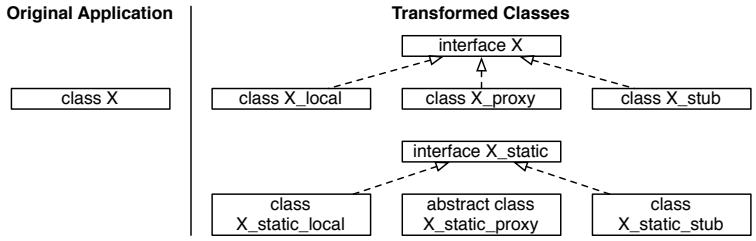


Figure 1 – The RuggedJ object model (UML class diagram)

to ensure load balancing), 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.

2.1 RuggedJ run-time system

Our focus in this paper is on the transformations that allow any application to be distributed according to the design choices of the distribution developer, so we only briefly describe the RuggedJ run-time system here. Naturally, the efficiency and scalability of a distribution will vary widely depending on the policy choices of the developer. These policy choices are enacted by the run-time system, which drives the dynamic class transformations that distribute the application and manages the objects manipulated by the application (whether to instantiate locally or remotely, to migrate or replicate, etc.). The run-time system also handles communication and provides utility code invoked by the transformed classes. Both transformation and object management are dictated from a policy specification provided by the developer. The policy specification allows the developer to express dynamically for any allocation site where each instance is to be allocated, to control which instances can migrate, and which operations (e.g., passing the instance as a method argument or returning it) will cause it to migrate, and to where.

2.2 RuggedJ class transformations

Key to distribution of applications within RuggedJ is the bytecode transformation library, which transforms the classes that make up an application to implement the RuggedJ object model [McG10], as shown in Figure 1. Each class from the original application spawns creation of three new classes and one interface to represent the class's instance *protocol*: the original method names and signatures, and additional accessor methods for every instance field in the class. The static methods and fields of the class similarly produce three new classes and an interface to represent the class's static protocol. Object references within RuggedJ are typed exclusively by interface;

abstracting out the protocol allows the concrete implementation of a class to vary without altering client code that refers to instances of the class. We rewrite method bodies within the class to refer to transformed objects, including redirecting method and field accesses through interfaces, modifying the types of objects to account for the object model, and so on.

The three classes `X_local`, `X_stub` and `X_proxy` provide these concrete implementations. A *local* object contains the instance fields and method implementations of the original class; it can be thought of as the canonical representation of the object. Local objects have a one-to-one relationship with objects in the original application, so only one local instance exists in the RuggedJ system for every instance in the original application. The second class, the *stub*, implements the interface by providing remote method calls to the local object. Stubs have a many-to-one relationship to local objects. Each node in the network (excluding the node that contains the local object) may have up to one stub per object in the original application. This way, any node can refer to any remote object in the system. Finally, the *proxy* object allows objects to migrate. Should an object be migratable, a proxy object will be allocated as well as the appropriate local or stub. The proxy holds a single reference to the local or stub object, and all references to that object pass through the proxy. This way if an object should migrate it is necessary only to update the reference within the proxy to refer to the new implementation. Since the majority of objects within an application never migrate, and we allocate proxies only for those that *may* migrate at some point, the majority of accesses do not incur this indirection.

Additionally, we extract the static parts of the original application from their rewritten counterparts. Static data is required to be unique within the system; individual nodes must not maintain their own, possibly inconsistent, versions of static state. To this end we create a *static singleton* object for each class that contains static data. These singletons are managed by the run-time library, and are guaranteed to be globally unique.

The second aspect of class rewriting involves copying and transforming the contents of the original class to the new local class. All object references must be re-typed to refer to rewritten interfaces rather than to the original classes. Additionally, field accesses are transformed to call get and set methods on the interface, rather than directly reading and writing fields. Finally, the method bodies are modified to call out to the run-time library to perform any additional functions required for distribution. These transformations are discussed in depth in Section 3.5.

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 eliminates the possibility of conflicts between modified and unmodified classes.

3 Class transformation

Class transformation is key to our distribution system. Injecting distribution logic into regular Java code allows classes to interoperate with remote objects and with the RuggedJ run-time distribution library without modifying the underlying Java virtual machine. We perform extensive transformations on each of the classes that make up the original application: we generate an interface that encapsulates the protocol of the class and three implementations of this interface to represent local, remote and migratable objects. Additionally, for classes with static data we create a static singleton that represents this content, generating a further interface and three classes.

Finally, we rewrite the contents of the original classes to be aware of these new classes and to work within a distributed environment.

We define two additional goals in the transformation process. First, we aim to keep the rewritten bytecode as simple as possible. This stems from the practical difficulties inherent to debugging bytecode; the simpler the rewritten bytecode the more straightforward the debugging process. Additionally, overly-verbose bytecode transformation sequences are more likely to lead to complex interactions where generated bytecode sequences are accidentally modified by subsequent transformations. The second goal is to optimize transformed code for local execution. This is a result of two constraints: the vast majority of object accesses in the distributed system should be to local objects, and the overhead of remote invocations is such that optimizing bytecode will do little to affect the overall performance penalty in these cases.

We perform class transformation at the bytecode level, using a custom Java class loader. Bytecode transformation offers several advantages over source-level modification. Modified Java source code would have to be compiled, which would require that the whole program be rewritten ahead of time. Instead, we transform our modified classes incrementally on-demand, without consideration for inter-class dependencies. We take advantage of incremental transformation to optimize classes for their location in the network. Additionally, bytecode is a significantly less complex representation of an application, since Java constructs and variables are collapsed to stack and local variable operations. This makes the transformation process simpler, as there are fewer cases to handle.

3.1 Bytecode rewriting

There exist several strategies to rewrite bytecode. Aspect oriented programming (AOP) is a design methodology that aims to separate cross-cutting concerns from the main logic of an application [KLM+97, KHH+01a, KHH+01b]. An *aspect* is a class that collects reusable or related code in a modular fashion. The aspect can be used to transform existing applications by *weaving* the elements of the aspect into the original application code at specific, well-defined points (known as *pointcuts*), augmenting or replacing the existing code. This way, aspects can be used to implement features such as logging or error handling separately from the main application. AOP suffers from a lack of low-level control; aspects are specified in terms of the classes that they modify, and allow *advice* to insert or modify code that corresponds to specific pointcuts. This matching process makes it difficult to design general aspects that perform specialized context-specific rewrites on arbitrary classes. MetaAspectJ [HS06] aims to remedy this issue by providing an aspect-generating framework that can create specific aspects programatically. However, even with this additional tool, AOP is capable only of modifying existing classes; it cannot be used to generate the new classes required by a system such as RuggedJ.

A number of other tools permit lower-level transformation of Java applications. Javassist [Chi00, CN03] allows specification of code transformations in Java syntax, which is then compiled with a custom compiler, as well as an API for manipulating bytecode directly, but we found Javassist's on-demand compilation approach difficult to apply for our whole-program transformations. Jinline [TSDNP02] is a related project that allows load-time rewriting of bytecode. It provides a version of AOP at the bytecode level, inlining a specified method body at a given bytecode location. JMangler [KCA01, JMa] intercepts and rewrites bytecode at load-time. It is able to work with user-level class-loaders by providing a modified version of the `ClassLoader`

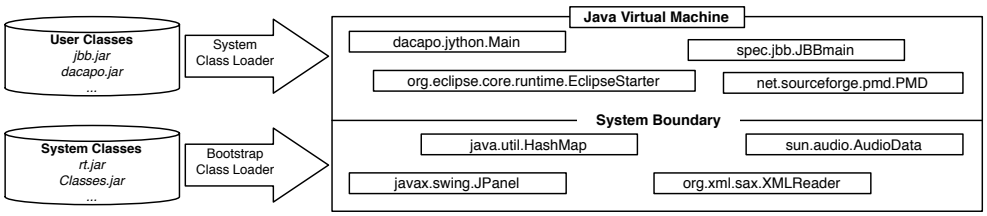


Figure 2 – User and system classes

class. JMangler is currently limited to Java 1.4, making it unsuitable for our needs. Barat [BS98] loads either bytecode or Java source and builds a complete AST. It performs name and type analysis on the code, making the results available for use in other rewriting systems. While the analyses provided by Barat would have been useful in developing RuggedJ, the system is currently limited to analyzing Java 1.1 class files.

