Application Partitioning without Programming (a White-Paper and Future Work Proposal)

Yannis Smaragdakis and the J-Orchestra Group
College of Computing
Georgia Tech

July 25, 2001

Application partitioning is the task of breaking up the functionality of an application into distinct entities that can operate independently, usually in a distributed setting. As networking changes the computing landscape, application partitioning is becoming the main kind of distributed programming. Even the plainest, non-performance-oriented applications may need to be partitioned due to functional considerations: the resources that the application needs (e.g., graphical workstation, database system, sensor hardware) may be distributed throughout a network. Traditional partitioning entails re-coding the application functionality so that it uses a middleware mechanism (e.g., CORBA, Java RMI) for communication between the different entities. This proposal examines an alternative approach that involves no programming. Instead, higher-level tools allow the user to express how the application is to be partitioned. The tools can then rewrite the existing application code to replace local data exchange (e.g., function calls, data sharing through pointers) with remote communication (e.g., remote function calls, remote pointers or mobile objects).

The no-programming approach has the potential to revolutionize the way applications are partitioned. Nevertheless, no-programming partitioning faces two major challenges. First, it is hard to guarantee the *completeness* of the translation process, by changing the application alone. "Completeness" refers to the ability to place any arbitrary subset of application data and code on any site. A second challenge is to obtain *acceptable performance* for a large class of applications. This requires both careful analysis of the data exchange patterns among application entities, and appropriate mechanisms for data migration and possibly replication. Nevertheless, doing either of the above with low overhead and no changes to the runtime system is a difficult task.

The goal of this proposal is to explore whether no-programming application partitioning can be advanced to "industrial strength" levels. The space of possible design choices is huge, but we will argue that a small segment of the spectrum holds some of the most promising design directions. In particular, we propose to explore no-programming partitioning of Java applications in bytecode format. A separate dynamic profiling phase will be used to supply information to guide both partitioning and distribution decisions. Graphical tools will aid the user in describing a correct and efficient partitioning of the application. The proposed rewriting algorithm (a new research result) is far more complete than previous approaches, and allows full mobility of application objects. Static analysis will be used to enlarge the set of partitionings that are guaranteed to be correctly distributed by the system and to perform optimizations in the distribution middleware. Preliminary work in these directions is being conducted at the time of writing this proposal.

1. Introduction

Programming distributed applications used to be a task reserved for high-performance computing and large, geographically separated systems, always designed from scratch with distribution in mind. With the widespread use of the Internet, distribution over the network became an issue for a large number of applications that before would operate on a single location. Distributing such applications leaves the functionality they offer to the user virtually unchanged. Physical constraints are the reason dictating the distribution.

For instance, an application should continue to work the same, but now its user is geographically separated from the data storage facility or the main computing engine. The Java *applet model* is a good example, when viewed as an instance of distributed computation. An applet is a piece of code that originally exists on a server machine but gets copied on a client machine to be executed on a user's Web browser. Typically, the applet is executed on the client machine not because this machine is faster than the server that the applet came from, but because the applet needs to use a local resource—the graphical screen of the user machine. Since the graphics have to reach the user screen and the code is initially on the server machine, distribution is inevitable. The main issue is how the distribution should take place. In the case of applets, the answer is hard-coded and it is the same for each applet: the code is downloaded and executed on the user side. Nevertheless, one can imagine many other solutions that are customizable for individual programs. Perhaps, the functionality should be split, with the core part executed on the server, while the user interface is executed on the client. Communication between the two parts could be performed with standard distributed computing techniques (e.g., CORBA [14], or Java RMI [20] middleware). Perhaps, objects should migrate on demand, or according to an application-specific pattern.

Such circumstances gave rise to application partitioning. *Application partitioning* is the task of breaking up the functionality of an application into distinct entities that can operate independently, usually in a distributed setting. Application partitioning is advocated strongly in computing magazines (e.g., [11]) as a way to use resources more efficiently. Traditional partitioning entails re-coding the application functionality to use a middleware mechanism for communication between the different entities. This is a significant undertaking, often prohibitively so. In this proposal, we promote the idea of partitioning existing centralized[1] applications without manually changing the application source code. Instead, a higher level tool allows the user to express how the application is to be partitioned. The tool can then rewrite the existing application code to replace local data exchange (e.g., function calls, data sharing through pointers) with remote communication (e.g., remote function calls, remote pointers or mobile objects). This *no-programming*[2] approach to application partitioning has significant simplicity advantages and can revolutionize the way applications are partitioned.

