Scalable Automatic Partitioning with J-Orchestra

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

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. 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. Additionally, we define the domain of "embarrassingly loosely coupled" applications, whose structure and communication patterns make them easily amenable to automatic distribution.

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. Examples abound. A local database that grows too large needs to be moved to a powerful server and becomes remote from the rest of the application. An application needs to redirect its output to a remote graphical display or to receive input from a remote digital camera. A desktop application needs to execute on a PDA, where it might not find all the referenced APIs and their corresponding hardware resources available locally and will need to access them remotely.

All the aforementioned scenarios give rise to *application partitioning*: the task of splitting up the functionality of a centralized application into distinct entities running across different network sites. To accomplish such partitioning one can modify the source code of the original application 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 programs.

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 re-written 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, mainly because of deployment reasons (it is easy to run a partitioned application on a standard VM, which can be found pre-compiled and installed on a large variety of platforms).

The conceptual difficulty of performing application partitioning in general-purpose languages 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 [11][17][20] and distributed shared memory systems [1][2][4][21]) consists of addressing the problems resulting from inability to analyze and modify all the code under the control flow of the application. Code that cannot be analyzed and modified is usually part of the runtime system (i.e., the Java VM). If such code accesses a remote object, a run-time error will occur since the code is unaware of distribution (e.g., it expects to access fields of a regular object but instead receives a proxy). Prior partitioning systems have ignored the issues arising from native system code and have had limited scalability, as a result. J-Orchestra features a novel rewrite mechanism 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 graphics code on another.

In a previous publication [18] we described J-Orchestra's general partitioning approach and the novelty of its rewriting engine. The current 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 try to identify the general class of applications to which automatic partitioning is applicable. We call these applications *embarrassingly loosely coupled*: they consist of loosely coupled parts that, furthermore, are clearly reflected in the static structure of the application (e.g., in the object types).

2. J-Orchestra Mechanisms

2.1. Technical Overview

The J-Orchestra user interacts with the system using a GUI that lists all application classes and the systems classes they reference. The user creates different "sites" and assigns classes to sites. In the end, J-Orchestra rewrites the application to produce distinct partitions that can be run on separate machines. No source code access or explicit programming is required.

Conceptually, the J-Orchestra rewrite of the application introduces an extra indirection to object references. The standard technique is to convert all direct references to indirect references by adding proxies. This creates an abstraction of shared memory in which 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 1 shows schematically the effects of the indirect referencing approach. This indirect referencing approach has been used in several prior systems [16][17][20].

Adding indirection without changing the JVM entails rewriting the code of the partitioned application. Thus, when the original application would create a new object,

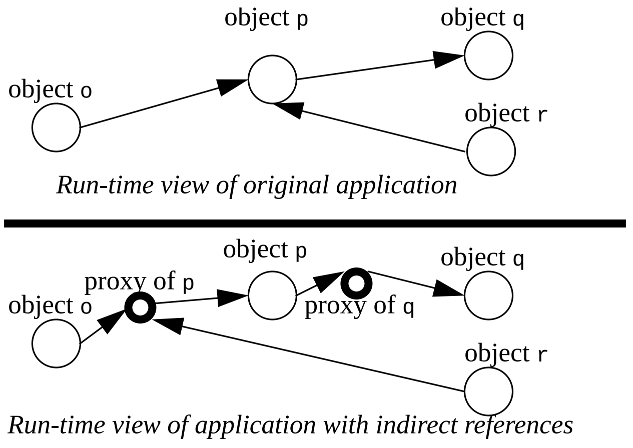
object p
object q
object o
object r
Run-time view of original application
object p
object q
proxy of p
object o
proxy of q
object r
Run-time view of application with indirect references

Figure 1: Indirect referencing schematically. Proxy objects could point to their targets either locally or over the network.

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.

The difficulty of this rewrite approach is that it needs to be applied to *all code that might hold references to remote objects*. 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 likely occur (e.g., because the native code will try to read fields directly from the object). (For simplicity, we assume the application itself does not contain native code—i.e., it is a "pure Java" application.)

We distinguish between two different kinds of classes in J-Orchestra: *anchored* classes and *mobile* classes. Anchored classes create "anchored" objects that will be in a single JVM for their entire lifetime. Mobile classes create objects that can migrate from site to site at run-time.