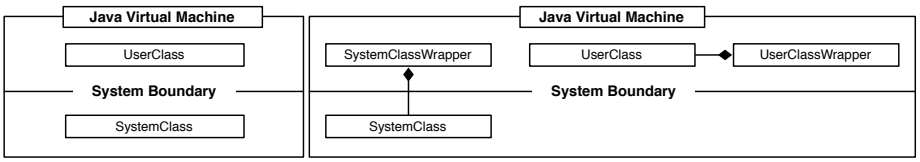
Ultimately, we determined that ASM [ASM, BLC] supports a good balance of direct access to method bytecode while hiding awkward details such as management of constant pools and the selection of instructions with hard-coded local variable slots. These two abstractions vastly simplified the design of transformations and generated bytecode, making ASM more useful to us than the similarly-featured BCEL [Dah98]. Additionally, ASM supports the class file extensions specified in Java 6, allowing us to make use of the latest language features. As a result, we performed the vast majority of our transformations using ASM.

Classes in RuggedJ are transformed on demand with ASM, using a rewriting class loader on each node. This way we can transform classes differently on different nodes (if we know in advance that a class will only ever be allocated upon a single node we can rewrite all accesses from that node as purely local, and all accesses from any other node as purely remote). In addition to the rewriting class loader, we also use a Java Virtual Machine Tool Interface (JVMTI) agent implemented in C to perform limited modifications to Java system code (see Section 3.6.2).

3.2 System and user classes

Figure 2 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 are those in the Java standard libraries, and so are loaded by the virtual machine's *bootstrap* class loader [LB98]. 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. We discuss Java's class loading mechanism, and its implications for our system, in Section 3.6.1.

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.



(a) Before transformation (b) After transformation

Figure 3 – An example of transforming classes using wrapping (UML class diagram)

3.3 Transformation

Figure 3 shows the implementation of *wrapping*; our fallback approach to handling *any* system class within a transformed application when there is no better alternative. In 3a we see one system and one user class before applying any transformations. 3b 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 `SystemClassWrapper` and `UserClass-Wrapper` as *new* types. They are generated at load-time by our rewriting class loader, and thus can implement our object model. In contrast, `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 (whether local or remote), 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.

As an example, consider the old Java system class `FileInputStream` whose instances each have a tight binding to a local file. Such instances cannot migrate to a remote host since their file is local. Yet it is still possible for remote references to instances of `FileInputStream` to be connected to a local instance of `BufferedInputStream`, at the cost of remote calls to the remote `FileInputStream`. We can wrap the local `FileInputStream` instances with a new class so that they implement the RuggedJ object model, allowing manipulation through the wrapper from both local and remote references.

3.4 The RuggedJ object model

The ability to distribute an application in RuggedJ stems from the uniform object model that we apply to all objects. Recall Figure 1, which shows the transformation of a single user class `X` to conform to the RuggedJ object model. We discuss first the instance parts of the transformed class, and defer the static parts to Section 3.4.5.

3.4.1 Generated classes

For each class within the original application we generate three classes and one interface. The generated interface, `X`, encapsulates the instance protocol of the original class `X`. It 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 concrete representations 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 remotely 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 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. In RuggedJ we use programmer input to determine how to partition an application across the network.

All of the classes in an application can be adapted to implement the RuggedJ object model. 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 across implementation techniques, they are so straightforward as to be uninteresting. We therefore focus our attention on the local classes.

3.4.2 Referring to transformed objects

Within rewritten code, we exclusively refer to values with generated interfaces using that interface. This allows varying the implementations of these interfaces among several alternatives (local, proxy, and stub classes) without affecting 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 3.6.3 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.

3.4.3 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 4 shows the relationship between transformed classes (omitting static parts).

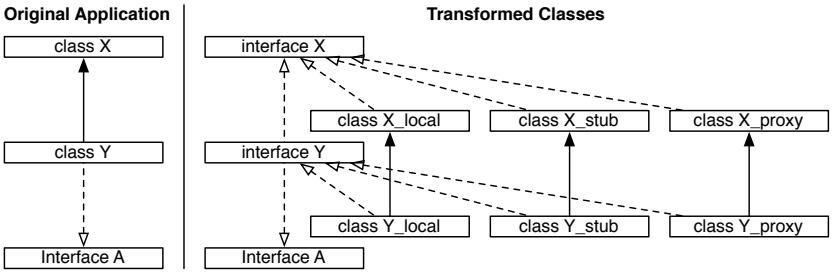


Figure 4 – Inheritance among transformed classes (UML class diagram)

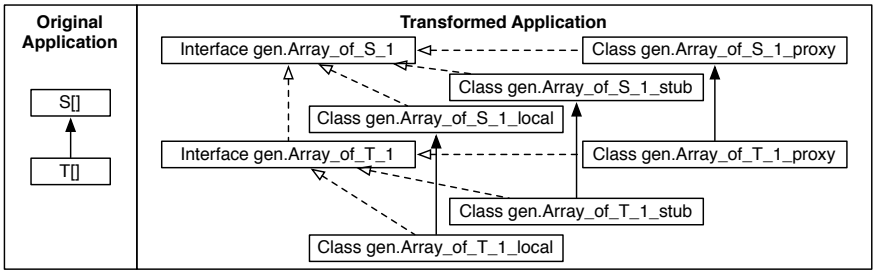


Figure 5 – Generated array types (UML class diagram)

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 instances, 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.

We do not transform interfaces from the original application (in general—see Section 3.6.3 for some exceptions) as they have no state that may be remotely accessed. However we must capture the relationship between a class that implements an interface; we do this by extending the original interface in the generated interface. This maintains the inheritance structure through generated interfaces in the same way that we do for class inheritance.

3.4.4 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 5. 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

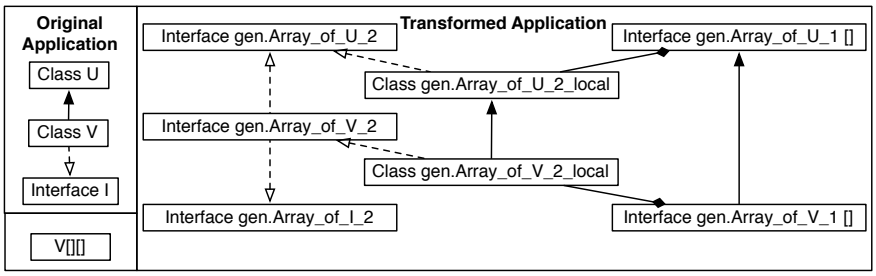


Figure 6 – Multi-dimensional arrays with interfaces (UML class diagram)

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 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 6 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 6 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`.

3.4.5 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 instance 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 1. 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 distribution 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 1, `X_static_proxy`, acts as a per-node cache for the appropriate static local/stub object, and is never instantiated. Accesses to static data in the original application (such as via the `invokestatic` bytecode) are handled by the virtual machine, resolving the class name to access the appropriate data. In our rewritten version, however, we need a static singleton object upon which to invoke methods. Obtaining this reference through the RuggedJ run-time library would be an expensive operation, requiring a hash table lookup for every static access. Instead we store the reference as a static field in the `X_static_proxy` class, which can be obtained through a regular static field access.

3.4.6 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 distribution 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 console 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.

3.5 Method and field transformations

The implementations for most of the generated classes within RuggedJ follow simple templates: the stub and proxy classes each implement every method of the interface, with a standard bytecode sequence that performs a remote method invocation in the case of the stub, or forwards the method call to a referent in the case of the proxy. In the run-time system we optimize the stub in some cases to cache immutable values. The remaining classes, `X_local` and `X_static_local` contain methods and fields copied from the original class. We rewrite the bodies of all copied methods to refer to the RuggedJ object model. This involves several rewrites:

Refer to new types. The first modification that we perform is to update copied method bodies (as well as copied fields) to refer to new types. In most cases this does not require a change. We type values by interface, and have designed our

object model to re-use the original class' name as the interface name. However, there are some cases where we must update type names. As seen in Section 3.6, we generate user-level equivalents for some system classes. In rewritten code we refer exclusively to these user-level types, and so we update any references in copied code. We also generate wrapper classes for arrays that make them conform to the RuggedJ object model. We similarly update references to arrays to correspond to these new types.