In the spectrum of technologies aimed at facilitating distributed computing, no-programming partitioning is among the most ambitious, because it imposes modest requirements. To elaborate this somewhat paradoxical statement, no-programming partitioning is an ambitious approach on the technical front, but very modest on the deployment front. The distinct element of the approach is that only the application changes—*no changes are required to the runtime environment where the applications are to be executed*. This distinguishes no-programming partitioning from distributed shared memory systems (e.g., CJVM [1], Java/DSM [25]). The deployment advantages include full portability and compatibility under third-party changes to the runtime system. Typical technical advantages include the compactness of the resulting distributed system and the transparency of the partitioning to other elements of the system (e.g., collaborating applications running on the same runtime system). No-programming partitioning aspires to be successful in distributing a large class of applications semi-automatically, while maintaining correctness and good performance. This is a challenging task but the deployment advantages guarantee that the impact of the approach depends only on the technical success of the rewriting process—not on the extent of adoption of new infrastructure.

---

1. We will use the term "centralized" for applications designed to run on a single machine. Note that the distinction between *centralized* and *distributed* is orthogonal to the distinction between *sequential* and *concurrent*. Both centralized and distributed applications can be either sequential or concurrent. More specifically, the no-programming partitioning approach has nothing to do with concurrency discovery (e.g., work on automatic parallelization).
2. As explained later, "no-programming" is a slight misnomer. If application developers want to add non-default failure handling to their distributed applications, they need to add code to automatically generated skeleton modules. Even in this case, the original application code does not need to be altered (e.g., it can well be in binary format) and the error handling is no more complex than in the case of traditional application partitioning.

Of course, it is utopian to expect that *all* applications can be distributed without code modifications and attain acceptable performance. Nevertheless, there are good reasons to hope that the class of applications for which no-programming partitioning can yield efficient solutions is large and only getting larger. Some of these reasons are:

- When distribution is dictated by physical constraints (as on the Internet and in embedded systems environments) communication patterns tend to be very simple. Consider again the example of applets, or the symmetric case of *Java servlets*: surely if the problem admits a solution that executes the entire code exclusively on the server (servlet) or exclusively on the client (applet), the communication requirements cannot be too great. It should be easy for an automatic system to perform strictly better partitioning than an inflexible solution like applets or servlets.
- The breakdown of applications in objects seems to offer a good granularity for making distribution decisions and applying them to binary code. Non-object-oriented applications offer abstraction boundaries only at the level of procedures or modules. The former seem too fine-grained for distribution decisions, while the latter are too coarse-grained. Binary executables in an architecture-specific format (e.g., x86 machine language) would be hard to process automatically. In contrast, the object-oriented coding style, in combination with more abstract execution environments (e.g., the Java VM, or the Microsoft CLR) offer both an appropriate partitioning granularity and significant ease of binary manipulation. Therefore, the current increasing trend of writing applications in object-oriented languages with abstract runtime systems (like Java or C#) favors no-programming partitioning.
- Good techniques for placement, replication, and mobility have been developed and appear in the distributed systems literature. These include placement and data consistency techniques from Distributed Shared Memory systems (e.g., Orca [2]), object mobility techniques (e.g., from the Emerald system [3]), etc. Additionally, with a judicious combination of static analysis and execution profiling, distribution decisions can be more educated than in past systems.

2. Technical Issues and Design Choices

No-programming application partitioning faces some serious challenges. The two main issues are those of the *completeness* of the translation process and the *performance* of the partitioned application in a distributed environment. These are the axes along which we will examine existing systems, as well as the proposed approach. For concreteness, we will contrast the proposed approach to the three most closely related prior art systems—Addistant [21], Pangaea [17][19], and Coign [9]. Other, less closely related, work (e.g., distributed shared memory systems and manual partitioning infrastructure) will be discussed in Section 3.

The past three years have seen a number of efforts to automatically partition Java programs [17][21] and COM applications [9]. These prior approaches are limited in scope and have not demonstrated success in partitioning third-party, pre-written applications. The Coign automatic partitioning system [9] has made some inroads in this direction, but the applications partitioned successfully (e.g., the Octarine word processor) were written explicitly to demonstrate a modular style of programming using COM components. No other real-world COM applications are written with such extreme care to ensure binary modularity.

