Scalable Application Partitioning with J-Orchestra

Eli Tilevich and Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{tilevich, yannis}@cc.gatech.edu
http://j-orchestra.org

ABSTRACT

J-Orchestra is an automatic partitioning system for Java programs. J-Orchestra takes as input a Java application in bytecode format and transforms it into a distributed application, running across multiple Java Virtual Machines. To accomplish such automatic partitioning, J-Orchestra uses bytecode rewriting to substitute method calls with remote method calls, direct object references with proxy references, etc. The partitioning is performed without programming and without making any modifications to the JVM or its standard runtime classes. The main novelty and source of scalability of J-Orchestra is in its approach to dealing with unmodifiable code (e.g., Java system classes). The approach consists of an analysis algorithm and a rewrite algorithm that injects code to transform direct references into indirect and vice versa at run-time. To show that our approach scales, we used J-Orchestra to partition a large commercial application (the JBits FPGA simulator by Xilinx) into a client-server application, with the client partition running on a Windows laptop and the server partition running on a Unix server. No access to the source code or explicit programming was required.

1 INTRODUCTION

The focus of distributed computing has been shifting from "distribution for parallelism" to "resource-driven distribution", with the resources of an application being naturally remote to each other or to the computation. Because of this shift, more and more centralized applications need to be adapted for distributed execution. Various centralized applications, written without any distribution in mind, might need to be changed to move certain parts of their execution functionality to a remote machine. Examples abound. A local database grows too large and needs to be moved to a powerful server, becoming remote to the rest of the application. An application wants to redirect its output to a superior remote graphical screen or to receive input from a remote digital camera. A desktop application when executed on a PDA might not find all the referenced APIs and their corresponding hardware resources available locally and wants to access them remotely.

All the aforementioned scenarios give rise to application partitioning. Application partitioning is the task of splitting up the functionality of a centralized application into distinct entities running across different network sites. A standard way to accomplish such partitioning is to modify the source code of the original application by hand to use a middleware mechanism. This approach is tedious, error prone, and often simply infeasible due to the unavailability of source code, which is usually the case for commercial applications. We present an alternative approach that entails using a tool that under human guidance handles all the tedious details of distribution. This relieves the programmer of the necessity to deal with middleware directly and to understand all the potentially complex data sharing through pointers. Our tool, J-Orchestra, operates on binary (Java bytecode) applications and enables the user to determine object placement and mobility to obtain a meaningful partitioning. The application is then rewritten to be partitioned automatically and different parts can run on different machines, on unmodified versions of the Java VM. For a large subset of Java, the resulting partitioned application is guaranteed to behave exactly like its original, centralized version. The requirement that the VM not be modified is important. We do not want to change the runtime, both because of deployment reasons (it is easy to run a partitioned application on a standard VM) and because of complexity reasons (Java code is platform-independent but the runtime system has a platform-specific, native-code implementation).

The conceptual difficulty of performing application partitioning in general-purpose languages (like Java, C#, but also C, C++, etc.) is that programs are written to assume a shared memory: an operation may change data and expect the change to be visible through all other pointers (*aliases*) to the same data. The conceptual novelty of J-

Orchestra (compared to past partitioning systems [12][20][23] and distributed shared memory systems [1][3][5][24]) consists of addressing the problems resulting from inability to analyze and modify all the code under the control flow of the application. Such unmodifiable code is usually part of the runtime system on which the application is running. In the case of Java, this runtime is the Java VM. In the case of free-standing applications, the runtime is the OS. Without complete control of the code, execution is in danger of letting a reference to a remote object get to code that is unaware of remoteness. Prior partitioning systems have ignored the issues arising from unmodifiable code and have had limited scalability, as a result. J-Orchestra features a novel rewrite tool that ensures that, at run-time, references are always in the expected form ("direct" = local or "indirect" = possibly remote) for the code that handles them. The result is that J-Orchestra can split code that deals with system resources, safely running, e.g., all sound synthesis code on one machine, while leaving all unrelated graphics code on another.

In a previous publication [22] we described J-Orchestra's general partitioning approach and the novelty of its rewriting engine. This paper updates the description of our rewrite algorithm with new features that allow it to scale better (especially *call-site wrapping*, which enables the user to restrict object mobility by "anchoring" objects on specific sites). Most importantly, however, the present paper confirms our claim of scalability of our approach: J-Orchestra can handle realistic applications, as it allows arbitrary partitioning without requiring an understanding of the internals of the application. We have used J-Orchestra to partition a commercial, third-party, binary-only application (the JBits FPGA simulator by Xilinx) so that it can be remotely controlled and monitored.

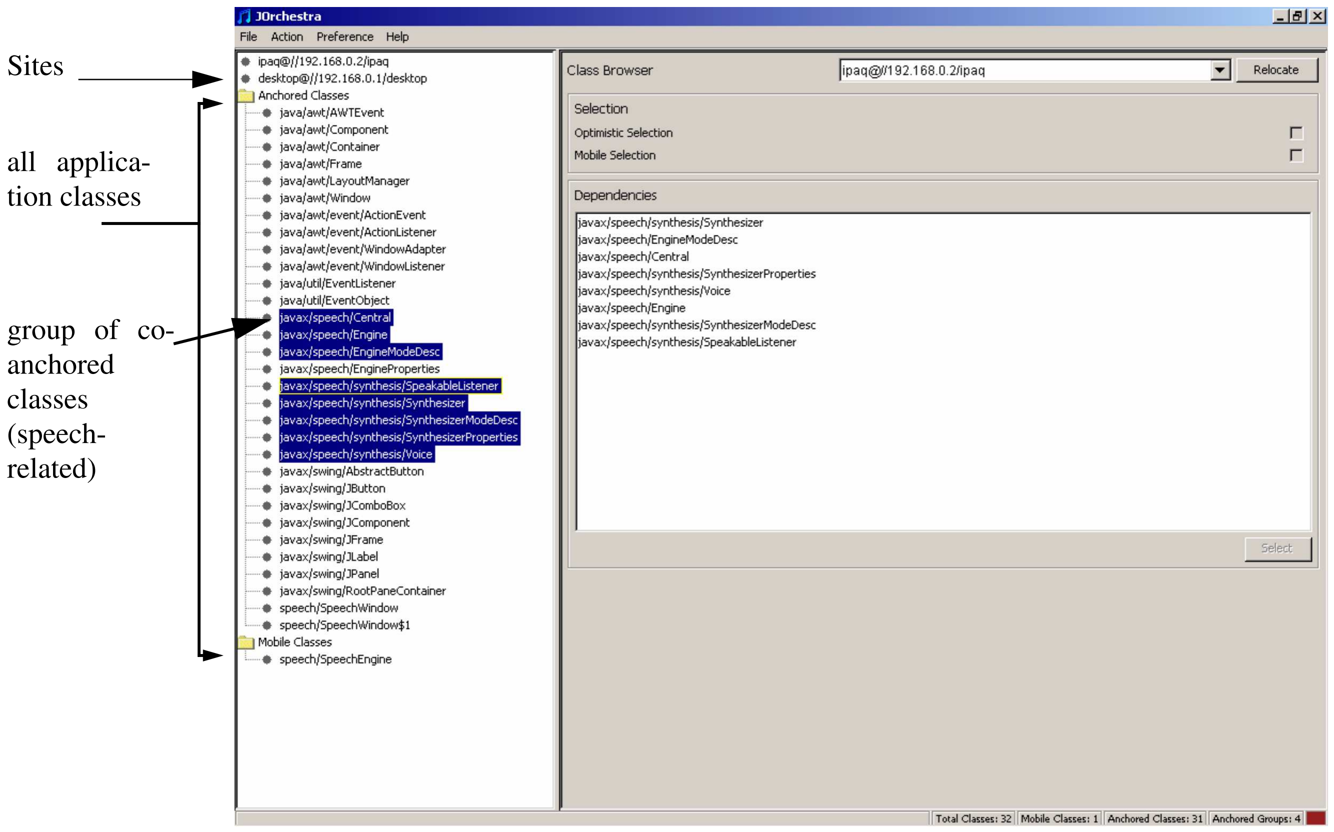
Additionally, in this paper we identify the environment features that make J-Orchestra possible. We believe that partitioning systems following the principles laid out by J-Orchestra are valuable in modern high-level run-time systems like the Java VM or Microsoft's CLR. We explain in detail how the rich type information in the interfaces of system services enables resource-driven distribution. (Although our low-level presentation is in Java, all concepts are equally applicable to other high-level runtimes, like the CLR.)

2 USER VIEW OF J-ORCHESTRA

Figure 1 shows a screenshot of J-Orchestra in the process of partitioning a small but realistic example application. The original example Swing application showcases the Java Speech API and works as follows: the user chooses predefined phrases from a drop-down box and the speech synthesizer pronounces them. As a motivation for partitioning, imagine a scenario when this application needs to be run on a machine, e.g., a PDA, that either has no speakers (hardware resource) or does not have the Speech API installed (software resource). The idea is to partition the original application in a client-server mode so that the graphical partition (i.e., the GUI) would run on a PDA and it will control the speech partition which would run on a desktop machine. We choose this example because it fits well into the realm of applications amenable for automatic application partitioning. The locality patterns here are very clear and defined by the specific hardware resources (graphical screen and speech synthesizer) and their corresponding classes (Swing and Speech API).

Figure 1 shows J-Orchestra at a point when it has finished importing all the referenced classes of the original application and has run its classification algorithm (Section 4.1) effectively dividing them into two major groups represented by tree folders *anchored* and *mobile*.