Call get and set methods. We generate get and set methods for the fields of each transformed class, allowing us to hide the location of these fields behind the interface. We rewrite the bytecode in copied method bodies to call these methods rather than directly access fields through `putfield` and `getfield` instructions. When calling these methods we take into account the different semantics of superclass methods and fields: methods override, while fields hide. A naïve implementation could access the wrong field if a subclass had a field of the same name and type. We avoid this by naming get and set methods with both the field name and the containing class.

Update method invocations. Since we type references by interface, we update method invocations from `invokevirtual` to `invokeinterface`. The state of the stack required for these bytecodes is identical, so we need only change the operand. The exception to this rewrite is where we have declared a class to be Direct (see Section 3.6.3), and so do not indirect through an interface.

Convert array operations. Array operations pose some difficulty when rewriting. Unlike field instructions (such as `getfield`), array instructions (such as `aaload`) do not encode the type of the array being operated upon (beyond whether it contains objects or primitives). The type of an object array's contents are determined at run time based upon the contents of the stack, and so are not available to us when we rewrite the class. Since we wrap arrays we need to know the type of the content in order to call the correct get or set method. We determine this information through a simple data flow analysis that tracks the array type from its declaration. We use the same mechanism to convert `arraylength` bytecodes to a method invocation on the wrapper object.

Convert static references. Since we extract static data to static singletons, we also update any references to static methods and fields to use these singletons. This transformation is similar to the method and field rewriting described above, but with the minor complication that we must insert a reference to the static singleton before the call. This requires obtaining the reference (which we do through the static proxy class) and inserting it before any method parameters (which we pop to and then restore from local variables).

Convert static methods to instance methods. We transform the bodies of the static methods themselves to account for their change to instance methods. Instance methods contain a reference to the containing object in their local variable slot zero, while static methods have no containing object, and so do not require this reference. When converting from static to instance, we increment the target slot for all local variable accesses by one, creating space for the `this` pointer. This could cause issues with offsets in the bytecode stream, since Java contains shorthand bytecodes to load to and store from low-numbered

local variable slots. The toolkit we use to rewrite, ASM, bypasses this problem by abstracting away the shorthand bytecodes until it produces a final output sequence.

Rewrite monitor operations. Global synchronization in RuggedJ is handled in the distribution library. When rewriting method bodies we convert all synchronization bytecodes (`monitorenter` and `monitorexit`) to call out to the library, which ensures that they are executed correctly.

Wrap and unwrap references. We make extensive use of wrapping both for arrays and for system objects. When passing wrapped objects as arguments across the system boundary from user to system code, or when returning them in the opposite direction, we wrap or unwrap the reference to ensure that the correct object is seen on either side of the boundary. Passing from user to system code requires a simple unwrap operation to obtain the wrapped reference. Wrapping, on the other hand, requires that we check whether the object has been wrapped before to avoid creating two wrappers for a single object. We add a reference to the wrapper in system classes using the JVMTI agent (discussed in Section 3.6.2) which allows us to re-use existing wrappers. During the bytecode rewriting phase we identify those points where references pass from one side of the system boundary to the other, and perform compensating wrapping or unwrapping operations.

A final function of the rewriting phase is to replace allocation sites with references to our transformed classes. Allocation sites are the only occasion where we directly refer to generated classes, rather than to interfaces. Where the original application allocates an object of type `X` (using the `New` bytecode) the transformed version creates either an `X_local` or `X_stub` object, depending on the node upon which the allocation occurs, and a `X_proxy` object if the partitioning policy provided by the developer determines that the object might migrate. 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.

The decision whether to allocate an object locally or remotely, as well as whether to allocate a proxy, is determined at run time by the developer-supplied partitioning policy. These decisions can be made statically (the classes to be allocated are hard-coded into the method bytecode), or dynamically (the partitioning policy is queried whenever the allocation site is reached). The majority of allocations are performed statically, with local objects generated without proxies.

3.6 System classes

The transformations described to this point apply only to user classes, which can be rewritten by a user-defined class loader. The presence of system classes within an application complicates the implementation of the RuggedJ object model. In this section we discuss 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 impact of those constraints upon user code, because we find that system code is

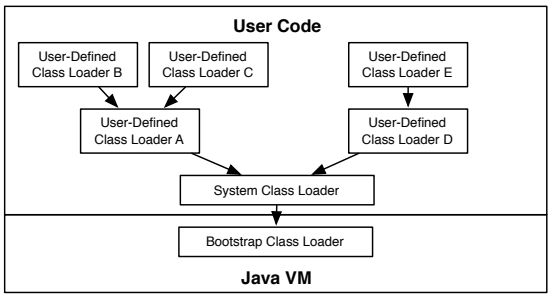


Figure 7 – Classloading in the Java Virtual Machine

generally subject to more constraints than user code. Thus, as we will see in Section 4, the majority of constraints on transforming user code are caused by dependencies on system classes.

3.6.1 Barriers to transformation

Java class loaders [LB98] are organized hierarchically, as shown in Figure 7. The *bootstrap* class loader forms the root of a tree structure, with the *system* class loader as its only child. The bootstrap class loader is implemented within the Java VM, while the system class loader can be replaced with a user-defined class loader when the VM starts up. Any other user-defined class loaders form a tree rooted at the system class loader. A class loading request can explicitly specify the class loader by which it is to be resolved (using the reflective `ClassLoader` class). When the class loader is not explicitly specified, the class is loaded by the class loader responsible for the invoking class. By default, class loaders delegate all class loading requests to their parent in the tree. Thus, a class requested from User-Defined Class Loader E in Figure 7 would be passed through each parent node in the tree to be resolved by the bootstrap class loader. Should the bootstrap class loader fail to resolve the class then the request would be passed back to the system class loader, and so on. If none of the class loaders on the path through the hierarchy can load the class, a `ClassNotFoundException` is thrown.

RuggedJ's transforming class loader is loaded into the VM at boot time as the system class loader. Thus, any class loading requests that are not fulfilled by the bootstrap class loader are intercepted by our class loader, allowing us to rewrite all user code. Due to the complexity incurred by composing multiple user-defined class loaders, we do not allow applications to use custom class loaders.

This class loading structure poses two major problems for our transformation system. The first is that all system classes will be loaded by the bootstrap class loader, meaning that we do not have the opportunity to transform them. Further, while we could override the delegation mechanism and transform the classes within RuggedJ's system class loader, Java's security mechanism would not allow us to load the transformed versions. User-defined class loaders may not load classes with reserved package names (such as `java.*`).

The second, and more fundamental, problem is that the class loading hierarchy imposes visibility constraints. A class can refer only to those classes loaded by the same class loader as itself or by a parent class loader. Thus, classes loaded by any user-defined class loader can refer to any system code (defined by the bootstrap class loader), but system code cannot refer to user code. This means that, even were we

able to load transformed versions of system classes, they could not refer to user code such as the RuggedJ run-time library.

A final barrier to transforming 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 modifying 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.

3.6.2 The RuggedJ JVMTI agent

The Java VM Tool Interface (JVMTI) specification [JVM] provides a set of native interfaces that allow access to many aspects of the JVM's operation. It allows debuggers or profilers to interface with the VM. For example, an agent can extract performance metrics, or could monitor the threads in a running VM. Of interest to us is the bytecode modification functionality of the interface. There are two ways in which bytecode can be modified using the JVMTI: at class-load time and at run time as a response to a class rewriting event. Of the two, the former provides more flexibility. Run-time rewriting is subject to more constraints than load-time, as the modified code must be compatible with the running system.

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 JVMTI agent 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. 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. Our JVMTI agent modifies such classes 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.