The goal of this proposal is to advance no-programming application partitioning to “industrial strength” levels. The platform of experimentation is Java and we propose to perform the partitioning through Java bytecode rewriting. Thus, no source code access to the original application is required. Other main elements of the proposed approach include: a test-case profiling phase for the application that will supply information to guide partitioning, placement, and mobility decisions; a powerful rewriting engine allowing correct partitioning of more applications than prior approaches; generating source code skeletons where failure handling code can be added by the user; the ability of objects to move, whenever possible; static analysis to enlarge the set of partitionings that are guaranteed to be correct and to perform optimizations; a

graphical interface that will present the results of partitioning and static analysis to the user and will allow the user to make distribution decisions; heuristic algorithms for partitioning (data placement) based on data exchange information.

Our group at Georgia Tech already conducts preliminary work in the proposed direction. The name of our system is *Java-Orchestra*, or *J-Orchestra*, for short.[3]

Next, we will argue that the J-Orchestra design decisions are sound and represent the most promising avenue to an industrial-strength no-programming partitioning system. A few of the aspects we will discuss have already been implemented, but most either have not or are at an immature stage. The research plan in Section 4 will list specifically what the proposed future work is—the current section is concerned with the overall approach.

Java Bytecodes as a Program Representation. The Java programming language [8] is the dominant language for Internet development and one of the most dominant programming languages overall. The enormous number of Java developers (conservatively estimated at 500,000 professional developers in 2000, to grow to over 2 million by 2005, not including students and hobbyists [7]) and the accumulated Java expertise guarantee the potential for significant impact. Additionally, Java is among the purest object-oriented languages. This ensures that "legacy" applications (i.e., applications written with no distribution in mind) are fairly modular. Class boundaries offer convenient lines along which the partitioning can take place. Objects offer a conveniently fine granularity for code and data mobility. Furthermore, Java programs are executed in an abstract execution environment—the Java Virtual Machine (JVM). This enables ease of binary manipulation of the application code, as well as the ability to manipulate the runtime environment (e.g., to transform code at load time, to enable dynamic profiling, etc.).

Among the three most closely related approaches to the one proposed, two—the Pangaea [17] and Addistant [21] systems—are Java based. Pangaea operates at the source code level, while Addistant, like the proposed approach, operates at the bytecode level. Operating at the bytecode level is essential for generality, because no access to the source code for the original application is needed, and because Java system classes also need to be manipulated (although not modified). The Coign system [9] operates on COM components. One disadvantage in this case is that no real-world applications are written as collections of many small COM components. The applications that constitute success cases for Coign (mainly the Octarine word processor) were written specifically to showcase that COM is a viable platform for developing applications from many small components. A second disadvantage is that COM applications (in platform-specific binary format) are hard to rewrite, which is necessary in order to change the way that remote data are accessed. This will be further discussed when examining the rewriting engine design choices.

Using Test-Case Profiling Data for Partitioning Decisions. Test-case profiling consists of examining the behavior of an application under some sample input. The observations made during the test run are then used to guide decisions that will affect application performance during actual use. Test-case profiling is ideal for no-programming application partitioning because it is hard to perform low-overhead online monitoring of application behavior without changing the execution environment. Recall that keeping an unmodified execution environment is one of the main goals of application partitioning, for portability and ease of adoption reasons. Of course, the main assumption of test-case profiling is that the test-case is representative of general application behavior. For initial partitioning of application classes (i.e., deciding which classes are strongly coupled with which others), this is likely to be the case. For more complex infer-

3. The motivation behind the name "J-Orchestra" is dual. First, it suggests the kind of orchestration of object mobility that the system aspires to perform. Second, there is a strong analogy between application partitioning and the way orchestral pieces are commonly composed: first a piano score is completed. Then an "orchestration" process takes place that determines which instrument should play which notes of the completed piano score. The analogy extends far. For instance, there are many examples of orchestrating piano music that was never intended by its composer for orchestral performance.

ences—like deciding when to move an object from a site to another—profiling has to be very sophisticated to be adequate. This is one of the most open-ended directions we propose to explore.

Of the three prior art systems discussed, only Coign uses test-case profiling to guide partitioning decisions.