- Anchored classes control specific hardware resources and make sense within the context of a single JVM. Their instances must run on the JVM that is installed on the machine that has the physical resources controlled by the classes. J-Orchestra clusters anchored classes into groups for safety: Intuitively, classes within the same anchored group reference each other directly and as such must be co-located during the execution of the partitioned application. If classes from the same group are placed on the same machine, the partitioned application will never try to access a remote object as if it were local, which would cause a fatal run-time error. J-Orchestra's classification algorithm (Section 4.1) has created four anchored groups for this example. One group contains all the referenced speech API classes. The remaining groups specify various Swing classes. While classes within the same anchored group can not be separated, anchored groups can be placed on different network sites. In our example,



```
ipaq@//192.168.0.2/ipaq
desktop@//192.168.0.1/desktop
Anchored Classes
    java/awt/AWTEvent
    java/awt/Component
    java/awt/Container
    java/awt/Frame
    java/awt/LayoutManager
    java/awt/Window
    java/awt/event/ActionEvent
    java/awt/event/ActionListener
    java/awt/event/WindowAdapter
    java/awt/event/WindowListener
    java/util/EventListener
    java/util/EventObject
    javax/speech/Central
    javax/speech/Engine
    javax/speech/EngineModeDesc
    javax/speech/EngineProperties
    javax/speech/synthesis/SpeakableListener
    javax/speech/synthesis/Synthesizer
    javax/speech/synthesis/SynthesizerModeDesc
    javax/speech/synthesis/SynthesizerProperties
    javax/speech/synthesis/Voice
    javax/swing/AbstractButton
    javax/swing/JButton
    javax/swing/JComboBox
    javax/swing/JComponent
    javax/swing/JFrame
    javax/swing/JLabel
    javax/swing/JPanel
    javax/swing/RootPaneContainer
    speech/SpeechWindow
    speech/SpeechWindow$1
Mobile Classes
    speech/SpeechEngine
```

Class Browser: ipaq@//192.168.0.2/ipaq

Optimistic Selection ☐
Mobile Selection ☐

Dependencies:

```
javax/speech/synthesis/Synthesizer
javax/speech/EngineModeDesc
javax/speech/Central
javax/speech/synthesis/SynthesizerProperties
javax/speech/synthesis/Voice
javax/speech/Engine
javax/speech/synthesis/SynthesizerModeDesc
javax/speech/synthesis/SpeakableListener
```

Total Classes: 32 | Mobile Classes: 1 | Anchored Classes: 31 | Anchored Groups: 4

Figure 1. Example user interaction with J-Orchestra. An application controlling speech output is partitioned so that the machine doing the speech synthesis is different from the machine controlling the application through a GUI.

all the Swing classes anchored groups should probably be placed on the site that will handle the GUI of the partitioned application.

- Mobile classes do not reference system resources directly and as such can be created on any JVM. Mobile classes do not get clustered into groups, except as an optimization suggestion. Instances of mobile classes can move to different JVMs independently during the execution to exploit locality. Supporting mobility requires adding some extra code to mobile classes at translation time to enable them to interact with the runtime system. Mobility support mechanisms create overhead that can be detrimental for performance if no mobility scenarios are meaningful for a given application. To eliminate this mobility overhead, a mobile class can be *anchored by choice*. We discuss anchoring by choice and its implications on the rewriting algorithm in Section 4.3.

J-Orchestra's GUI represents each network node in the distributed application by a dedicated tree folder. The user then drag-and-drops classes from the anchored and mobile folders to their destination network site folder. Putting an anchored class in a particular network folder assigns its final location. For a mobile class, it merely assigns its initial creation location. Later, an instance of a mobile object can move as described by a given mobility policy. When all classes are assigned to destination folders, the J-Orchestra rewriting tool transforms the original centralized application into a distributed application. At the end, J-Orchestra puts all the modified classes, generated supporting classes, and J-Orchestra run-time configuration files into `jar` files, one per destination network site.

At run-time, J-Orchestra employs its runtime service to handle such tasks as remote object creation, object mobility, and various bookkeeping tasks.

3 CONCEPTUAL CONTRIBUTION

In abstract terms, the problem that J-Orchestra solves is *to emulate a shared memory abstraction for unsuspecting applications without changing the runtime system for these applications*. Two important points are worth mentioning. First, the requirement of not changing the run-time system while supporting unsuspecting applications distinguishes J-Orchestra from traditional Distributed Shared Memory (DSM) systems. (The related work section offers a more complete comparison.) Second, the implicit assumption is that of a pointer-based language. It is conceptually trivial to support a shared memory abstraction in a language environment where no sharing of data through pointers (aliases) is possible. Although it may seem obvious that realistic systems will be based on data sharing through pointers,[1] the lack of data sharing has been a fundamental assumption for some past work in partitioning systems—e.g., the Coign approach [12].

It is worth asking why mature partitioning systems have not been implemented in the past. For example, why is there not a technology that will alow the user to partition a platform-specific binary (e.g., an x86 executable) so that different parts of the code can run on different machines? We argue that the problem can be addressed much better in the context of a high-level, object-oriented runtime system, like the JVM or the CLR, than in the case of a platform-specific binary and runtime. There are three concrete problems that need to be overcome before partitioning is possible:

1. The granularity of partitioning needs to be coarse enough: the user needs to have a good vocabulary for specifying different partitions. High-level, object-oriented runtime systems, like the Java VM, help in this respect because they allow the user to specify the partitioning at the level of objects or classes, as opposed to memory words.
2. A mechanism that adds an indirection to every pointer access needs to be established. This involves some engineering complexity, especially under the requirement that the runtime system remain unmodified.
3. The indirection needs to be maintained even in the presence of unmodifiable code. Unmodifiable code is usually code in the application's runtime system. For example, in the case of a stand-alone executable running on an unmodified operating system, the program may create entities of type "file" and pass them to the operating system. If these files are remote, a runtime error will occur when they are passed to the unsuspecting OS. Addressing the problem of adding indirection in the presence of unmodifiable code is the main novelty of J-Orchestra. This problem, in different forms, has plagued not just past partitioning systems but also traditional Distributed Shared Memory systems. Even page-based DSMs often see their execution fail because protected pages get passed to code (e.g., an OS system call expecting a buffer) that is unaware of the mechanism used to hide remoteness.

We now look at the problem in more detail, in order to see the complications of adding indirection to all pointer references. The standard approach to such indirection is to convert all direct references to indirect references by adding proxies. This creates an abstraction of shared memory where proxies hide the actual location of objects—the actual object may be on a different network site than the proxy used to access it. This abstraction is necessary for correct execution of the program across different machines because of *aliasing*: the same data may be accessible through different names (e.g., two different pointers) on different network sites. Changes introduced through one name/pointer should be visible to the other, as if on a single machine. Figure 2 shows schematically the effects of the indirect referencing approach. This indirect referencing approach has been used in several prior systems [17][20][23].

Since one of our requirements is to leave the runtime system unchanged, we cannot change the JVM's pointer/reference abstraction. Instead, J-Orchestra rewrites the entire partitioned application to introduce proxies for every reference in the application. Thus, when the original application would create a new object, the partitioned application will also create a proxy and return it; whenever an object in the original application would access another object's fields, the corresponding object in the partitioned application would have to call a method in the proxy to get/set the field data; whenever a method would be called on an object, the same method now needs to be called on the object's proxy; etc.

1. The pointers may be hidden from the end user (e.g., data sharing may only take place inside a Haskell monad). The problems identified and addressed by J-Orchestra remain the same regardless of whether the end programmer is aware of the data sharing or not.

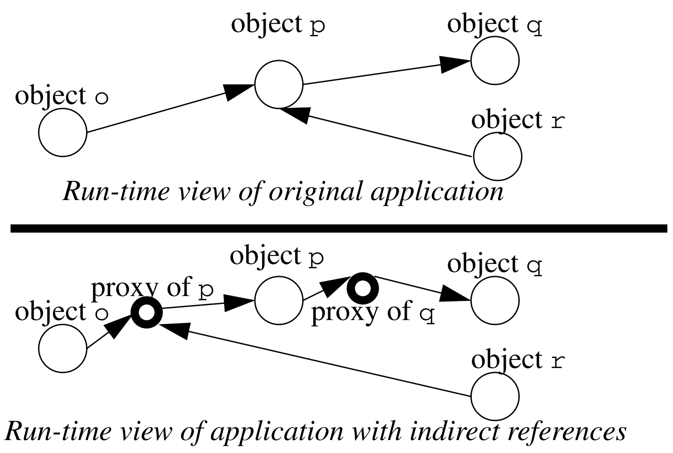

object p
object q
object o
object r
Run-time view of original application
object p
object q
proxy of p
object o
proxy of q
object r
Run-time view of application with indirect references

Figure 2: Results of the indirect reference approach schematically. Proxy objects could point to their targets either locally or over the network.

The difficulty of this rewrite approach is that it needs to be applied to *all code that might hold references to remote objects*. Nevertheless, this is not just the code of the original application, but also the code inside the runtime system. In the case of the Java VM, such code is encapsulated by system classes that control various system resources through native code. Java VM code can, for instance, have a reference to a thread, window, file, etc., object created by the application. Since we cannot modify the runtime system code, however, there is no way to make it aware of the indirection. For instance, we cannot change the code that performs a file operation to make it access the file object correctly for both local and remote files: the file operation code is part of the Java VM (i.e., in machine-specific binary code) and partly implemented in the operating system. If a proxy is passed instead of the expected object to runtime system code that is unaware of the distribution, a run-time error will occur. Without changing the platform-specific runtime (JVM+OS) of the application, we cannot enable remoteness for all of our code.[2] (For simplicity, we assume the application itself does not contain native code—i.e., it is a "pure Java" application.)

J-Orchestra effectively solves many of the problems of dealing with unmodifiable code by partitioning *around* unmodifiable code. There are two parts to the solution: the first is the classification algorithm: a static analysis that determines which classes should be co-located. The second is the rewrite algorithm, which inserts the right code in the partitioned application so that, at run-time, indirect references are converted to direct and vice versa when they pass from mobile to anchored code. In order to perform classification, even though we cannot analyze the platform-specific binary code for every platform, J-Orchestra employs a heuristic that relies on the type information of the interfaces to the Java runtime (i.e., the type signatures of Java system classes). This is another way in which high-level, object-oriented runtime systems make application partitioning possible.