Our JVMTI agent is implemented in C, using a custom bytecode modification library. The bytecode rewriting must be implemented in C; simply calling back to our ASM-based Java rewriting library would be tempting, but impractical. In order to rewrite a class this way would require loading of the entire ASM framework, along with the system classes upon which it depends. This would defeat the purpose of the agent, since it would miss rewriting hundreds of system classes before ASM had fully loaded.

The agent is called by the VM after a class is presented for loading by a class loader, but before it is actually loaded. We modify the class and return a new bytecode stream that is then loaded to the VM. This interface represents the major limitation to rewriting with JVMTI—we can modify classes but we cannot create or rename them.

A final limitation to class transformation is the presence of *primordial* classes.

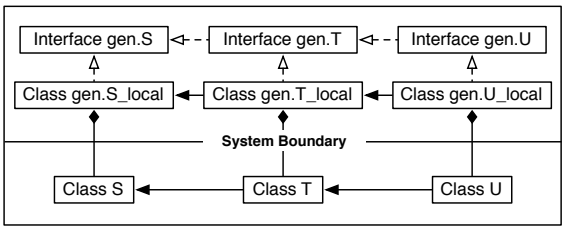


Figure 8 – Wrapping class hierarchy (UML class diagram)

These are approximately seventy classes (with the exact number varying across VM implementations) that cannot be modified at all. Primordial classes are intimately tied to the VM, such as `java.lang.Object` or `java.lang.String`. Depending on the VM implementation, these classes may be hard-coded or directly memory-mapped to optimize startup times, and so cannot be intercepted.

3.6.3 Templates for rewriting

Our strategy when handling system classes is to abstract away the distinction between system and user code, allowing rewritten code to refer to either without special cases. Thus we ensure that all system classes can be made to conform to the RuggedJ object model. Our class transformations use four basic techniques to obtain new types for system classes:

- The local instance of a *Wrapping* class holds a reference to a paired instance of the old type. This allows *any* system class to be remotely referenced, at the cost of wrapping and unwrapping overheads.
- *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 for the target domain (the other transformations can be applied to such classes, but would result in unnecessary overhead). In a distributed system, immutable objects such as `Integer` need not be transformed, as its instances can be replicated as needs demand. Similarly, classes whose instances are always local need not be transformed.

System Wrapping. Wrapping is the most straightforward of the transformation templates and is shown in Figure 8. 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.

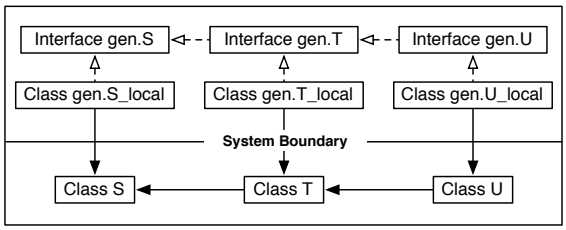


Figure 9 – Extending class hierarchy (UML class diagram)

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. We avoid this by inserting a reference to the wrapper within each wrapped system class, along with get and set methods to access it. Since this involves the modification of system code, we perform this rewrite using the JVMTI agent. The wrapper reference is typed as `Object`, as a system class cannot refer to a user class. Finally, since we cannot add fields to primordial classes, we maintain a hash table for these objects, 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.

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 9. 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

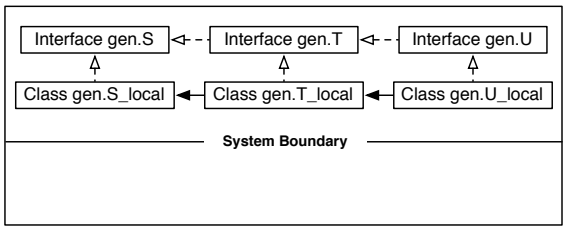


Figure 10 – Promotable class hierarchy (UML class diagram)

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.

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. Since we can rewrite all references to the Promotable class we can ensure that the original class is never referred to, and so is never loaded by the bootstrap class loader.

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 Figure 10. 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 [FSS04], 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.

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

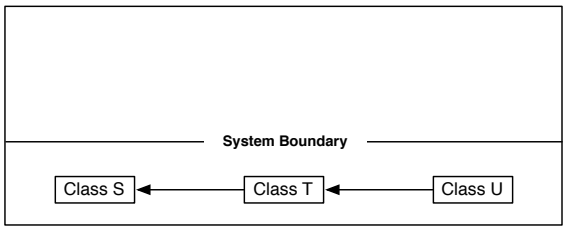


Figure 11 – Direct class hierarchy (UML class diagram)

Table 1 – Subclassing between templates

	Wrapping	Extending	Promotable	Direct
Wrapping	Can subclass	Cannot wrap superclass	Cannot wrap superclass	No interface
Extending	Cannot alter base hierarchy	Can subclass	Cannot alter base hierarchy	No interface
Promotable	Cannot extend wrapper	Can subclass	Can subclass	No interface
Direct	No interface	No interface	No interface	Can subclass

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.

Those classes we designate to be System Direct are not transformed in any way (as shown in Figure 11). 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.

3.6.4 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. The relationships are shown in Table 1.

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 Promotable arguments).

3.6.5 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 compute classification during a pre-processing phase, which we run once per application for a given set of class libraries.

We arrange the various classification templates using a total 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 12.

3.6.6 System class static singletons

Extracting static members from a class is a simple process when transforming user code, but is not possible for system classes. Since we cannot rewrite system code, we cannot change static references (`invokestatic`, `getstatic`, etc.) to use static singletons. Thus, we implement the static local class differently for user and system code.

We refer to the static local class generated for a user or Promotable class as a *mobile static singleton* (MSS). This local class functions as described in Section 3.4.5, and contains implementations of each static method from the original class, as well as versions of each static field. The methods and fields are transformed from static to instance members, allowing the singleton to implement the static interface. Additionally, by transforming static methods to instance methods we remove the dependency on a particular VM: static data is usually stored in a VM-specific manner and cannot easily be moved from node to node, while instance data is stored in the heap and can be migrated (hence *mobile* static singleton).