A General, Efficient Rewriting Engine that Allows Object Mobility. The most important part of a no-programming partitioning approach is the rewriting it performs. The issues involved range from engineering considerations to deep research problems. First, we will present the necessary background and a critical view of prior art—the issues discussed here are generally overlooked in the literature and past no-programming partitioning systems have not documented exactly the limitations of their approaches. Although the discussion is often Java-specific, the techniques described are fairly general.

To begin, let us distinguish between different kinds of classes that form a Java application. The first kind is *application classes*. These are part of the original application to be partitioned. As such, they can freely be modified (e.g., to refer to other objects through proxies). Next there are *system classes*—the classes forming the Java runtime libraries, which are universally available. For brevity, we will occasionally call instances of system classes *system objects*. Many system classes are implemented using *native code*—i.e., their functionality is encoded in a platform-dependent binary file, either the JVM executable itself or a dynamically linked library. There is a distinction between system classes that invoke native methods and ones that do not. The latter may be treated as application classes under some (strict) conditions.

Completeness is the main problem of no-programming application partitioning. Without modifying the runtime system, it is hard to guarantee correct execution for *all* applications partitioned along *any* user-defined boundary. The problem stems from the fact that most mainstream programming languages allow data sharing through references (i.e., pointers). Sharing through references is valid on a single address space. When clients are distributed over a network, however, there is no way to directly access remote data. For application code, this may be fine, as long as code modification is possible: application code can be rewritten to always access data through indirect references. If, however, references ever leak to unmodified code that is unaware of the distribution (e.g., native code in the Java VM, system libraries for a platform-specific executable) disaster will ensue: the code will try to access the data directly, even if the data are remote. Even if care is taken to only pass direct references to unmodified code, there is no guarantee of correctness: the unmodified code may alias the data, so that replicating or moving them will violate the original application semantics. In general, if an arbitrary subset of the code is unmodifiable, it is an undecidable problem whether a given partition of the application will respect the original semantics. The reason is that the aliasing behavior of the unmodifiable code depends on run-time information. The aliasing behavior, in turn, determines whether a piece of data can be safely moved to a remote site.

Performance is the other major issue in no-programming application partitioning. There are certainly some applications for which performance is secondary. Nevertheless, the ultimate success of no-programming partitioning depends on the efficiency of the partitioned application. It is tempting to view the problem as one of data locality and apply standard techniques (e.g., from distributed shared memory systems). Most of these techniques have to do with data replication and data mobility. A wealth of past research experience has shown that replication and mobility are essential tools for good performance in distributed applications. Nevertheless, in this case, correctness is the enemy of performance. For instance, it is not necessarily the case that a set of data (e.g., a vector) can be moved to a different host or can be replicated. Clients of the vector may be unmodifiable, thus needing to access the vector data directly (which is not possible if the vector is remote). Unmodifiable system code may hold an alias to the vector, preventing it from moving, or, rather, violating the semantics of the application if the vector moves. Similarly, replicating data may be unsafe, as it may lead to inconsistencies between copies. Traditional mechanisms for keeping copies consistent are not sufficient because there is no way to intercept access to the data without modifying either the client or the runtime environment.

The above discussion suggests that good conservative approximations are required to achieve correctness of the partitioning, but these should not sacrifice performance. The proposed approach consists of just such a conservative approximation. First, though, let us consider how the issues of completeness and performance have been addressed in prior systems. All three of the examined systems impose very severe restrictions on the kinds of partitionings allowed and have little, if any, support for data mobility. The limitations are such as to render these systems fundamentally unscalable:

- the Coign system does not distribute components when they share data through memory pointers. Such components are deemed non-distributable and are located on the same machine. Practical experience with Coign [9] showed that this is a severe limitation for the only real-world application included in Coign's example set (the Microsoft PhotoDraw program). Note that the Coign approach would be impossible in the case of Java: almost all program data are accessed through references in Java. No support for synchronous data mobility exists in Coign, but the application can be periodically repartitioned based on its recent behavior.
- the Pangaea system uses the JavaParty [15] infrastructure for application partitioning. Since JavaParty is designed for manual partitioning and operates at the source code level, Pangaea is also limited in this respect. Thus, Pangaea cannot be used to make Java system classes (which are supplied in bytecode format) remotely accessible. This is a very severe limitation as most data exchange in Java programs happens through system classes (e.g., collection classes, like `java.util.Vector`). If such classes are not remotely accessible, all their clients need to be located on the same site, making partitioning almost impossible for realistic applications.
- the Addistant system concentrates on functional distribution along library boundaries. The kind of rewrite employed (i.e., the semantics supported) is picked manually (e.g., the user has to pick a different rewrite for classes that can be freely replicated, a different rewrite if an application class references an unmodifiable system class, etc.). There are certainly some arbitrary limitations in the Addistant approach. (For instance, `final` system classes cannot be accessed remotely due to the subclassing-based rewrite technique. All clients of such classes need to be on the same host.) Even so, however, the set of partitionings supported is much richer than that of Pangaea. The main problem with the Addistant rewrite, however, is that objects cannot move from site to site. Instead, objects stay on the site where they were initially allocated but can be accessed remotely. Thus, the only way to get acceptable performance with Addistant is by *copying* objects. Nevertheless, allowing copying semantics is a responsibility left for the user! That is, a user with only bytecode access to the original application is expected to know whether a class in that application can be safely replicated without violating the application semantics. This limitation makes Addistant impractical for any but trivial applications, usually partitioned by the original author. Essentially, Addistant does not solve the completeness issues, but shifts the burden to the end user. Even under this restriction, the user is limited to specifying copy semantics for the objects—object mobility is not allowed.

Given the above restrictions, it is easy to see why prior systems have not scaled to industrial level applications. The rewriting engine proposed here addresses most of the above concerns. First, we will sketch a conservative (with respect to Java system classes) approach, which is still much more general than all the above techniques. (This constitutes a new, implemented, but yet unpublished, research result.) Later in this section, we will discuss how static analysis can be used to enable safe mobility even of some Java system classes. The discussion is simplified (by abstraction but also omission of special cases—e.g., treatment of `this` references) but maintains all the crucial insights.

The J-Orchestra rewriting engine uses the standard technique (e.g., see JavaParty [15]) of changing references to objects (which we will call *direct references*) to point to a proxy object instead (*indirect references*).[4] The proxy object hides the details of whether the actual object is local or remote. The invariant maintained is that clients never get direct references to objects that can potentially be remote—access is

always through a proxy. Application code needs to be rewritten to maintain the invariant: for instance, all `new` statements have to be rewritten to create a proxy object and return it, an object has to be prevented from passing direct references to itself (as the value of the `this` expression) to other objects, etc. If other objects need to refer to data fields of a rewritten object directly, the code needs to be rewritten to invoke accessor and mutator methods, instead. Such methods are generated automatically for every piece of data in application classes. (For instance, if the original application code tried to increment a field of a potentially remote object directly, like in `o1.a_field++`, the code will have to change into `o1.set_a_field(o1.get_a_field() + 1)`. The rewrite will actually occur at the bytecode level.). Consider now the completeness issues discussed earlier: what if unmodified code (e.g., native methods) tries to access object fields directly? What if unmodified code aliases the data?

The main observation is that none of these cases applies to instances of application classes (i.e., classes of the original application to be partitioned, as opposed to Java system classes). Unmodified (system) code can only access application objects in three ways: through generic (`Object`) references, through Java interfaces, or if their classes are derived from system classes. In the third case, the classes are not really application classes—they represent subtypes of system classes and should be treated just like system classes. In the first and second cases, no problem exists (after careful rewriting). Direct access to the data is not allowed, thus the proxy is used. Also, since no object can directly refer to a potentially remote object, if unmodified code is to alias the application object, it will instead alias its proxy object. For instance, if the program creates a collection (e.g., a `Vector`) of instances of application class `C`, a vector of objects that are proxies to instances of `C` will be created. In this way, instances of application classes can freely move—their location is entirely transparent to the rest of the system.