Anchored classes can be anchored for two reasons. First, the potential of accessing application objects through native code determines whether and where such objects should be anchored. If an object can be accessed by native code running on some machine, the object should be anchored on that machine. Nevertheless, the object can still be accessed from other machines and it is accessed indirectly (through a proxy) by mobile objects even when these happen to be on the same machine. The second reason why a class can be anchored is to reduce the overhead to access its objects from other code on a specific site. We call this *anchoring by choice*. Objects that are anchored by choice can be accessed in local code without any indirection overhead (i.e., as quickly as in the original centralized application). In a typical J-Orchestra partitioning, the vast majority of objects are anchored by choice. We discuss anchoring by choice and its implications on the rewriting algorithm in Section 2.3.

Due to lack of space and previous publication [18], our discussion of J-Orchestra in this paper omits some interesting elements. These include:

- a type-based "classification" heuristic that groups classes whose instances can be accessed by the same native code. Although this heuristic is not sound (native code can potentially access all application objects) in practice it is useful in helping the user decide groupings for classes that should be co-located. The groupings typically reflect distinct resources, e.g., classes that deal with graphics, classes that deal with sound, and classes that deal with files end up in three distinct groups.
- a profiling tool that helps the user determine the coupling (i.e., volume of data exchange) between different parts of the application during test runs.
- optimizations for creating remote objects lazily, i.e., when the object first gets accessed remotely.
- the handling of Java language features, such as static methods, inner classes, inheritance, etc.
- limitations of the system: J-Orchestra does not handle all of the Java language. Unsupported features include reflective field access, dynamic loading, volatile variables, and more. Prior limitations [18] with respect to multithreading and monitor-style synchronization have been addressed and the J-Orchestra distributed threading mechanism is described in a recent paper [19].

2.2. The J-Orchestra Rewrite

"Success is 1% inspiration and 99% perspiration"
paraphrasing Thomas Edison

The power of J-Orchestra is not due to any single abstract idea but to many specific techniques for making centralized Java applications run over standard middleware. The J-Orchestra "rewrite engine" is responsible for transforming existing application code and generating new code to turn a centralized application into a distributed one. Transformations happen through bytecode manipulation. (We use BCEL [5] 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 anchored on a different site, 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 a remote implementation class. Remote implementation classes extend the Java RMI class `UnicastRemoteObject`. Subclasses of `UnicastRemoteObject` can be registered as RMI remote objects, which means that they get passed by-reference over the network. That is, 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.

The remote implementation class implements a generated interface that defines all the methods of the original class and extends `java.rmi.Remote`. This interface enables proxies to access remote implementation classes over the network. (RMI creates a "stub" object for such remote access.) We show below a simplified version of the code generated for a class `A`.

```
//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 to either
 //remote implementation directly or RMI stub.
 A__interface ref;
 ...
 void foo () {
  try {
   ref.foo ();
  } catch (RemoteException e) {
    //let 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 {...}
}
```

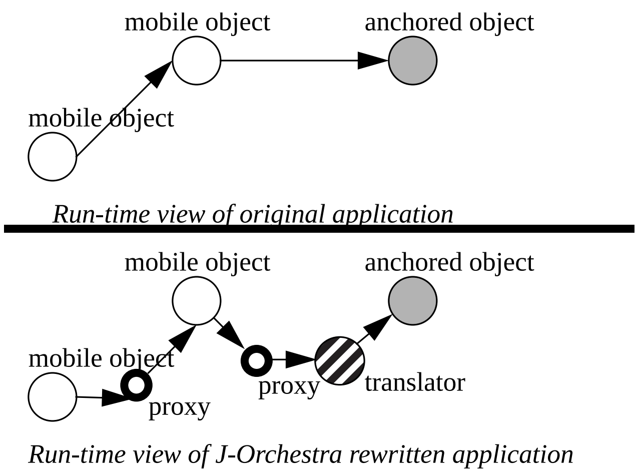

Figure 2: Results of the J-Orchestra rewrite schematically. Proxy objects could point to their targets either locally or over the network.