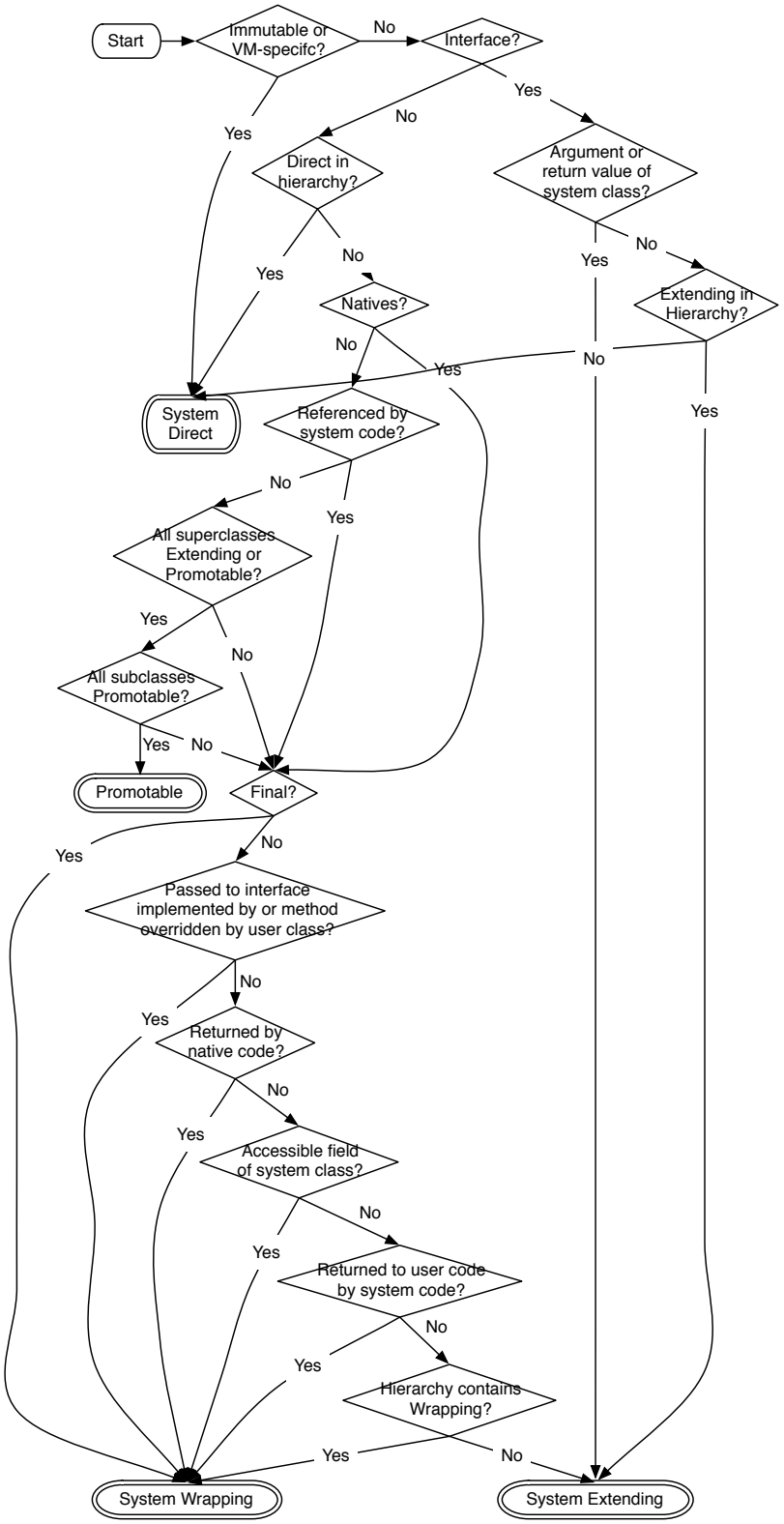


Figure 12 – Classifying system classes

While system classes cannot use static singletons themselves, we make their static state available to remote user code by generating a *pinned static singleton*. This implements the same interface as a mobile static singleton, but can exist on only one node. The static local class contains an instance method with the signature of each static method. In this case it simply acts as a redirector, calling the static method of the system class, allowing that data to be accessed remotely.

Pinned static singletons pose a major barrier to distribution. Not only must all objects of a class with a pinned static singleton be allocated on the same node, but so must any other system classes that refer to the static parts of that class. Fortunately static singletons are required only for classes with static data. As seen in Section 4.3, we do not need static singletons in most cases; ultimately, only 12% on average of the classes we consider require a pinned static singleton.

3.7 User classes

The transformation of system classes constrains that of user code. As discussed in Section 3.6.4, 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.

The four templates for rewriting user code closely mirror those for system code. 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 shown in Section 4, the vast majority of user classes are either User Direct or User Unconstrained.

3.7.1 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 3.4). 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.

3.7.2 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 [TS06] that determine which classes are most likely to be accessed by native code. 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. We ensure that any classes that are likely to be exposed to native code conform to the User Direct, Wrapping or Extending templates. This way they retain a base object upon which native code can operate.

3.7.3 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 system-level interfaces or 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.

3.7.4 Classification

The classification of user code follows a similar approach to that of system classes. Figure 13 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.

4 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.

4.1 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 single-node 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 we perform all rewrites as we would in a distributed system, referring to objects via our new interfaces, etc.

All classifications 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.

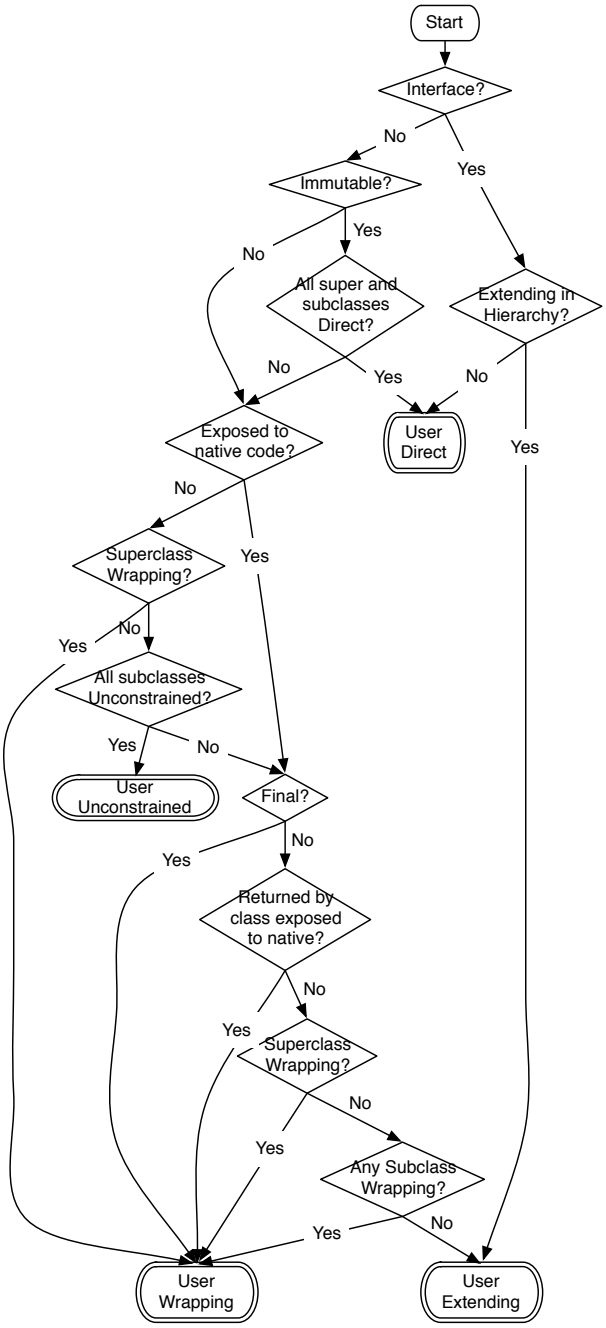


Figure 13 – Classifying user classes

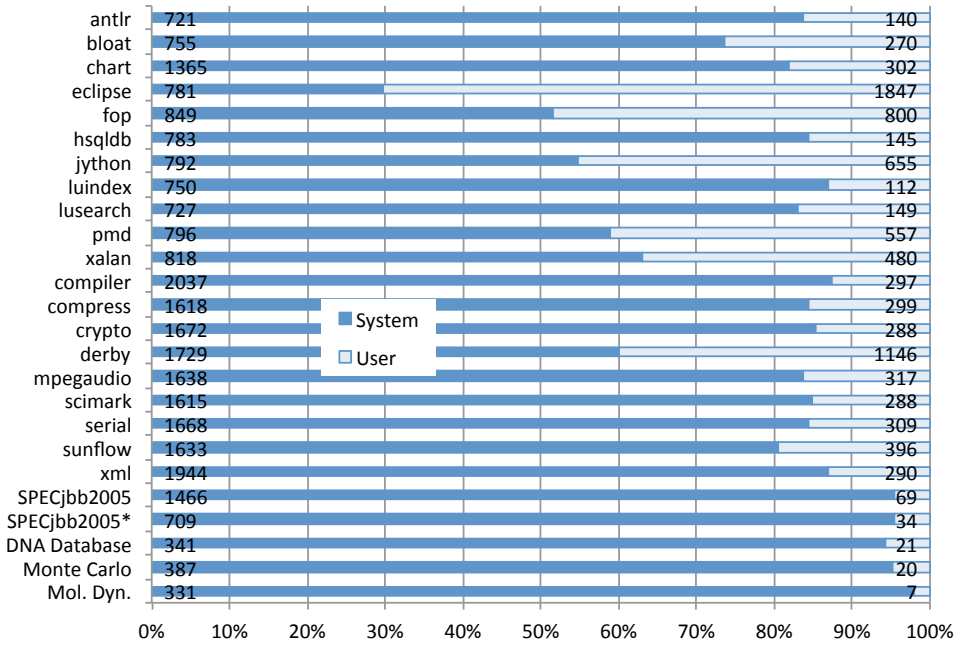


Figure 14 – Mix of system versus user classes (labels show absolute counts)

4.2 Classification

We ran the classification algorithm on applications from different benchmark suites, shown in Figure 14: ten benchmarks from the DaCapo suite (version 2006-10-MR2 [BGH+06]), nine from the SPECjvm2008 suite [SPE08], plus SPECjbb2005 [SPE05]. In addition, we analyzed four distributable applications: a re-implemented *distributable* version of SPECjbb2005 which we call SPECjbb2005*, a DNA database matching application [KN05], and distributable versions of the Monte Carlo and Molecular Dynamic benchmarks from the Java Grande suite [MCH99, Gra]. SPECjbb2005* differs from SPECjbb2005 by distributing the contents of certain key data structures across the "warehouses" of the benchmark to avoid unnecessary non-local accesses.