The end result is that, unlike past systems [17][20][23], J-Orchestra ensures safety while imposing a minimum amount of restrictions on the placement of objects in a pure Java application. The programmer does not need to worry about whether remote objects can ever get passed to unmodifiable code. Additionally, objects can refer to system objects through an indirection from everywhere on the network. If they need to ever pass such references to code that expects direct access, a direct reference will be produced at run-time.

2. It is interesting to compare the requirements of adding indirection to those of a garbage collector. A garbage collector needs to be aware of references to objects, even if these references are manipulated entirely through native code in a runtime system. Additionally, in the case of a copying collector, the GC needs to be able to change references handled by native code. Nonetheless, being aware of references and being able to change them is not sufficient in our case: we need full control of all code that manipulates references, since the references may be to objects on a different machine and no direct access may be possible.

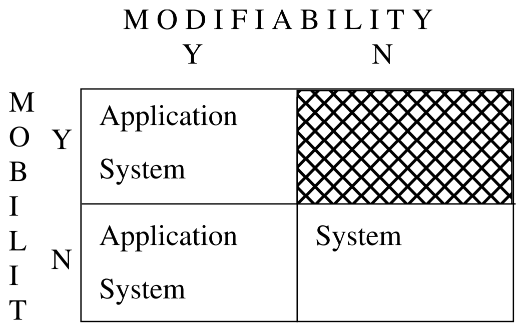


Figure 3: J-Orchestra classification criteria. For simplicity, we assume a "pure Java" application: no unmodifiable application classes exist.

4 SYSTEM VIEW OF J-ORCHESTRA

In this section we describe in detail the two main parts of the approach for dealing with unmodifiable code in J-Orchestra: the classification algorithm and the translation engine.

4.1 Classification Algorithm

The J-Orchestra classification algorithm [22] classifies each class as anchored or mobile and determines anchored class groups. Classes in an anchored group must be placed on the same network site since they access each other directly.

The purpose of the classification algorithm is to determine what rewriting strategy must be used to enable the indirect referencing approach for each class in the partitioned application. I.e., classification is used to tell the rewriter where to inject code, as opposed to having the user specify this information manually. We have already described the first criterion of classification: each class can be either anchored or mobile. The second criterion deals with modifiability properties of a class: each class is either modifiable or not. A class is unmodifiable if its instances are manipulated by native code, e.g., if it has native methods or if its instances may be passed to native methods of other objects. Such dependencies inhibit what kind of bytecode changes can be safely made to the class (sometimes none) without rendering it invalid. Figure 3 presents a diagram that shows all possible combinations of the classification criteria. As can be seen from the diagram, J-Orchestra distinguishes between three categories of classes: *mobile*, *anchored modifiable*, and *anchored unmodifiable*.

As shown in Figure 3, modifiability is a required property to enable mobility in rewritten classes—the unmodifiable mobile quadrant does not have any entries. Classes with native methods need to be anchored. Take, for example, class `java.awt.Component`. This class is anchored unmodifiable because it has a native method `initIDs`. It is directly dependent on native runtime libraries that are platform specific. Furthermore, modifying its bytecode could render it invalid. Consider, for example, changing the name of the class. This would immediately break the linkage with the native libraries: class names are part of a key that matches native method calls in the bytecode to their actual native binary implementations. Classes also need to be anchored even if they do not have native methods, if their objects can be passed to native code. The reason is that native code may be accessing directly the object layout (e.g., reading object fields). In this case, a proxy cannot be used to replace the remote object. For instance, `java.awt.Component` has a method `contains` that takes a parameter of type `java.awt.Point`. Despite the fact that `java.awt.Point` does not have any native dependencies, it will be classified as anchored if it is used along with `java.awt.Component` in the application. As a consequence, `java.awt.Component` and `java.awt.Point` will be put into the same anchored group. Recall that classes within the same anchored group might access each other directly. In the J-Orchestra methodology, we call such classes *co-anchored*.

Conceptually, the classification algorithm has a simple task. It computes for each class *A* and *B* an answer to the question: *can references to objects of class A leak to unmodifiable (native) code of class B*? If the answer is affirma-

```
compute_co-anchored (A) {
  AS := set of all mutable system classes and all array types used in any way in the application
  A := A $\cup$ Superclasses(A) $\cup$ Subclasses(A)
  do {
    AS := AS - A
    AArg := MethodArguments(A)
    AArg := AArg $\cup$ Superclasses(AArg) $\cup$ Subclasses(AArg) $\cup$ Constituents(AArg)
    ArgS := AS $\cap$ AArg
    A := A $\cup$ ArgS
  } while (ArgS $\neq \emptyset$)
  return A
}
```

Figure 4. J-Orchestra algorithm to compute anchored unmodifiable class groups.

tive, *A* cannot be remote to *B*: otherwise the unmodifiable code will try to access objects of class *A* directly (e.g., to read their fields), without being aware that it accesses an indirection (i.e., a proxy) resulting in a run-time error. This criterion determines whether *A* and *B* both belong to the same anchored group. If no constraint of this kind makes class *A* be part of an anchored group, and class *A* itself does not have native code, then it can be mobile. A simplified version of the classification algorithm (not handling all special cases, but showing the main insights) is shown in set pseudo-code form in Figure 4. This algorithm finds the classes that need to be anchored on the same site as any one of the classes of an initial set *A*. The auxiliary set routines used in this algorithm are defined as follows: *Super(Sub)classes(X)* returns the set of all super(sub)classes (other than `java.lang.Object`) of classes in set *X*; *MethodArguments(X)* returns the set of all argument and return types of all methods called in the application on all classes in *X*; *Constituents(X)* returns the set of all constituent types of all array types in *X*. For instance, an array type `T[][]` has constituent types `T[]` and `T`.

There are a few points worth emphasizing about our classification algorithm:

- Not all access to application objects inside native code/anchored classes is prohibited—only access that would break if a proxy was passed instead of the real object. Notably, our classification algorithm ignores Java interfaces. Interface access from unmodifiable code is safe and imposes no restriction. Indeed, anchored unmodifiable code can even refer to mobile objects and to anchored objects in different groups through interfaces. The reason is that an interface does not allow direct access to an object: it does not create a name dependency to a specific class and it cannot be used to access object fields. A proxy can serve just as well as the original object for access through an interface—thus distribution remains transparent for interface accesses.
- The classification algorithm is really a simple type-based heuristic. It computes all types that get passed to anchored code, based on information in the type signatures of native methods and the calling information in the methods (in either application or system classes) that consist of regular Java bytecode. This is a conservative approach as it only provides analysis on a per-type granularity and always assumes the worst: if an instance of class A can be passed to native code, all instances of any subtype of A are considered anchored (we make an explicit exception for `java.lang.Object` or no partitioning would be possible). Despite the conservatism, however, the algorithm is not safe! The unsafety is inherent in the domain: no analysis algorithm, however conservative or sophisticated, can be safe if the unmodifiable code itself cannot be analyzed. The real data flow question we would like to ask is "what objects get accessed directly (i.e., in a way that would break if a proxy were used) by unmodifiable code?" The fully conservative answer is "all objects" since unmodifiable code can perform arbitrary side-effects and is not realistically analyzable, because it is only available in platform-specific binary form. Thus, unmodifiable code in the Java VM could (in theory) be accessing directly *any* object created by the application. For example, when an application creates a `java.awt.Component`, there is no guarantee that other, seemingly unrelated system code will not maintain a reference to this object and later access its fields directly, preventing that code from being on a remote machine. In the face of inherent unsafety, our classification is an engineering approximation. We rely on the rich type interfaces and on the informal conventions used to

code the Java system services. That is, we assume that no object is accessed by unmodifiable code unless it is passed at some point in the program explicitly to such code through a method call. We also assume that no system object is created spontaneously—they all get created under direct application control. These are the primary engineering assumptions of our classification approach. As we discuss in Section 6, the assumptions do not hold, e.g., for `Thread` objects, or implicit objects like `System.in`, `System.out`. We have a special treatment for such objects. We also assume that the system services do not discover type information not present in the type signatures—if a system class has weak type information (e.g., if native code were passed an `Object` reference but the code made assumptions about or dynamically discovered the real type of the object) the classification will not be safe.

Although the assumptions of our classification heuristic are arbitrary, it is important to emphasize again that any different assumptions would be just as arbitrary: safety is impossible to ensure unless either partitioning is disallowed (i.e., a single partition is produced) or platform-specific native code can be analyzed. Since the classification analysis will be heuristic anyway, its success or failure is determined purely by its scalability in practice. We have found our assumptions to be justified empirically, as shown by our success in partitioning large applications with J-Orchestra, without ever encountering a run-time error due to the lack of safety of our classification.

- A more exact (less conservative) classification algorithm would be possible. For example, we could perform a data-flow analysis to determine what objects can leak to unmodifiable code on a per-instance basis. The current classification algorithm, however, fits well the J-Orchestra model of type-based partitioning (recall that the system is semi-automatic and does not assume source code access: the user influences the partitioning by choosing anchorings on a per-class basis). Choosing a more sophisticated algorithm is orthogonal to other aspects of J-Orchestra. In particular, the J-Orchestra rewriting engine (Section 4.2) will remain valid regardless of the analysis used. In practice, J-Orchestra allows its user to override the classification results and explicitly steer the rewrite algorithm.

Our discussion so far covered modifiable and anchored unmodifiable classes, but left out *anchored modifiable classes*. The vast majority of these classes are not put in this category by the classification algorithm. Instead, these classes could be mobile, but are anchored by choice by the user of J-Orchestra. As briefly mentioned earlier, anchoring by choice is useful because it lets the class's code access all co-anchored objects without suffering any indirection penalty. Some of the anchored modifiable classes, however, are automatically classified as such by the classification algorithm. These classes are direct subclasses of anchored unmodifiable classes. An application class `MyComponent` that extends `java.awt.Component` would be an example of such a class. This class does not have any native dependencies of its own but it inherits those dependencies from its super class. As a result, both classes have to be co-anchored on the same site. Since `MyComponent` is an application class, it can support some limited bytecode manipulations. For example, it is possible to change bytecodes of individual methods or add new methods without invalidating the class. At the same time, changing `MyComponent`'s superclass would change the intended original semantics. That is why J-Orchestra must use a different approach to enable remote access to anchored modifiable classes.