Consider now system classes. Objects of system classes will also be accessed through a proxy, when used in application code. Nevertheless, the problem is that other system classes may need to reference these objects. For example, system code (or even native code) may create an instance of a system class, keep a reference (alias) to it, and return that reference as the result of a method invocation. Additionally, system code is for practical purposes unmodifiable. (In reality, some system code is perfectly modifiable, but not all system code is—due to native methods. Treating all system code as unmodifiable code yields a cleaner solution and does not run the danger of possibly violating licensing agreements.) Thus, there are at least two modes of operating on system objects: using direct references (from inside other unmodifiable system classes) or using indirect references/proxies (from inside application classes). The issue then becomes how to translate between direct access to system classes and indirect access. Fortunately, the proxies for system objects offer the only interface between application and system code. The code of these proxy classes can take care of the translation so that system classes can refer to system objects directly but application classes only do so through proxies. In particular, if a direct reference to a system object is about to be returned to application code, it is first "wrapped" with a proxy object and the proxy object is returned. Similarly, if a system class is to be passed into system code from application code, it is first "unwrapped". (In fact, a similar transformation takes place for application classes that implement system interfaces.)

The rewrite algorithm sketched above takes care of unmodified code trying to access data directly, but it does not address the issue of unmodified code possibly aliasing the data. Indeed, system code can keep an alias to objects (instances of other system classes). If these objects are later moved, the aliases will be invalid. Therefore, in the strictest case, this rewrite only takes care of the case where *all* system objects can only exist on one network site. Nevertheless, Java system classes can be subdivided into libraries, which have hierarchical dependencies. For instance, the Swing graphics library may access classes from the

4. This point is worth repeating for clarity: we will use the term "direct reference" to indicate a normal Java reference (i.e., an indirection). This could well be called an "indirect reference" but since Java does not allow any more "direct" access to objects, we can "hijack" the term without ambiguity. Summarizing the terminology used, "direct references" offer a single point of indirection, while "indirect references" offer two points of indirection, the second one hiding whether the object is local or remote.

`java.util` package, but not the other way around. Thus, Swing classes can easily (all together) be assigned to a single host, while `java.util` classes are on a different host. This is the same property that the Addistant system exploits in order to allocate Swing classes on remote hosts.

Now let us summarize the completeness of the J-Orchestra rewrite algorithm and compare it to past rewrite algorithms. The proposed rewrite allows correct distribution for *all* applications partitioned along *any* user-defined boundary across application classes. That is, the user can put *any* instance of *any* application class on *any* site, and the semantics of the original centralized application will be preserved. Additionally, application objects can freely move from site to site during application execution, without endangering the correctness of the application. Therefore, the rewrite algorithm is much more general than all previous no-programming partitioning approaches. The algorithm's only limitations have to do with partitionings of system classes. System objects have to be located on the same site as any other system objects that potentially alias them. As a conservative approximation, all instances of system classes in the same conceptual "library" (e.g., Swing classes, `java.util` classes, etc.) have to be on the same host, although they can still be accessed remotely. Even in the case of system classes, the J-Orchestra rewriting algorithm is at least as complete as any previous one.

Static Analysis for Performance Optimizations. The limitation of the rewriting algorithm sketched above is that system objects have to be on the same site as other system objects that can potentially alias them. There is no way to compute exactly what objects will alias other objects. Thus, we have to resort to conservative approximations, like enforcing that all instances of classes in the same "library" have to be on the same host. This can be too restrictive, even for realistic examples, as it means that such objects cannot move. This restriction can be relaxed with careful application of static analysis. For instance, many classes in the same library never alias instances of each other in a way that will cause problems if the instances move. Objects of such classes can freely exist on different sites on the network.

Static analysis can also enable a wealth of optimizations. For instance, if the code of a system class only accesses other system objects by calling methods through an interface, then we are able to avoid the overhead of translating between indirect references and direct references when passing arguments to methods of this class. If a certain method is guaranteed to never change some of its arguments, these arguments can be passed using "by-copy" semantics instead of the "by-reference" semantics that the proposed rewrite algorithm guarantees in order to emulate Java behavior on a single address space. If a method does change its arguments, but references to them never escape the body of the method, then we can pass the arguments using "by-copy-return" semantics, again resulting in a more efficient implementation.

Of the three prior art systems discussed in this section, only Pangaea has attempted to use static analysis to infer relationships between objects [18]. This work is still at a preliminary stage.