To obtain an accurate count of the classes referred to by the DaCapo applications, we analyzed them without the DaCapo harness, to ensure that we classified only those classes referred to by the benchmark application.

As Figure 14 shows, the majority of classes (78% 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.

Figure 15 shows that the majority of user classes are split between User Direct (42% of user classes and 10% of the total application) and User Unconstrained (53% of user classes and 13% of the total 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

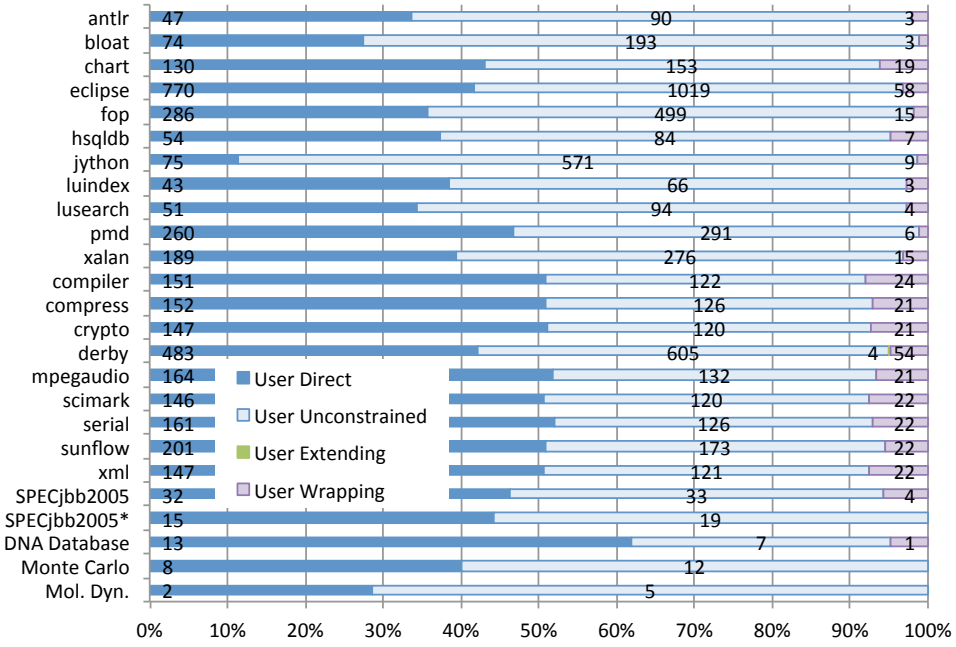


Figure 15 – Classification of user classes (labels show absolute counts)

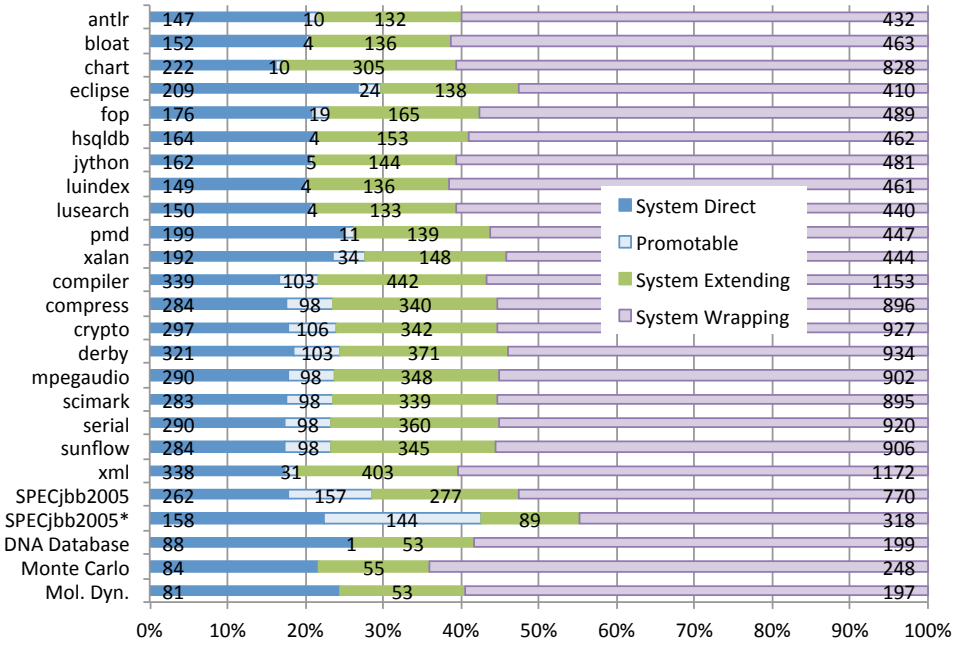


Figure 16 – Classification of system classes (labels show absolute counts)

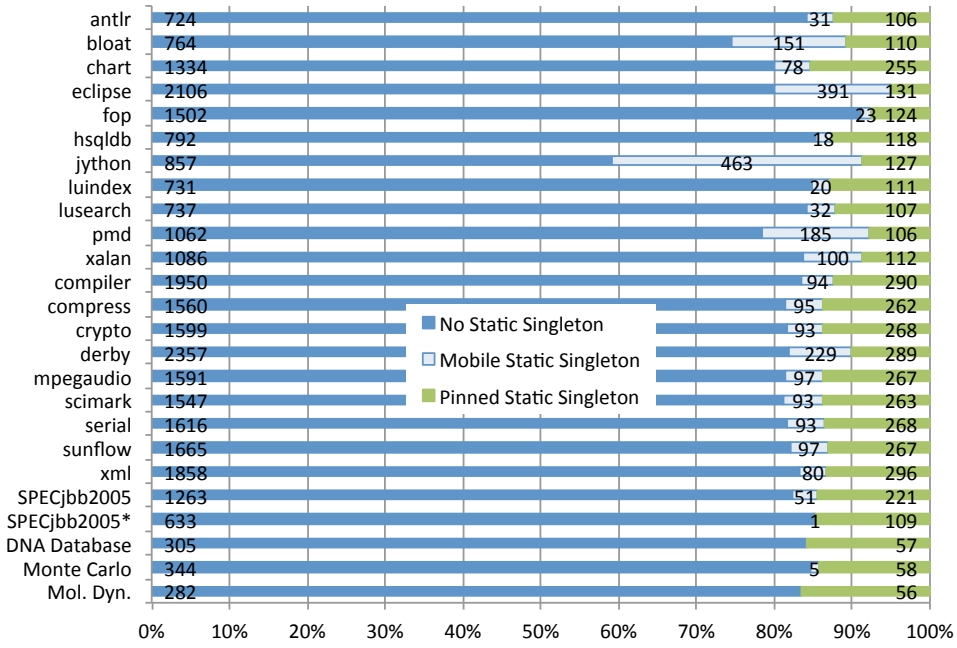


Figure 17 – Elimination of static singletons (labels show absolute counts)

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.

Figure 16 shows that, below the system boundary, System Wrapping classes are the most common, representing 57% of the system classes and 42% of the total application 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 18% of system classes, while 20% of system classes are System Direct. Finally, 3% of classes on average can be promoted.