4.2 Rewriting Engine

Having introduced J-Orchestra's classification algorithm, we can now describe how the classification information gets used. The J-Orchestra rewriting engine is parameterized with the classification information. The classification category of a class determines the set of transformations it goes through during rewriting. The term "rewriting engine" is a slight misnomer due to the fact that applying binary changes to existing classes is not the only piece of functionality required to enable indirect referencing. In addition to bytecode manipulation,[3] the rewriting engine generates several supporting classes and interfaces in source code form. Subsequently, all the generated classes get compiled into bytecode using a regular Java compiler. We next describe the main ideas of the rewriting approach.

3. We use the BCEL library [6] for bytecode engineering.

The J-Orchestra rewrite first makes sure that all data exchange among potentially remote objects is done through method calls. That is, every time an object reference is used to access fields of a different object and that object is either mobile or in a different anchored group, the corresponding instructions are replaced with a method invocation that will get/set the required data.

For each mobile class, J-Orchestra generates a proxy that assumes the original name of the class. A proxy class has the same method interface as the original class and dynamically delegates to an implementation class. Implementation classes, which get generated by binary-modifying the original class, come in two varieties: *remote* and *local-only*. The difference between the two is that the remote version extends `UnicastRemoteObject` while the local-only does not. Subclasses of `UnicastRemoteObject` can be registered as RMI remote objects, which means that they get passed by-reference over the network. I.e., when used as arguments to a remote call, RMI remote objects do not get copied. A remote reference is created instead and can be used to call methods of the remote object.

Local-only classes are an optimization that allows those clients that are co-located on the same JVM with a given mobile object to access it without the overhead of remote registration. (We discuss the local-only optimization in Section 5.1—for now it can be safely ignored.) The implementation classes implement a generated interface that defines all the methods of the original class and extends `java.rmi.Remote`. Remote execution is accomplished by generating an RMI stub for the remote implementation class. We show below a simplified version of the code generated for a class.

```
//Original mobile class A
class A {
 void foo () { ... }
}

//Proxy for A (generated in source code form)
class A implements java.io.Externalizable {
 //ref at different points can point either to local-only implementation, remote
 //implementation, or RMI stub.
 A__interface ref;
 ...
 void foo () {
  try {
   ref.foo ();
  } catch (RemoteException e) {
    //let the user provide custom failure handling
  }
 }//foo
}//A

//Interface for A (generated in source code form)
interface A__interface extends java.rmi.Remote {
 void foo () throws RemoteException;
}

//Remote implementation (generated in bytecode form by modifying original class A)
class A__remote extends UnicastRemoteObject implements A__interface {
 void foo () throws RemoteException {...}
}

//Local-only version is identical to remote but does not extend UnicastRemoteObject
```

Proxy classes handle several important tasks. One such task is the management of globally unique identifiers. J-Orchestra maintains an "at most one proxy per site" invariant via the help of such globally unique identifiers. Each proxy maintains a unique identifier that it uses to interact with the J-Orchestra runtime system. All proxies imple-

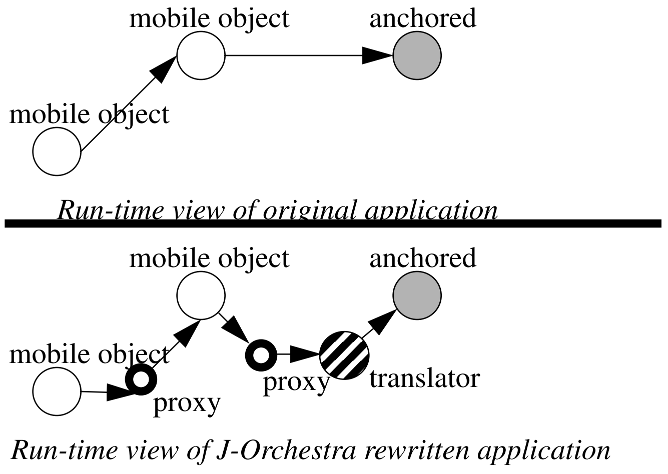


Figure 5: Results of the J-Orchestra rewrite schematically. Proxy objects could point to their targets either locally or over the network.

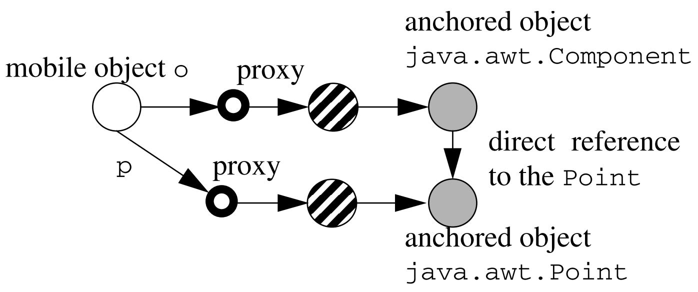


Figure 6: Mobile code refers to anchored objects indirectly (through proxies) but anchored code refers to the same objects directly. Each kind of reference should be derivable from the other.

ment `java.io.Externalizable` to take full control of their own serialization. This enables the support for object mobility: at serialization time proxies can move their implementation objects as specified by a given mobility scenario. Note that proxy classes are generated in source code, thus enabling the sophisticated user to supply handling code for remote errors.

For anchored classes, proxies provide similar functionality but do not assume the names of their original classes. Since both modifiable and unmodifiable anchored classes cannot change their superclass (to `UnicastRemoteObject`), a different mechanism is required to enable remote execution. An extra level of indirection is added through special purpose classes called *translators*. Translators implement remote interfaces and their purpose is to make anchored classes look like mobile classes as far as the rest of the J-Orchestra rewrite is concerned. Regular proxies, as well as remote and local-only implementation versions are created for translators, exactly like for mobile classes. The code generator puts anchored proxies, interfaces and translators into a special package starting with the prefix `remotecapable`. Since it is impossible to add classes to system packages, this approach works uniformly for all anchored classes. Figure 5 shows schematically what an object graph looks like during execution of both the original and the J-Orchestra rewritten code. The two levels of indirection introduced by J-Orchestra for anchored classes can be seen. Note that proxies may also refer to their targets indirectly (through RMI stubs) if these targets are on a remote machine.

In addition to giving anchored classes a "remote" identity, translators perform one of the most important functions of the J-Orchestra rewrite: the dynamic translation of direct references into indirect (through proxy) and vice versa, as these references get passed between anchored and mobile code. Consider what happens when references to anchored objects are passed from mobile code (or anchored modifiable code as we will see in the next section) to anchored code. For instance, in Figure 6, a mobile application object `o` holds a reference `p` to an object of type `java.awt.Point`. Object `o` can pass reference `p` as an argument to the method `contains` of a `java.awt.Component` object. The problem is that the reference `p` in mobile code is really a reference *to a proxy* for the `java.awt.Point` but the `contains` method cannot be rewritten and, thus, expects a direct reference to a `java.awt.Point` (for instance, so it can assign it or compare it with a different reference). In general, the two kinds of references should be implicitly convertible to each other at run-time, depending on what kind is expected by the code currently being run.

It is worth noting that past systems that follow a similar rewrite as J-Orchestra [10][17][20][21][23] do not offer a translation mechanism. Thus, the partitioned application is safe only if objects passed to unmodifiable (system) code are guaranteed to always be on the same site as that code. This is a big burden to put on the user, especially without analysis tools, like the J-Orchestra classification tool. With the J-Orchestra classification and translation, no object will ever be accessed directly if it can possibly be remote. (See Section 6 for some limitations.)

Translation takes place when a method is called on an anchored object. The translator implementation of the method "unwraps" all method parameters (i.e., converts them from indirect to direct) and "wraps" all results (i.e., converts them from direct to indirect). Since all data exchange between mobile code and anchored code happens through method calls (which go through a translator) we can be certain that references are always of the correct kind. For a code example, consider invoking (from a mobile object) methods `foo` and `bar` in an anchored class `C` passing it a parameter of type `P`. Classes `C` and `P` are co-anchored on the same site. The original class `C` and its generated translator are shown below (slightly simplified):

```
//original anchored class C
class C {
 void foo (P p) {...}
 P bar () { return new P(); }
}

//translator for class C
package remotecapable;
class C__translator extends UnicastRemoteObject implements C__interface {
 C originalClass;
 ...
 void foo (remotecapable.P p) throws RemoteException {
  originalClass.foo ((P) Runtime.unwrap(p));
 }

 remotecapable.P bar() throws RemoteException {
  return (remotecapable.P)Runtime.wrap(originalClass.bar());
 }
}
```

4.3 Call-Site Wrapping for Anchored Modifiable Code

In the previous section we presented the dynamic conversion of references when calls are made to methods of anchored objects by mobile objects. Nevertheless, wrapping and unwrapping need to also take place when (modifiable) anchored (usually by-choice) objects call other anchored objects that are in a different anchored group. This case is more complex, but handling it is valuable as it is the only way to enable anchoring by choice. This section explains in detail the wrapping mechanism for anchored modifiable objects.

Anchored and mobile classes present an interesting dichotomy. Anchored objects call methods of all of their co-anchored objects directly without any overhead. Accesses from anchored objects to anchored objects of a different anchored group, on the other hand, result in significant overhead (see Section 5) for every method call (because of proxy and translator indirection) and field reference (because direct field references are rewritten to go through method calls). Mobile objects suffer a slightly lower overhead for indirection: calling a method of a mobile object, irrespective of the location of the caller, always results in a single indirection overhead (for the proxy). At the same time, mobile objects can move at will to exploit locality. The result is that if objects of a modifiable class tend to be accessed mostly locally and only rarely remotely, it can be advantageous to anchor this class by choice. In this way, no indirection overhead is incurred for accesses to methods and fields of co-anchored objects. An anchored modifiable class is still remotely accessible (like all classes in a J-Orchestra-rewritten application) but proxies are only used for true remote access.