Enabling the User to Add Failure Handling Code. The overall approach of programming distributed systems as if they were centralized ("papering over the network") has been occasionally criticized (e.g., see the best known "manifesto" on the topic [23]). The main point of criticism has been that distributed systems fundamentally differ from centralized systems because of the possibility of partial failure, which needs to be handled differently for each application. Nevertheless, the proposed approach does not suffer from this problem: although the input of the system is a binary application, the output is both a rewritten binary and the *source code* of new front-end classes (skeletons) required to run the application in a distributed environment. These front-end classes offer a wrapper for the rewritten binary functionality of the original application. Application-specific (i.e., non-default) partial-failure handling can be effected by manually editing the source code of the front-end classes and handling the corresponding Java language exceptions. Thus, although the proposed work involves hiding (much of) the complexity of distribution, it allows the user to handle distribution-specific failure exactly like it would be handled through manual partitioning. Alternatively viewed, the user can concentrate on the part of the application that really matters for distributed computing: partial failure handling. This part is the only code that needs to be written by hand in order to partition an application.

Graphical Front-End for User Interaction and Heuristics for Partitioning. The goal of the proposed approach is to enable application partitioning at a higher level of abstraction. Therefore, it is natural to include a graphical front-end to allow the user to specify partitioning parameters. Ideally, such a user interface should present all the results of program analysis so far and allow the user full flexibility in making further decisions. This presents some challenging user-interface issues. How should static analysis information be represented in an approachable form? How can profiling information be represented? How can the user deal with the complexity of hundreds or thousands of classes and methods? How can the user easily specify object migration policies (e.g., "when method `foo` is called, its third argument should move permanently to the site of `foo` if it is not already there")? How can the user override static analysis information (e.g., to assert that a method never modifies its arguments, even if this is not apparent to the static analysis algorithm)?

Although we cannot provide complete answers to these questions, preliminary experience suggests a few good directions. First, the user should always be in full control of the distribution process, if needed. On the other hand, heuristics for distribution (e.g., a flow-based static partitioning algorithm) should be readily available to provide some automatic decision making. In this way, the user can be sure that most of the "don't-care" cases are handled in an acceptable way. If a structured language (e.g., XML-based) is used for externalizing the distribution information, then an editor for the more complex structures (e.g., migration policies) can be integrated directly in the graphical user interface. In this way, graphical information and complex structures with no direct graphical representation are integrated smoothly. This is a technique successfully employed in development environments like Visual Basic. The advantage over separate editing of the complex structures is that the hierarchical capabilities of the graphical environment are exploited: the user can click on a class, choose one of its methods, then edit the migration policy for arguments of the method. In general, a hierarchical philosophy in the user interface is a good way to deal with complexity. The user should be able to group classes together to form larger entities that are used as a unit. The system should then be able to summarize profiling and static analysis information for the entire group.

3. Related Work

The most closely related pieces of work were discussed in detail in the previous section. Here we will discuss other work that is less directly related to the proposed approach. The spectrum of infrastructure for distributed computing is huge so presentation will be selective.

Distributed shared memory systems (DSMs) like Munin [4], Orca [2], and, in the Java world, CJVM [1], and Java/DSM [25], can be used to offer transparent distribution, but they are different from no-programming partitioning in that what changes is the runtime environment and not the application. Furthermore, distributed shared memory systems are typically used in local environments, and are geared towards scalable high-performance applications. Thus, unlike the proposed approach, most programs do get modified to run efficiently on a distributed shared memory system, and little opportunity exists for handling network failures intelligently at the application level.

Among distributed shared memory systems, the ones most closely resembling the proposed approach are object-based DSMs, like Orca [2]. The Orca system has a dedicated language and runtime, but also has similarities to the proposed approach in its treatment of data at the object level, and its use of static analysis. In fact, a no-programming partitioning approach in Java can be viewed as a way to simulate an object-based DSM using traditional middleware and application rewriting. The rewriting algorithm described in Section 2 essentially causes an appropriate mobile object environment to be maintained at application run time, even though the runtime environment remains unchanged.

Mobile object systems, like Emerald [3][10] have similarities with the proposed approach. Some of the ideas on implementing mobile objects and choosing appropriate semantics for method invocations (synchronous object migration) are identical in the proposed approach and in Emerald.

The Doorastha system [6] represents another piece of work closely related to no-programming partitioning. Doorastha allows the user to annotate a centralized program to turn it into a distributed application. Unfortunately, all the burden is shifted to the user to specify what semantics are valid for a specific class (e.g., whether objects are mobile, whether they can be passed by-copy, etc.). The Doorastha annotations are quite expressive in terms of how method arguments, different fields of a class, etc., are manipulated. Nevertheless, programming in this way is tedious and error-prone: a slight error in an annotation may cause insidious inconsistency errors.