Proxy classes handle several important tasks. One such task is managing globally unique identifiers, via which J-Orchestra maintains an "at most one proxy per site" invariant. That is, each proxy maintains a unique identifier that it uses to interact with the J-Orchestra runtime system. In addition, all proxies implement `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. Finally, proxy classes are generated in source code form, thus enabling the sophisticated user to supply custom handling code for remote errors.

For anchored classes, proxies provide similar functionality but do not assume the names of their original classes. Anchored classes are accessed directly by their co-anchored clients (i.e., classes anchored on the same site). Therefore, anchored classes cannot change their superclass (to `UnicastRemoteObject`) and must use a different mechanism 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 implementation versions are created for translators, exactly like for mobile classes. Since it is impossible to add classes to system packages, the code generator puts anchored proxies, interfaces and translators into a special package starting with the prefix `remotecapable`. Figure 2 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

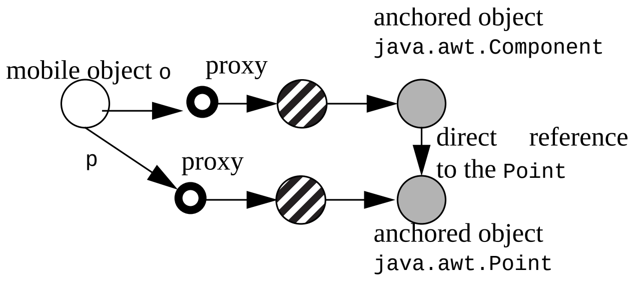

Figure 3: Mobile code refers to anchored objects indirectly (through proxies) but anchored code refers to the same objects directly. Each kind of reference should be derivable from the other.

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 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 to anchored code. For instance, in Figure 3, 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.