From a practical standpoint, anchoring by choice is invaluable. It usually allows an application to execute with no slowdown, except for calls that are truly remote. Anchoring by choice is particularly successful when most of the processing in an application occurs on one network site and only some resources (e.g., graphics, sound, keyboard input) are accessed remotely.

Translators of anchored classes, as discussed in the previous section, are the only avenue for data exchange between mobile objects and anchored objects. Translators are a simple way to perform the wrapping/unwrapping operation because there is no need to analyze and modify the bytecode of the caller: the call is just indirected to go through the translator, which always performs the necessary translations. This approach is sufficient, as long as all the control flow (i.e., the method calls) happens *from* the outside *to* the anchored group but an anchored object never calls methods of objects outside its group. This is the case for pure Java applications consisting of only mobile and anchored unmodifiable (i.e., system) objects. In this case, system code is unaware of application objects and can only call their methods through superclasses or interfaces, in which case no wrapping/unwrapping is required. When anchored modifiable classes are introduced, however, the control-flow patterns become more complex. Anchored modifiable code is regular application code, and thus can call methods in any other application object. Thus, one anchored modifiable object can well be calling an anchored modifiable object in a different anchored group, which may be remote.

Dynamic wrapping/unwrapping needs to take place in this case. The problem is that an anchored modifiable object has direct references to all its co-anchored objects, but may need to pass those direct references to objects outside the anchored group (either mobile or anchored). For instance, imagine a scenario with co-anchored classes `A` and `B`, and class `C` anchored on a different site. The original application code may look like the following:

```
class A {
 B b;
 C c;
 void baz () {
  c.foo (b);
  B b = c.bar ();
 }
}

class B {...}

class C {
 void foo (B b) {...}
 B bar () {...}
}
```

If we were to perform a straightforward rewrite of class `A` to refer to `B` directly but to `C` by proxy we would get:

```
class A {
 B b;
 remotecapable.C c;
 void baz  () {
  c.foo (b);      //incorrectly passing a direct reference to B!
  B b = c.bar(); //incorrectly returning an indirect ref. to B!
 }
}
```

```
//proxy for class C
package remotecapable;
class C {
 ...
 void foo (remotecapable.B b) {...}
 remotecapable.B bar () {...}
}
```

As indicated by the comments in the code, this rewrite would result in erroneous bytecodes: direct references are passed to code that expects an indirection and vice versa. There are two places where a fix could be applied: either at the call site (e.g., the code in class `A` that calls `c.bar()`) or at the indirection site (i.e., at the proxy `C`, or at some other intermediate object, analogous to the translators we saw in the previous section). The translators of the previous section do the wrapping/unwrapping at the indirection site. Unfortunately this solution is not applicable here. If we were to do the wrapping/unwrapping inside the proxy, the proxy for `C` would look like:

```
// This is imaginary code! Irrelevant details (e.g., exception handling) omitted
class C {
 C__interface ref;
 ...
 // used when caller is outside B's anchored group
 void foo (remotecapable.B b) {
  ref.foo ((B) Runtime.unwrap(b));
 }
 // used when caller is in B's anchored group
 void foo (B b) {
  ref.foo((remotecapable.B) Runtime.wrap(b));
 }
 // used when caller is outside B's anchored group
 remotecapable.B bar() {
  return ref.bar();
 }
 // used when caller is in B's anchored group
 B bar() {
  return ((B) Runtime.unwrap(ref.bar());
 }
}
```

Unfortunately, the last two methods differ only in their return type, thus overloading cannot be used to resolve a call to `bar`. This is why a call-site rewrite is required. Since J-Orchestra operates at the bytecode level, this action is not trivial. We need to analyze the bytecode, reconstruct argument types, see if a conversion is necessary, and insert code to wrap and unwrap objects. The resulting code for our example class `A` is shown below (in source code form, for ease of exposition).

```
class A {
 B b;
 remotecapable.C c;
 void baz  () {
  c.foo ((remotecapable.B)Runtime.wrap (b)); //wrap b in the call to foo
  B b = (B) Runtime.unwrap (c.bar());        //unwrap b after the call to bar
 }
}
```

A special case of the above problem is self-reference. An object always refers to itself (`this`) directly. If it attempts to pass such references outside its anchored group (or, in the case of a mobile object, to any other object) the reference should be wrapped.

5 RUN-TIME PERFORMANCE

This section examines issues of run-time performance of J-Orchestra. Although anchoring by choice can practically eliminate the indirection overheads of the J-Orchestra rewrite, it is worth examining how high these overheads can be in the worst case. Section 5.1 presents measurements of these overheads and details the local-only optimization employed in J-Orchestra. In order to get a good placement of anchored groups on network sites, J-Orchestra offers a profiling tool for estimating the data exchange among anchored groups and heuristically deriving a good placement. Additionally, object mobility can yield run-time benefits. Section 5.2 discusses the J-Orchestra profiler and object mobility.

5.1 Indirection Overheads and Optimization

5.1.1 Indirection Overheads

The most significant overheads of the J-Orchestra rewrite are one level of indirection for each method call to a different application object, two levels of indirection for each method call to an anchored system object, and one extra method call for every direct access to another object's fields. The J-Orchestra rewrite keeps overheads as low as possible. For instance, for an application object created and used only locally, the overhead is only one interface call for every virtual call, because proxy objects refer directly to the target object and not through RMI. Interface calls are not expensive in modern JVMs (only about as much as virtual calls [1]) but the overall slowdown can be significant.

The overall impact of the indirection overhead on an application depends on how much work the application's methods perform per method call. A simple experiment puts the costs in perspective. Table 1 shows the overhead of adding an extra interface indirection per virtual method call for a simple benchmark program. The overall overhead rises from 17% (when a method performs 10 multiplications, 10 increment, and 10 test operations) to 35% (when the method only performs 2 of these operations).

Table 1. J-Orchestra worst-case indirection overhead as a function of average work per method call (a billion calls total)

Work (multiply, increment, test)	Original Time	Rewritten Time	Overhead
2	35.17s	47.52s	35%
4	42.06s	51.30s	22%
10	62.5s	73.32s	17%

Penalizing programs that have small methods is against good object-oriented design, however. Furthermore, the above numbers do not include the extra cost of accessing anchored objects and fields of other objects indirectly (although these costs are secondary). To get an idea of the total overhead for an actual application, we measured the slowdown of the J-Orchestra rewrite using J-Orchestra itself as input. That is, we used J-Orchestra to translate the main loop of the J-Orchestra rewriter, consisting of 41 class files totalling 192KB. Thus, the rewritten version of the J-Orchestra rewriter (as well as all system classes it accesses) became remote-capable but still consisted of a single partition. In local execution, the rewritten version was about 37% slower (see Table 2 later). Although a 37% slowdown of local processing can be acceptable for some applications, for many others it is too high. Recall, however, that this would be the overhead of the J-Orchestra rewrite for a partitioning where all application objects were mobile. Anchoring by choice all but a few mobile classes completely eliminates this overhead.

5.1.2 Local-Only Optimization

Recall that remote objects extend the RMI class `UnicastRemoteObject` to enable remote execution. The constructor of `UnicastRemoteObject` exports the remote object to the RMI run-time. This is an intensive process that significantly slows down the overall object creation. J-Orchestra tries to avoid this slowdown by employing lazy remote object creation for all the objects that might never be invoked remotely. If a proxy constructor determines that the object it wraps is to be created on the local machine, then the creation process does not go through the object factory. Instead, a *local-only* version of the remote object is created directly. A local-only object is isomorphic to a remote one but with a different name and without inheriting from `UnicastRemoteObject`. A proxy continues to point to such a local-only object until the application attempts to use the proxy in a remote method call. In that case, the proxy converts its local-only object to a remote one using a special conversion constructor. This constructor reassigns every member field from the local-only object to the remote one. All static fields are kept in the remote version of the object to avoid data inconsistencies.

Although this optimization may at first seem RMI-specific, in fact it is not. Every middleware mechanism suffers significant overhead for registering remotely accessible objects. Lazy remote object creation ensures that the overhead is not suffered until it is absolutely necessary. In the case of RMI, our experiments show that the creation of a remotely accessible object is over 200 times more expensive than a single constructor invocation. In contrast, the extra cost of converting a local-only object into a remotely accessible one is about the same as a few variable assignments in Java. Therefore, it makes sense to optimistically assume that objects are created only for local use, until they are actually passed to a remote site. Considering that a well-partitioned application will only move few objects over the network, the optimization is likely to be valuable.

The impact of speeding up object creation is significant in terms of total application execution time. We measured the effects using the J-Orchestra code itself as a benchmark. The result is shown below (Table 2). The measurements are on the full J-Orchestra rewrite: all objects are made remote-capable, although they are executed on a single machine. 767 objects were constructed during this execution. The overhead for the version of J-Orchestra that eagerly constructs all objects to be remote-capable is 58%, while the same overhead when the objects are created for local use is less than 38% (an overall speedup of 1.15, or 15%).

Table 2. Effect of lazy remote object creation (local-only objects) and J-Orchestra indirection

Original time	Indirect lazy	Overhead	Indirect non-lazy	Overhead
6.63s	9.11s	37.4%	10.48s	58.1%

5.2 Optimizing Performance

Partitioned applications need to be carefully tuned to get reasonable distributed performance. J-Orchestra allows tuning of the partitioned application in two ways: through the use of mobility policies and by profiling application runs to determine a good placement for anchored groups.

Object mobility can significantly affect the performance of a distributed application. Mobile objects can exploit application locality and eliminate the need for network communication. Mobility is the only mechanism in J-Orchestra that enables per-instance, instead of per-class treatment. That is, two objects of the same mobile class can behave entirely differently at run-time based on their uses (i.e., to which methods they are passed as parameters, etc.). J-Orchestra supports synchronous object mobility: objects move in response to method calls. There are three supported types of moving scenarios: moving a parameter of a remote method call to the site of the call, moving the return value of a remote method call to the site of the caller, and moving "this" object to the site of the call. In terms of design, our object migration policies are similar to what is commonly found in the mobile objects literature [4][13]. In terms of mechanisms, our implementation is similar to that of JavaParty [17].