4.3 Static singletons

While we present a mechanism for handling static data within a distributed system in Section 3.4.5 and Section 3.6.6, we recognize that static singletons pose a major source of overhead. At best, a local static method invocation requires additional work to locate and insert a reference to the static singleton, while at worst every static method invocation could become a remote call. We aim, therefore, to eliminate static singletons in those cases where they are not strictly necessary (for classes that have no static state, or for which static state is immutable). Figure 17 shows that we are able to completely eliminate static singletons for 82% of classes on average across our applications, 12% of classes require a pinned static singleton and the remaining 6% of classes need a mobile static singleton.

4.4 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. For example, with RuggedJ there is little to be gained by distributing an application whose running time is dominated by start-up costs. 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 ten benchmark scores, representing the steady-state throughput. We then ran the transformed code (working from the classification for SPECjbb2005 reported earlier) using the same setup. Overall, transformed SPECjbb2005 produces 64% of the throughput of the untransformed benchmark. By running the transformed SPECjbb2005 under the YourKit Java profiler [You] we found that the overheads of the transformations 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.

Our focus in this paper is the transformation strategy to deploy the RuggedJ object model. For space reasons we are unable here to provide full performance results for *distributed* execution with the RuggedJ run-time system. Full details can be found in McGachey's PhD dissertation [McG10], which considers several realistic benchmark applications and how to distribute them. In brief, we have been able to show that the rewritten *distributable* SPECjbb2005* version of SPECjbb2005 scales similarly to the original, though at the cost of some degradation in throughput due to transformation *plus* the overhead for communicating among *distributed* warehouses. Moreover, whereas scaling the number of warehouses beyond the number of cores available on a single node saturates the cores for untransformed SPECjbb2005 and SPECjbb2005*, distributed SPECjbb2005* exploits cores on additional nodes to produce further scaling for larger numbers of warehouses. Figure 18 shows the performance of the untransformed benchmarks running on a single 16-core node against that of distributed SPECjbb2005* running on a cluster of two 16-core nodes. Interestingly, the non-distributed SPECjbb2005* outperforms the original SPECjbb2005 because it decentralizes key data structures and avoids some contention present in the original benchmark. We have also demonstrated that naturally distributable parallel benchmarks such as the DNA Database matching benchmark scale only slightly less than perfectly up to 48 cores on a 3-node cluster of 16-core machines.

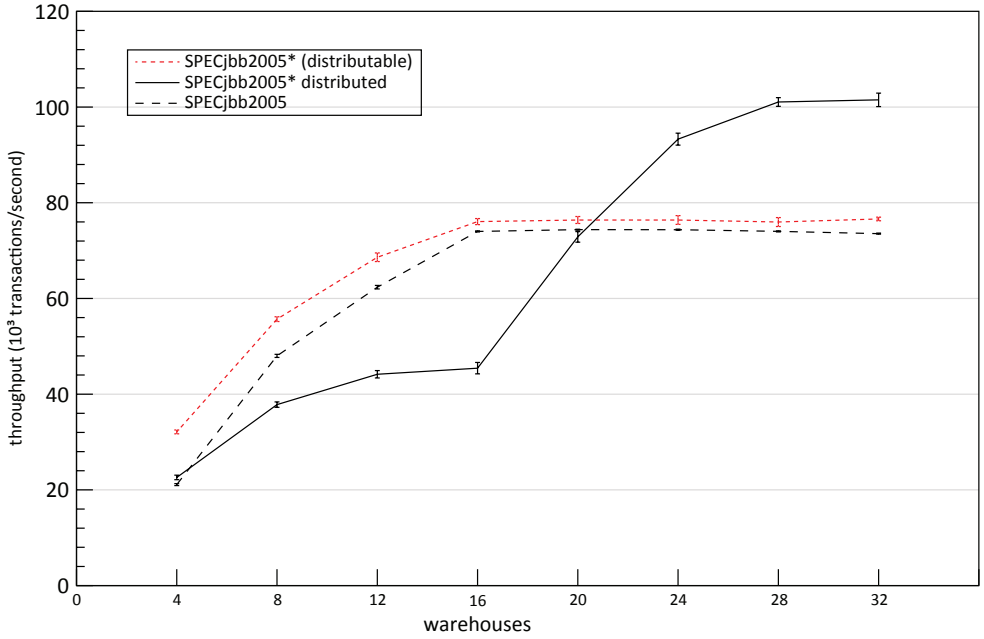


Figure 18 – SPECjbb2005* performance (with 95% confidence intervals)

5 Related Work

The issue of rewriting system code has been considered in the past. The Twin Class Hierarchy (TCH) approach [FSS04] 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 [SAPE05] 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 [TSH05, TS02, TS09], which also

performs transparent distribution, though targeted at a different domain. J-Orchestra *partitions* the *classes* of an application for heterogeneity across a predetermined set of hosts while RuggedJ *distributes* the *instances* of the application dynamically across an arbitrary number of hosts in a cluster to enable scaling. This difference is visible in our rewriting approach: RuggedJ performs all rewriting at class load time, taking advantage of the particular configuration (e.g., number of available nodes) to distribute the work of the application scalably, while J-Orchestra is able to use its advance knowledge of the platform to generate a customized `jar` file ahead of time for each site to partition the application to take advantage of heterogeneous capabilities.

Thus, J-Orchestra allocates all instances of a given class at a single pre-defined node. Distribution of instances in J-Orchestra requires subsequent migration. In contrast, RuggedJ enables instance allocation to be controlled via run-time allocation policies. Simple policies often work very well. For example, RuggedJ generally allocates work units of the application (implementers of `java.lang.Runnable`) on remote nodes, with subsequent allocations by those work units on the local node. Allocation policies can be refined on a per-node plus per-allocation-site basis, statically at rewrite time, or dynamically (via policy call-backs) at run time. Policy choices (both at per-node class load time and at run time) have access to cluster metadata (such as the number of nodes and their capabilities) to decide how to handle each allocation site.

The RuggedJ distribution policies also allow dynamic declaration of instance *immutability*, allowing them to *replicated* once they have reached their point of immutability. This mechanism is useful for initializing instances locally (for efficiency) and then allowing their state to be replicated across nodes once they have been initialized. We must be careful here to preserve identity of the replicas. For example, synchronizing on a replica requires synchronizing on the original at its home node.

J-Orchestra assumes that any instance of a *mobile* class can migrate from the node on which it was allocated. Instances of *anchored* classes (similar to our system classes) must remain at their home node. In RuggedJ migration is controlled by the distribution policy, which declares which classes may have migrating instances (and so must be proxied), while avoiding the run-time overhead of proxies for instances that never migrate. The policy specification allows migration of an object to be triggered when it is returned from a method, after some number of remote invocations on it, or at call/return of arbitrary methods. The latter mechanism specifies the class and method into which a policy callback is inserted at the start or end, along with a local variable to be passed to the callback at run-time holding the object to migrate.

In summary, while J-Orchestra shares many similarities to RuggedJ, the flexibility of our distribution policies allows more refinement of the ways in which different classes are rewritten. For example, in RuggedJ, non-migratable classes can always be referred to directly if local, whereas J-Orchestra treats all remotely accessible classes uniformly.

Addistant [TSCI01] enables the distribution of "legacy" Java applications (the developers define legacy as any Java software written without distribution). Similarly to RuggedJ, the system makes use of load-time bytecode rewriting using the Javassist transformation tool, and provides a run-time system. It requires no modification to the Java VM. Developers are constrained to partitioning by *class* (instances of the class will either all be local or all be remote) unless they explicitly request *heterogeneous* transformation allowing instances to reside both locally and remotely. RuggedJ infers this classification automatically based on the origin of the class (system or user) and the ways in which its instances are used (passed to system code or not), and always generates heterogeneous code matching the inferred classification. The Addistant

run-time system has the interesting feature that it automatically delivers rewritten source code to the respective nodes, simplifying application deployment.