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());
 }
}
```

It is worth noting that past systems that follow a similar rewrite as J-Orchestra [9][16][17][20] do not offer a translation mechanism. Thus, the partitioned application is safe only if objects passed to system code are guaranteed to always be on the same site as that code. This is a big burden to put on the user. The translation mechanism is one of the main reasons why J-Orchestra scales to large applications without knowledge of their internals.

2.3. Call-Site Wrapping for Anchoring By Choice

Wrapping and unwrapping need to also take place when anchored objects call other objects anchored on different sites. This case is important in practice, as it enables anchoring by choice. The mechanism is analogous to the mechanism of the previous section, and only differs in the specifics of the code transformation for wrapping/unwrapping. (Readers who are not interested in the low-level mechanisms for wrapping/unwrapping in J-Orchestra can safely skip to the next section.)

Anchoring by choice is beneficial because it can eliminate the J-Orchestra indirection overhead: objects can access co-anchored objects directly. That is, fields can be read/written directly and methods are called as fast as in the centralized application. In practice this 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.

As discussed in the previous section, translators of anchored classes are the only avenue for data exchange between mobile 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* anchored objects, but an anchored object never calls methods of objects that are not co-anchored with it. This was the case for applications of J-Orchestra before anchoring by choice was supported. If the only anchored classes are system classes (whose objects can be touched by native code) then if they access each other they need to be co-anchored and they never access application objects directly (they can only call their methods through superclasses or interfaces, in which case no wrapping/unwrapping is required).

When anchoring by choice is introduced, however, the control-flow patterns become more complex. Since code in classes anchored by choice is regular application code, it can access any other application object. Thus, an anchored object can call a remote anchored object, requiring dynamic wrapping/unwrapping. The problem is that an anchored object has direct references to all its co-anchored objects, but may need to pass those direct references to objects that are not anchored on the same site (they are either mobile or anchored elsewhere). 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 `c.bar()` in class `A`) 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). 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 if caller is not co-anchored with B
 void foo (remotecapable.B b) {
  ref.foo ((B) Runtime.unwrap(b));
 }
 // used when caller is co-anchored with B
 void foo (B b) {
  ref.foo((remotecapable.B) Runtime.wrap(b));
 }
 // used if caller is not co-anchored with B
 remotecapable.B bar() {
  return ref.bar();
 }
 // used when caller is co-anchored with B
 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 could be done if the proxies were created in bytecode form, which we avoided because it would prevent the user from adding custom failure handling code.) We perform a call-site rewrite instead. Since the client is in bytecode form, the 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.

3. Applicability Discussion

"It's not how well the bear dances,
it's that it can dance at all."

J-Orchestra can handle a large subset of Java and, thus, can correctly partition a large class of realistic unsuspecting applications. Nevertheless, among these, J-Orchestra will be useful only in a few well-defined cases. Automatic partitioning is not a substitute for general distributed systems development. The striking element of the approach is not that it is widely applicable but that it is at all applicable, given how automated it is.

We use the term *embarrassingly loosely coupled* to describe the kinds of applications to which J-Orchestra is applicable. An embarrassingly loosely coupled application satisfies two criteria:

- it has components that exchange little data with the rest of the application, and
- these components are statically identifiable by looking at the structure of the application code at the class or the module level.

That is, by looking at static relations among application classes, the user of J-Orchestra (aided by our analysis tools) should be able to identify distinct components comprising multiple classes. Then, during run-time, the data coupling among distinct components should be very small.

Embarrassingly loosely coupled applications can be partitioned automatically without significant loss in performance due to network communication. In order to get any benefit, however, the application needs to have a reason to be distributed. The foremost reason for distributing an application with J-Orchestra is to take advantage of remote hardware or software resources (e.g., a processor, a database, a graphical screen, or a sound card). There are special-purpose technologies that do this already: distributed file systems allow storage on remote disks; remote desktop applications (e.g., VNC, X) allow transferring graphical data from a remote machine; network printer protocols let users print remotely. Nevertheless, the advantage of automatic partitioning is that it can put 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. Of course, there are already technologies for putting code near a resource: Java applets are used to move graphics-producing code from a server to a client with the screen where the graphics will be displayed. Nevertheless, this solution is inflexible: the whole program moves. In contrast, automatic partitioning can split an application so that any part of it can become a "virtual applet" and can run on a client machine.

Of course, one reason to partition an application is to take advantage of parallelism. Distinct machines will have distinct CPUs. If the original centralized application is multi-threaded, we can use multiple CPUs to run threads in parallel. Although distribution-for-parallelism is a potential application of J-Orchestra, we have not examined this space so far. The reason is that parallel applications either are written to run in distributed memory environments in the first place, or their concurrent computations are tightly coupled.

To summarize, we can characterize the domain of J-Orchestra as *partitioning embarrassingly loosely coupled applications for resource-driven distribution*.

4. Scalability of J-Orchestra

In a previous paper [18], we claimed that J-Orchestra is scalable relative to other automatic partitioning systems. In this context, “scalable” means that a correct partitioning is 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 run-time error. Past systems have put the burden of ensuring correctness on the user. For instance, consider Addistant [20], 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 approach is intrinsically unscalable: it means that any object accessed remotely can never be passed to system code. In practice, the only safe approach would be to keep all system code (e.g., graphics, files, threads, etc.) in one partition and only place application objects that never access system objects in other partitions.

4.1. 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 [8]. 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 [8] 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.

4.1.1. 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. Thus co-anchored objects can access each other directly and impose 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 steps of JBits incur *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 4.1.3. 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).[1] 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.

1. 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.

4.1.2. 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.

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.

4.1.3. 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 remote calls take place instead of 484 for a static partitioning, in order to start JBits, load a file and perform 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 unclear 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 suboptimal. 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 those 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.

4.2. 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:

- A Java Speech API demo. 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.
- 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 [14]: 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.

5. 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 [20] and Pangaea [17] systems. The Coign system [11] 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 [11] 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 [17] 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 [9][16] 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 [9]. Similar comments apply to the FarGo [10] and AdJava [7] 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 they do it all together. 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 [4], Orca [2], and, in the Java world, cJVM [1], and Java/DSM [21] 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 [2].

Mobile object systems, like Emerald [3][12] have formed the inspiration for many of the J-Orchestra ideas on object mobility scenarios.

Both the D [13] and the Doorastha [6] 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 full understanding of the application behavior and can be error-prone: a slight error in an annotation may cause insidious inconsistency errors.

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

References

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

[2] 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.

[3] 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.

[4] 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.

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

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

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

[8] 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/`.

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

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

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

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

[19] Eli Tilevich and Yannis Smaragdakis, "Portable and Efficient Distributed Threads for Java", submitted for publication, available at `http://j-orchestra.org`.

[20] 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.

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