Another factor affecting run-time performance is the placement of anchored classes. To help the user in determining a good placement, J-Orchestra offers an off-line profiling tool. Off-line profiling consists of examining the behavior of an application under some sample input. The profiler reports to the user statistics about the data exchange among different anchored groups. Integrated with the profiler is an implementation of a clustering algorithm that, given some locations for a few anchored groups and the data exchange patterns among all anchored groups, determines a good placement for all anchored classes. The clustering algorithm is really a greedy heuristic—it co-anchors new groups to the site for which they currently display greater affinity. Even though this is not an optimal policy (exhaustive search of all combinations would be necessary for guaranteeing optimality, since clustering is NP-hard in general) it is good enough in practice. The clustering algorithm's output is strictly advisory: the user can override it during partitioning.

In terms of implementation, the J-Orchestra profiler has evolved through several incarnations. The first profiler worked by instrumenting the Java VM through the JVMPI and JVMDI (Java Virtual Machine Profiling/Debugging Interface) binary interfaces. We found the overheads of this approach to be very high, even for recent VMs that enable compiled execution under debug mode. The reason is the "impedance mismatch" between the profiling code (which is written in C++ and compiled into a dynamic library that instruments the VM) and the Java object layout. Either the C++ code needs to use JNI to access object fields, or the C++ code needs to call a Java library that will use reflection to access fields. We have found both approaches to be much more expensive (15x) than using bytecode engineering to inject our own profiling code in the application. The profiler rewrite is isomorphic to the J-Orchestra rewrite, except that no distribution is supported—proxies keep track of the amount of data passed instead.

An important issue with profiling concerns the use of off-line vs. on-line profiling. Several systems with goals similar to ours (e.g., Coign [12] and AIDE [16]) use on-line profiling in order to dynamically discover properties of the application and possibly alter partitioning decisions on-the-fly. So far, we have not explored an on-line approach in J-Orchestra, because of its overheads for regular application execution. These overheads can be expected to be higher in the case of J-Orchestra than in other systems that explicitly control the runtime environment. The reason is that we have to do on-line profiling without modifying the runtime but just by injecting the right code in the application using bytecode rewriting. (Recall that keeping an unmodified runtime environment is one of the main goals of application partitioning, for portability and ease of adoption reasons.) Nevertheless, without low-level control, it is hard to keep such overhead to a minimum. Sampling techniques can alleviate the overhead (at the expense of some accuracy) but not eliminate it: some sampling logic has to be executed in each method call, for instance. We hope to explore the on-line profiling direction in the future. Note that on-line repartitioning would only be applicable for classes anchored by-choice—classes anchored due to native dependencies cannot be moved after the application has started executing.

6 JAVA LANGUAGE FEATURES AND LIMITATIONS

J-Orchestra needs to handle many Java language features specially in order to enable partitioning of unsuspecting applications. Features with special handling include inheritance, static methods and fields, object creation, arrays, object identity, synchronization, reflection, method access modifiers, garbage collection, and inner classes. We do not describe the low-level specifics of dealing with every language feature here, as they are mostly straightforward—the interested reader should consult our previous paper [22]. Nevertheless, it is interesting to survey some of the limitations of the system, both in its safety guarantees and in offering a complete emulation of a single Java VM over a distributed environment.

Unsafety. As mentioned in Section 4.1, there will always be unsafeties in the J-Orchestra classification, but these are inherent in the domain of automatic partitioning and not specific to J-Orchestra. No partitioning algorithm will ever be safe without assumptions about (or analysis of) the platform-specific binary code in the system classes. System code can always behave badly, keeping aliases to any object that gets created and accessing its fields directly, so that no proxy can be used instead. Additionally, there are several objects that are only created and used implicitly by native code, without their presence ever becoming explicit at the level of the interface between system and application code. For example, every site is implicitly associated with at least one thread object. If the application semantics is sensitive to all threads being created on the same machine, then the execution of the partitioned application will not be indentical to the original one. Similarly, every JVM offers pre-defined objects like `Sys-`

`tem.in`, `System.out`, `System.err`, `System.properties` and `System.exit`. The behavior of an application using these stateful implicit objects will not be the same on a single JVM and on multiple ones. Indeed, it is not even clear that there is a single correct behavior for the partitioned application—different policies may be appropriate for different scenarios. For example, when one of the partitions writes something to the standard output stream, should the results be visible only on the network site of the partition, all the network sites, or one specially designated network site that handles I/O? If one of the partitions makes a call to `System.exit`, should only the JVM that runs that partition exit or the request should be applied to all the remaining network sites? J-Orchestra allows defining these policies on a per-application basis.

Conservative classification. The J-Orchestra classification is quite conservative. For instance, it is perfectly reasonable to want to partition an application so that two different sites manipulate instances of a certain anchored unmodifiable class. For example, two different machines may need to use graphical windows, but without the windows manipulated by code on one machine ever leaking to code on the other. J-Orchestra cannot tell this automatically since it has to assume the worst about all references that potentially leak to native code. Thus, partitionings that require objects of the same anchored unmodifiable class to be created on two different sites are not safe according to the J-Orchestra classification. This is a problem that is commonly encountered in practice. In those cases, the user needs to manually override the J-Orchestra classification and assert that the classes can safely exist on two sites. Everything else proceeds as usual: the translation wrapping/unwrapping technique is still necessary, as it enables indirect access to anchored unmodifiable objects (e.g., so that code on site *A* can draw on a window of site *B*, as long as it never passes the remote reference to local unmodifiable code).

Reflection and dynamic loading. Reflection can render the J-Orchestra translation incorrect. For instance, an application class may get an `Object` reference, query it to determine its actual type, and fail if the type is a proxy. Nevertheless, the common case of reflection that is used only to invoke methods of an object is compatible with the J-Orchestra rewrite—the corresponding method will be invoked on the proxy object. Similar observations hold regarding dynamic class loading. J-Orchestra is meant for use in cases where the entire application is available and gets analyzed, so that the J-Orchestra classification and translation are guaranteed correct. Currently, dynamically loading code that was not rewritten by J-Orchestra may fail because the code may try to access remote data directly. Nevertheless, one can imagine a loader installed by J-Orchestra that takes care of rewriting any dynamically loaded classes before they are used. Essentially, this would implement the entire J-Orchestra translation at load time. Unfortunately, classification cannot be performed at load time. The J-Orchestra classification is a whole-program analysis and cannot be done incrementally: unmodifiable classes may be loaded and anchored on some nodes before loading another class makes apparent that the previous anchorings are inconsistent.

Inherited limitations. J-Orchestra inherits some limitations from its underlying middleware—Java RMI. These limitations are better addressed uniformly at the middleware level than by J-Orchestra. One limitation has to do with efficiency. Although RMI efficiency has improved in JDK 1.4, RMI still remains a heavyweight protocol. Another limitation concerns distributed garbage collection. J-Orchestra relies on the RMI distributed reference counting mechanism for garbage collection. This means that cyclic garbage, where the cycle traverses the network, will never be collected.

Additionally, J-Orchestra does not explicitly address security and administrative domain issues—indeed the J-Orchestra rewrite even weakens the protections of some methods, e.g., to make them accessible through an interface. We assume that the user has taken care of security concerns using an orthogonal approach to establish a trusted domain (e.g., a VPN).

7 SCALABILITY OF J-ORCHESTRA

J-Orchestra is a large engineering artifact (already more than 35Kloc) and is still under development. Despite the engineering work of J-Orchestra, however, we saw earlier that there will always be legal Java programs that J-Orchestra will not be able to partition correctly or efficiently. So, in which sense do we claim that J-Orchestra is scalable and useful?

The answer is that J-Orchestra can handle a large subset of Java and, thus, can correctly partition a large class of realistic applications. Among these, there are well-defined cases where J-Orchestra will yield benefit. The J-Orchestra motto is "*code near resource*". J-Orchestra yields its greatest benefits when the partitioned application needs to use heterogeneous resources (e.g., a processor, a database, a graphical screen, and a sound card) and there are advantages in putting the code near the resource that it controls. For instance, if a graphical representation can be computed from less data than it takes to transfer the entire graphical representation over the network, then J-Orchestra has an advantage.

The main novelty of J-Orchestra is its approach to partitioning applications even in the presence of unmodifiable code. Although we saw that there are limitations, it is important that *a correct partitioning is at all possible for programs of realistic size without intimate knowledge of their internals*! As mentioned earlier, the main problem of past automatic distribution systems has been that references to remote data can leak to code that is unaware of the distribution, thus causing a fatal run-time error. Past systems have put the burden of ensuring correctness on the user. This approach is unscalable. For instance, consider Addistant [23], the closest system to J-Orchestra in design terms. Addistant has no counterpart of the J-Orchestra dynamic wrapping-unwrapping of references. Thus, the user has to have excellent knowledge of the internals of the application being rewritten. This is the only way to ensure that application objects never pass references to remote system objects to system code unaware of the distribution. If even a single remote reference leaks to code expecting direct access, a run-time error will occur.

7.1 Case Study: Distributing JBits

To demonstrate the scalability of J-Orchestra, we used it to partition a commercial, third-party, binary-only application. The application is the JBits FPGA simulator by Xilinx [9]. JBits is a true industrial application—a web search reveals many cases of industrial use. The partitioning was requested by one of our colleagues who is an active user of the software. The desired partitioning scenario is one where all the user interaction through a GUI is performed on one machine (a home machine with a slow link, possibly) while the simulation computations are performed on a central server.

The JBits GUI (see [9] for a picture of an older version) is very rich with a graphical area presenting the results of the simulation cells, as well as multiple smaller areas presenting the simulated components. The GUI allows connecting to various hardware boards and simulators and depicting them in a graphical form. It also allows stepping through a simulation offering multiple views of a hardware board, each of which can be zoomed in and out, scrolled, etc. The JBits GUI is quite representative of CAD tools in general.