The major contribution of the Addistant system is its object model. Like RuggedJ, Addistant uses proxies to forward remote references to the appropriate objects. Its classification scheme allows system code to integrate into distributed applications, based on two properties. *Modifiability* refers to the capacity of their tool to rewrite bytecode; this is similar to our differentiation between user and system code. *Heterogeneity* refers to the references that a class holds; a heterogeneous class can refer to both local and remote instances. Based on these two criteria, Addistant defines four approaches to developing proxies. The *Replace* approach is usable when a class is modifiable and non-heterogeneous. It assigns the class to one node and generates a proxy with the same name on all remote nodes. The *Rename* approach is used when the class is unmodifiable, but is referred to only by modifiable classes. In this case the system creates a proxy with a different name, and rewrites all references to point to this proxy. The *Subclass* approach allows heterogeneity: the proxy is a subclass of the base class. References pointing to the base class can instead refer to the proxy. Finally the *Copy* approach is used for primitive and immutable objects, with replicas passed around the network.

Addistant takes the same approach to object equality as RuggedJ; equality is guaranteed by ensuring that exactly one proxy object per host refers to any given master object. It also uses a similar thread affinity system to RuggedJ, ensuring that callbacks from remote methods are handled by the same thread.

6 Conclusion

Bytecode transformation allows RuggedJ to integrate original application code within its distribution run-time. 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 uniform object model for distribution, as well as the classification process to determine which transformation template should be applied to each class. The RuggedJ object model is flexible and supports remote referencing, remote invocation, and migration. The run-time system ensures proper Java behavior for synchronization, threads, and object identity. Finally, we have described the implementation of our prototype system, and presented an analysis of the classification process when applied to several benchmarks. We have demonstrated the feasibility and generality of our transformations, and shown that their performance overheads are low enough for practical use: 36% on SPECjbb2005. The RuggedJ transformation techniques presented here offer a complete and flexible strategy for distributing Java applications to achieve scalability in clusters of many Java virtual machines.

References

[ASM] ASM [online]. Available from: `http://asm.ow2.org`.

[BGH+06] 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 *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, Portland, Oregon, October 2006. doi:10.1145/1167473.1167488.

[BLC] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. Available from: http://asm.ow2.org/current/asm-eng.pdf.

[BS98] Boris Bokowski and André Spiegel. Barat — a front-end for Java. Technical Report B 98-09, Institut für Informatik, Freie Universität Berlin, December 1998. Available from: http://www.inf.fu-berlin.de/inst/pubs/tr-b-98-09.abstract.html.

[Chi00] Shigeru Chiba. Load-time structural reflection in Java. In Elisa Bertino, editor, *European Conference on Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336, Cannes, France, June 2000. Springer. doi:10.1007/3-540-45102-1_16.

[CN03] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *International Conference on Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*, pages 364–376, Erfurt, Germany, September 2003. Springer. doi:10.1007/978-3-540-39815-8_22.

[Dah98] Markus Dahm. Byte code engineering with the JavaClass API. Technical Report B 98-17, Institut für Informatik, Freie Universität Berlin, 1998. Available from: http://www.inf.fu-berlin.de/inst/pubs/tr-b-98-17.abstract.html.

[FSS04] Michael Factor, Assaf Schuster, and Konstantin Shagin. Instrumentation of standard libraries in object-oriented languages: The Twin Class Hierarchy approach. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 288–300, Vancouver, Canada, October 2004. doi:10.1145/1028976.1029000.

[Gra] Java Grande benchmark suite [online]. Available from: http://www.epcc.ed.ac.uk/research/java-grande.

[HS06] Shan Shan Huang and Yannis Smaragdakis. Easy language extension with Meta-AspectJ. In *28th ACM International Conference on Software Engineering*, pages 865–868, Shanghai, China, May 2006. doi:10.1145/1134285.1134436.

[JMa] The JMangler project [online]. Available from: http://roots.iai.uni-bonn.de/research/jmangler.

[JVM] The JVM tool interface [online]. Sun Microsystems, Inc. Available from: http://java.sun.com/j2se/1.5.0/docs/guide/jvmti.

[KCA01] Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler—a framework for load-time transformation of Java class files. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, pages 100–110, Florence, Italy, November 2001. doi:10.1109/SCAM.2001.972671.

[KHH+01a] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with As-

pectJ. *Communications of the ACM*, 44(10):59–65, October 2001. doi:10.1145/383845.383858.

[KHH+01b] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Knudsen [Knu01], pages 327–353. doi:10.1007/3-540-45337-7_18.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jeanmarc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the Eleventh European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. doi:10.1007/BFb0053381.

[KN05] T. M. Keane and T. J. Naughton. DSEARCH: sensitive database searching using distributed computing. *Bioinformatics*, 21(8):1705–1706, 2005. doi:10.1093/bioinformatics/bti163.

[Knu01] Jørgen Lindskov Knudsen, editor. *European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, Budapest, Hungary, June 2001. Springer. doi:10.1007/3-540-45337-7.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36–44, Vancouver, Canada, October 1998. doi:10.1145/286936.286945.

[McG10] Philip McGachey. *Transparent Distribution for Java Applications*. PhD thesis, Purdue University, West Lafayette, IN, May 2010.

[MCH99] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of Java Grande benchmarks. In *ACM Conference on Java Grande*, pages 72–80, San Francisco, CA, June 1999. doi:10.1145/304065.304101.

[MHM09a] Phil McGachey, Antony L. Hosking, and J. Eliot B. Moss. Classifying Java class transformations for pervasive virtualized access. In *International Conference on Generative Programming and Component Engineering*, pages 75–84, Denver, CO, October 2009. doi:10.1145/1621607.1621620.

[MHM09b] Phil McGachey, Antony L. Hosking, and J. Eliot B. Moss. Pervasive load-time transformation for transparently distributed Java. *Electronic Notes in Theoretical Computer Science*, 253(1):47–64, December 2009. Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE), York, UK (March 2009). doi:10.1016/j.entcs.2009.11.014.

[SAPE05] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for Java. In *International Conference on Automated Software Engineering*, pages 114–123, Long Beach, CA, November 2005. doi:10.1145/1101908.1101927.

[SPE05] Java server benchmark (SPECjbb2005) [online]. 2005. Standard Performance Evaluation Corporation. Available from: http://www.spec.org/jbb2005.

[SPE08] Java virtual machine benchmarks (SPECjvm2008) [online]. 2008. Standard Performance Evaluation Corporation. Available from: http://www.spec.org/jvm2008/.

[TS02] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java application partitioning. In Boris Magnusson, editor, *European Conference on Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 178–204, Málaga, Spain, June 2002. Springer. doi:10.1007/3-540-47993-7_8.

[TS06] Eli Tilevich and Yannis Smaragdakis. Transparent program transformations in the presence of opaque code. In *International Conference on Generative Programming and Component Engineering*, pages 89–94, Portland, OR, October 2006. doi:10.1145/1173706.1173720.

[TS09] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Transactions on Software Engineering and Methodology*, 19(1):1–40, 2009. doi:10.1145/1555392.1555394.

[TSCI01] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A bytecode translator for distributed execution of "legacy" Java software. In Knudsen [Knu01], pages 236–255. doi:10.1007/3-540-45337-7_13.

[TSDNP02] Éric Tanter, Marc Ségura-Devillechaise, Jacques Noyé, and José M. Piquer. Altering Java semantics via bytecode manipulation. In *International Conference on Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 283–298, Pittsburgh, PA, October 2002. doi:10.1007/3-540-45821-2_18.

[TSH05] Eli Tilevich, Yannis Smaragdakis, and Marcus Handte. Appletizing: Running legacy Java code remotely from a Web browser. In *IEEE International Conference on Software Maintanance*, pages 91–100, Budapest, Hungary, September 2005. doi:10.1109/ICSM.2005.25.

[You] The YourKit Java profiler [online]. Available from: http://www.yourkit.com.

About the authors

Phil McGachey is Senior Member of technical staff at VMware. Contact him at pmcgachey@vmware.com.

Antony L. Hosking is Associate Professor of Computer Science at Purdue University. Contact him at hosking@cs.purdue.edu, or visit http://www.cs.purdue.edu/~hosking.

J. Eliot B. Moss is Professor of Computer Science at the University of Massachusetts. Contact him at moss@cs.umass.edu, or visit http://www.cs.umass.edu/~moss.

Acknowledgments This work is supported by the National Science Foundation under grants Nos. CNS-0720505/0720242, CNS-0551658/0509186, and CCF-0540866/05-40862, 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.