JBits was given to us as a bytecode-only application. The installed distribution (with only Java binary code counted) consists of *1,920 application classes* that have a combined size of *7,577 KBytes*. These application classes use a large number of system classes—a significant part of the Java system libraries. We have no understanding of the internals of JBits, and only limited understanding of its user-level functionality.

JBits is a good candidate for automatic partitioning because its locality patterns are well defined and the "split" is conceptually quite simple: all graphics-related code has to reside on a single machine, while most of the rest of the code resides on a different machine. At the implementation level, however, the conceptual simplicity breaks down as objects can be referenced from all different parts of the code.

7.1.1 Partitioning Specifics

To obtain an efficient partitioned version of JBits, we needed to use many of the J-Orchestra features. Specifically, we anchored most objects by choice, we experimented with different static placements, and we experimented with migration policies for objects. J-Orchestra features simple profiling tools to help with the trial-and-error partitioning task by showing the number of remote calls and data exchange patterns.

For our final partitioning, the vast majority (about 1,800) of the application's classes are anchored by choice on the server. Recall that the advantage of this arrangement is that co-anchored objects can access each other directly, thus imposing no overhead on the application's execution. This is particularly important in this case, as the main functionality of JBits is the simulation, which is compute-intensive. With the anchoring by choice, the simulation incurs *no measurable overhead* in its execution time.

259 classes are always anchored on the client (i.e., GUI) site. Of these, 144 are JBits application classes and the rest are classes from the Java system's graphical packages (AWT and Swing). The rest of the classes are anchored on the server site. We discuss a variation where we allow mobile objects in Section 7.1.3. The total experimentation time before we arrived at our "good" partitioning was in the order of 1-2 days. This is certainly much less than the effort a developer would need to expend to change an application with about 2,000 classes, more than 200 of which need to be modified to be accessed remotely.

It is worth noting that for practical scalability reasons (disk space and rewrite time), we performed a special-case optimization that is not yet part of the J-Orchestra arsenal. We implemented a domain-specific heuristic that determines what application classes in a Swing/AWT application will never interact with the GUI aspect of the application (e.g., will never be passed to a GUI class as call-backs).[4] Then J-Orchestra can avoid creating proxies and translators for these server-anchored classes, since they will never be called remotely. This optimization does not enhance the conceptual scalability of the J-Orchestra approach (useless code would be created but never loaded) but cuts down rewrite time from hours to minutes.

Knowing the design principles of the Swing/AWT libraries allowed us to reduce the set of rewritten classes even further. The Swing/AWT event model distinguishes between event producers and event listeners. Event objects get passed from event producers to events listeners. Event producers keep lists of event listeners that can be updated at any time. Event listeners tend to use event objects as read-only objects since the programming model makes it very difficult to determine in what order event listeners receive events. A read-only object can be safely passed by-copy to a remote call—there is no danger of it being modified through aliases. This allowed us to use a special rewrite on all the event objects. We simply made all the event objects serializable by making them implement `java.io.Serializable` and adding a default no arguments constructor if it is not already present. (Both of these modifications were performed by bytecode rewrites.) This not only provides a nice optimization but also further reduces the number of classes that need to be considered for all the standard J-Orchestra functions. Additionally, knowing only the JBits execution from the user perspective, we speculated that the integer arrays transferred from the server towards the GUI part of JBits could safely be passed by-copy. These arrays turned out to never be modified at the GUI part of the application. Passing immutable objects by-copy is a standard optimization for J-Orchestra. Instances of well-known immutable classes (e.g., `java.lang.String`, `java.awt.Color`) are always passed by-copy.

7.1.2 Partitioning Benefits

To demonstrate the benefits of the J-Orchestra partitioning, we analyze the partitioned application behavior in comparison with using the X window system to remotely control and monitor the application. Overall, J-Orchestra showed significant benefits. We discuss them and analyze different contributing factors below. Since JBits is an interactive application and we could not modify what it does, we mainly got measurements of the data transferred and not the total time taken to update the screen (i.e., we measured bandwidth consumption but not latency, except subjectively). Thus, this number would not change in a different measurement environment. For reference, however, our environment consisted of a SunBlade 1000 (two UltraSparc III 750MHz processors and 2GB of RAM) and a Pentium III, 600MHz laptop connected through 10Mbps ethernet.

4. The heuristic is type-based—it would not be safe if type information were completely obscured in the Swing API (e.g., if a method accepted an `Object` type and used reflection to determine if the object is suitable). First, we compute a set of all the application classes that are subclasses of system classes with package names starting with `java.awt.*` or `javax.swing.*`. Then we compute the set of classes that reference or are referenced by any of the classes in the first set. The union of those two sets consists of the classes that have to be considered for rewriting. The rest of application classes can be considered anchored by choice and we simply omit generating any of the supporting classes for them.

Local GUI operations. The overall responsiveness of the J-Orchestra partitioned application is much better than using a remote X-Window display. From the perspective of the interactive user, the latency of GUI operations is very short in the J-Orchestra partitioned version. Indeed, many GUI operations require no network transfer. Thus, any real usage scenario can be made to show the J-Orchestra partitioned application perform arbitrarily better than a remote X-Window display. For instance:

- JBits has multiple views of the simulation results (“State View”, “Power View”, “Core View”, and “Routing Density View”). Switching between views is a completely local operation in the J-Orchestra partitioned version—no network transfers are caused. In contrast, the X window system needs to constantly refresh the graphics on screen. For cycling through all four views and returning to the original, 3.4MBytes needed to be transferred over the network under the X window system.
- JBits has deep drop-down menus (e.g., a 4-level deep menu under “Board->Connect”). Navigating these drop-down menus is again a local operation for the J-Orchestra partitioned application, but not for remote access with the X window system. For interactively navigating 4 levels of drop-down menus, X transferred 1.8MBytes of data.
- GUI operations like resizing the virtual display, scrolling the simulated board, or zooming in and out (four of the ten buttons on the JBits main toolbar are for resizing operations) do not result in network traffic with the partitioned version of JBits. In contrast, the remote X display produces heavy network traffic for such operations. With our example board, one action each of zooming-in completely and zooming-out results in 3.5MBytes of data transferred. Scrolling left once and down once produces about 2MBytes of data over the network with X, but no network traffic with the J-Orchestra partitioned version. Continuous scrolling over a 10Mbps link is unusably slow with the X window system. Clearly, a dial-up modem link is too slow for remote interactive use of JBits with X and even a DSL connection is quite slow.

Although there could be ways (e.g., compression, or a more efficient protocol) to reduce the amount of data transferred by X, the important point is that some data transfer needs to take place anyway. In contrast, J-Orchestra only needs to transfer a data object to the remote site, and all GUI operations presenting the same data can then be performed locally.

Data transferred for board updates. Even for a regular board redraw, where J-Orchestra needs to transfer data over the network, less data get transferred than in the X version. Specifically, the J-Orchestra partitioned application needs to transfer about 1.28MB of data in total for a complete simulation step (with the middle-of-three in complexity simulator provided with JBits) including a redraw of the view. The X window system transfers about 1.68MBytes for the same task. Furthermore, J-Orchestra transfers these data using five times fewer total TCP segments, suggesting that for a network where latency is the bottleneck, X would be even less efficient.

Although the amounts of data transferred for a board update can certainly be reduced by compression (or a more efficient representation) the same argument applies both to X and to Java RMI. Currently J-Orchestra does not in any way try to optimize communication. The only benefits obtained are because of moving the code close to the resource it manages.

7.1.3 More Experiments and Discussion

In the previous discussion we did not discuss the effects of mobility. In fact, very few of the mobile objects in our partitioning actually need to move in an interesting way. The one exception is JBits View Adaptor objects (instances of four `*ViewAdaptor` classes). View adaptors seem to be logical representations of visual components and they also handle different kinds of user events such as mouse movements. During our profiling we noticed that such objects are used both on the server and the client partition and in fact can be seen as carriers of data among the two partitions. Thus, no static placement of all view adaptor objects is optimal—the objects need to move to exploit locality. We specified a mobility policy that originally creates view adaptors on the client site, moves them to the server site when they need to be updated, and then moves them back to the client site. We discuss this case next.

Mobility for reduction of remote calls and latency. Our first observation from measuring the effect of mobility is that it actually results in more data transferred over the network! With mobile view adaptor objects and an otherwise indistinguishable partitioning, J-Orchestra transferred more than 2.59MBytes per simulation step (as opposed to 1.28MBytes without a mobility policy for view adaptor objects). The reason for this is that the mobile objects are quite large (in the order of 300-400KBytes) but only a small part of their data are read/written. In terms of bytes transferred it would make sense to leave these objects on one site and send them their method parameters remotely. Nevertheless, mobility results in a decrease in the total number of remote calls: 386 instead of 484 with a static partitioning for starting JBits, loading a file and performing 5 simulation steps. Thus, the partitioned version of JBits with mobile objects may perform better for fast networks where latency is the bottleneck instead of bandwidth.

Mobility for tolerance of bad partitioning. An important benefit of mobility is that it provides tolerance to bad partitionings. When it is not clear whether objects of a certain type are referenced more on the client or the server site, it is good to make them mobile. A good mobility scenario in this case is to move an object as soon as it receives a method call from a remote site or is passed as a parameter to a remote site. As a simple example, the very first partitioning of JBits that we attempted was grossly sub-optimal. This resulted in an enormous number of remote calls (over 200,000) for a few simulation steps. We then tried a partitioning where 49 of the application classes produced mobile objects. The 49 classes were the ones for which it was not clear whether they should be placed on the client or the server. By making the objects mobile, the number of remote calls dropped to 183 for the same execution.

7.2 Other examples

JBits is not the only example of an application partitioned with J-Orchestra, but it is certainly the largest, as well as being commercially available and bytecode-only. (Several example applications are available on the J-Orchestra web site for tutorial purposes.) Some of the most representative other applications we have partitioned and we use to demonstrate J-Orchestra include:

- the Java Speech API demo mentioned in Section 2. Speech is produced on one machine while the application GUI is running on a handheld (IPaq machine). In general, Java sound APIs can easily be separated from an application's logic using J-Orchestra.
- JShell: a third-party command shell for Java. The command parsing is done on one machine, while the commands are executed on another.
- PowerPoint controller: we have written a small Java GUI application that controls MS PowerPoint through its COM interface. We partitioned the GUI of this application from its back-end. We run the GUI on a IPaq PDA with a wireless card and use it to control a Windows laptop. We have given multiple presentations using this tool.
- A remote load monitoring application: machine load statistics are collected and filtered locally with all the results forwarded to a handheld (IPaq) machine over a wireless connection and displayed graphically. The original application was written to run on a single Windows machine.
- The new version (complete re-engineering) of Kimura [15]: a system for future computing environments research, managing multiple "working contexts" (virtual desktops on multiple machines) and the interactions among them. Although Kimura is a large application, it was rewritten with the explicit purpose to develop a centralized version that will later become distributed using J-Orchestra. Thus, it showcases a very different use of J-Orchestra than JBits does.

8 RELATED WORK

Much research work is closely related to J-Orchestra, either in terms of goals or in terms of methodologies. We discuss some of this work next.

Several recent systems other than J-Orchestra can also be classified as automatic partitioning tools. In the Java world, the closest approaches are the Addistant [23] and Pangaea [20] systems. The Coign system [12] has promoted the idea of automatic partitioning for applications based on COM components. All three systems do not address the problem of distribution in the presence of unmodifiable code.

Coign is the only one of these systems to have a claim at scalability, but the applications partitioned by Coign consist of independent components to begin with. Coign does not address the hard problems of application partitioning, which have to do with pointers and aliasing: components cannot share data through memory pointers. Such components are deemed non-distributable and are located on the same machine. Practical experience with Coign [12] showed that this is a severe limitation for the only real-world application included in Coign's example set (the Microsoft PhotoDraw program). The overall Coign approach would not be feasible for applications in a general purpose language (like Java, C, C#, or C++) where pointers are prevalent, unless a strict component-based implementation methodology is followed.

The Pangaea system [20][21] has very similar goals to J-Orchestra. Pangaea, however, includes no support for making Java system classes remotely accessible. Thus, Pangaea cannot be used for resource-driven distribution, as most real-world resources (e.g., sound, graphics, file system) are hidden behind system code. Pangaea utilizes interesting static analyses to aid partitioning tasks (e.g., object placement) but these analyses ignore unmodifiable (system) code.

JavaParty [10][17] itself is closely related to J-Orchestra. The similarity is not so evident in the objectives, since JavaParty only aims to support manual partitioning and does not deal with system classes. The implementation techniques used, however, are very similar to J-Orchestra, especially for the newest versions of JavaParty [10]. Similar comments apply to the FarGo [11] and AdJava [8] systems. Notably, however, FarGo has focused on grouping classes together and moving them as a group. FarGo groups are similar to J-Orchestra anchored groups. In fact, groups of J-Orchestra objects that are all anchored by choice could well move, as long as all objects in the group move. We have not yet investigated such mobile groups, however.

Automatic partitioning is essentially a Distributed Shared Memory (DSM) technique. Nevertheless, automatic partitioning differs from traditional DSMs in several ways. First, automatic partitioning systems like J-Orchestra do not change the runtime system, but only the application. Traditional DSM systems like Munin [5], Orca [3], and, in the Java world, cJVM [1], and Java/DSM [24] use a specialized run-time environment in order to detect access to remote data and ensure data consistency. Also, DSMs have usually focused on parallel applications and require programmer intervention to achieve high-performance. In contrast, automatic partitioning concentrates on resource-driven distribution, which introduces a new set of problems (e.g., the problem of distributing around unmodifiable system code, as discussed). Among distributed shared memory systems, the ones most closely resembling the J-Orchestra approach are object-based DSMs, like Orca [3].

Mobile object systems, like Emerald [4][13] have formed the inspiration for many of the J-Orchestra ideas on object mobility scenarios. The novelty of J-Orchestra is not in the object mobility ideas but in the rewrite that allows them to be applied to an oblivious centralized application.

Both the D [14] and the Doorastha [7] systems allow the user to easily annotate a centralized program to turn it into a distributed application. Although these systems are higher-level than explicit distributed programming, they are significantly lower-level than J-Orchestra. All the burden is shifted to the programmer to specify what semantics is valid for a specific class (e.g., whether objects are mobile, whether they can be passed by-copy, etc.). Programming in this way requires complete understanding of the application behavior and can be error-prone: a slight error in an annotation may cause insidious inconsistency errors.

9 APPLICABILITY AND CONCLUSIONS

As the advent of the Internet has changed the computing landscape, the need for new distributed applications will only keep growing. Accessing remote resources has now become one of the primary motivations for distribution. In this paper we have shown how J-Orchestra allows the partitioning of programs onto multiple machines without programming. Although J-Orchestra allows programmatic control of crucial distribution aspects (e.g., handling errors related to distribution) it neither attempts to change nor facilitates changing the structure of the original application. Thus, J-Orchestra is applicable in cases where the original application has loosely coupled parts, as is commonly the case when multiple resources are controlled.

Although J-Orchestra is certainly not a "naive end-user" tool, it is also not a "distributed systems guru" tool. Its ideal user is the system administrator or third-party programmer who wants to change the code and data locations of an existing application with only a superficial understanding of the inner workings of the application.

We believe that J-Orchestra is a versatile tool that offers practical value and interesting design ideas. J-Orchestra is interesting on the technical front as the first representative of partitioning tools with what we consider important characteristics:

- use of a high-level language runtime, like the Java VM or the Microsoft CLR. Modifications are performed directly at the binary level.
- no change to the runtime required for partitioning.
- provisions for correct execution even in the presence of code unaware of the distribution (e.g., Java system code).

We hope that partitioning tools will become mainstream in the future and that the techniques of J-Orchestra will prove valuable in such efforts.

ACKNOWLEDGMENTS

This work has been supported by the NSF through a CAREER and an ITR award, by the Yamacraw Foundation, and by DARPA/IXO under the PCES program. Equipment was provided by Sun Corporation through an Academic Equipment Grant. We thank Ken Mackenzie for suggesting JBits as an application to partition and showing us how to use the software. Marcus Handte wrote the J-Orchestra GUI and contributed to the code base.

REFERENCES

[1] Bowen Alpern, Anthony Cocchi, Stephen Fink, David Grove, and Derek Lieber, "Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless", in Proc. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[2] Yariv Aridor, Michael Factor, and Avi Teperman, "CJVM: a Single System Image of a JVM on a Cluster", in Proc. *ICPP'99*.

[3] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Ruhl, and M. Frans Kaashoek, "Performance Evaluation of the Orca Shared-Object System", *ACM Trans. on Computer Systems*, 16(1):1-40, February 1998.

[4] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter, "Distribution and Abstract Types in Emerald", in *IEEE Trans. Softw. Eng.*, 13(1):65-76, 1987.

[5] John B. Carter, John K. Bennett, and Willy Zwaenepoel, "Implementation and performance of Munin", *Proc. 13th ACM Symposium on Operating Systems Principles*, pp. 152-164, October 1991.

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

[7] Markus Dahm, "Doorastha—a step towards distribution transparency", *JIT* 2000. See `http://www.inf.fu-berlin.de/~dahm/doorastha/`.

[8] Mohammad M. Fuad and Michael J. Oudshoorn, "AdJava— Automatic Distribution of Java Applications", 25th *Australasian Computer Science Conference (ACSC)*, 2002.

[9] Steven A. Guccione, Delon Levi and Prasanna Sundararajan, "JBits: A Java-based Interface for Reconfigurable Computing", *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999. See also `http://www.xilinx.com/products/jbits/`.

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

[11] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit, "Dynamic Layout of Distributed Applications in FarGo", *Int. Conf. on Softw. Engineering (ICSE)* 1999.

[12] Galen C. Hunt, and Michael L. Scott, “The Coign Automatic Distributed Partitioning System”, *3rd Symposium on Operating System Design and Implementation (OSDI’99)*, pp. 187-200, New Orleans, 1999.

[13] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black, “Fine-Grained Mobility in the Emerald System”, *ACM Trans. on Computer Systems*, 6(1):109-133, February 1988.

[14] Cristina Videira Lopes and Gregor Kiczales, “D: A Language Framework for Distributed Programming”, PARC Technical report, February 97, SPL97-010 P9710047.

[15] Blair MacIntyre, Elizabeth Mynatt, Stephen Voida, Klaus Hansen, Joe Tullio, and Gregory Corso, “Support for multitasking and background awareness using interactive peripheral displays”, *ACM Symposium on User Interface Software and Technology (UIST)*, 2001.

[16] Alan Messer, Ira Greenberg, Philippe Bernadat, Dejan Milojicic, Deqing Chen, T.J. Giuli, Xiaohui Gu, “Towards a Distributed Platform for Resource-Constrained Devices”, *International Conference on Distributed Computing Systems (ICDCS)*, 2002.

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

[18] Robert W. Scheifler, and Jim Gettys, “The X Window System”, *ACM Transactions on Graphics*, 5(2): 79-109, April 1986.

[19] Robert W. Scheifler, “X Window System Protocol, Version 11”, *Network Working Group RFC 1013*, April 1987.

[20] Andre Spiegel, *Automatic Distribution of Object-Oriented Programs*, PhD Thesis. FU Berlin, FB Mathematik und Informatik, December 2002.

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

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

[23] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano, “A Bytecode Translator for Distributed Execution of ‘Legacy’ Java Software”, *European Conference on Object-Oriented Programming (ECOOP)*, Budapest, June 2001.

[24] Weimin Yu, and Alan Cox, “Java/DSM: A Platform for Heterogeneous Computing”, *Concurrency: Practice and Experience*, 9(11):1213-1224, 1997.