The need for infrastructure to support application partitioning has been recognized in the systems community. Proposals for such infrastructure (most recently, Protium [24]) usually try to address different concerns from those covered in this proposal. High performance is an essential element, with the infrastructure trying to hide the latency of remote accesses. The no-programming partitioning approach aims at a much higher degree of automation, but for applications with more modest network performance requirements.

4. Research Plan

The main goal will be to evaluate whether no-programming application partitioning can become an industrial-strength technique, scaling to large third-party applications and providing acceptable performance. Previously (Section 2) we argued that the proposed approach makes the right general design choices. Thus, the proposed approach is capable of demonstrating or disproving the potential of no-programming partitioning. Nevertheless, the design choices outlined in Section 2 are general enough that significant space for exploration exists. This section describes a specific research plan covering this space.

Some preliminary work has already taken place, but only just started yielding fruit. (No publications on this preliminary work exist.) In particular, at this point, the J-Orchestra system implements the general rewrite algorithm outlined in Section 2 using Java RMI [20] as the target middleware. The implementation is currently inefficient—completeness has been the primary goal so far. Now performance becomes a major consideration. Other aspects of the proposed approach have primitive implementations. There is a rudimentary profiler, constructed by instrumenting Sun's Java VM using JVMPI (the JVM Profiler Interface) [22]. The profiler currently only records the total amount of data exchanged during all calls of all methods of a class. Static analysis is at an early stage with only a preliminary Java implementation of a published alias analysis algorithm [12]. The static analysis part of the approach is one that needs to reach a high level of maturity before it can be integrated with the rest of the components. A simple graphical interface allows the user to see a list of classes together with call-graph information (what other classes' methods call/are called by methods of the class). Simple profiling information can be input and is represented in a weighted-graph view of the application classes. An implementation of a greedy static (class-based) partitioning algorithm is available to aid the user in object placement.

Most of the elements of the approach that will enable it to achieve industrial strength are still to be developed. Our specific research agenda will include exploration along the following axes:

- *Powerful profiling*: the profiler should be the main source of information for object placement and migration.
- *Static analysis*: is static analysis accurate enough to help? Are more complex algorithms better? Is static analysis essential? How should static analysis interact with profiling information?
- *Supporting Technology*: how important is the underlying middleware for performance? What is a good middleware infrastructure for no-programming partitioning? What optimizations can be performed at the bytecode level to eliminate the overhead of the rewriting process?
- *Applications and Evaluation*: what applications can be successfully partitioned? Are there important practical benefits of the approach? What application domains most benefit from no-programming

partitioning? Is no-programming partitioning suitable to high-performance applications? How does it compare to traditional Distributed Shared Memory systems?

These directions are analyzed below in detail:

Powerful Profiling. Test-case profiling is a major element of the proposed approach. The needs of no-programming partitioning will likely test the limits of profiling. The profiler should not just be able to report cumulative information about data exchange, but also to extract a model of the application (e.g., based on temporal patterns of application behavior). For instance, the profiler could recognize situations that will enable object migration strategies, such as "whenever method `m` is called, its arguments should migrate to the site where method `m` is executed". Because of the dual nature of the profiler, it is best to think of it as two tools: one that will record program actions in great detail, and one that will analyze the actions and infer strategies from them.

Specifically, we will do the following:

- Build a scalable profiler that records synchronous events in the system (mainly method calls), the amount of data exchanged, and any other information deemed necessary in the course of the project. Producing a scalable profiler tool is not trivial and may require modifying a Java VM implementation. (The modified JVM will only be used for profiling and will never need to be deployed with the partitioned application.) Our past experience with a profiler based on the JVM Profiling Interface was not entirely satisfactory: application profiling is slower than normal execution by a factor of almost 1,000. This may be sufficient for infrequent profiling, but it is a problem if multiple profiling runs of each application are needed.
- Build a tool to analyze the information from the test-case execution of an application. Experiment with different strategies to draw high-level inferences. For each method and each argument to the method, collect information about the use of arguments. Example questions include:
  "Is the client of a method call more tightly coupled to the data passed as arguments than the object on which the method is called?"
  "When a reference is passed as an argument, how deep are the paths from this reference that a method traverses?"
  "Does a method typically change its n-th argument?"
- Explore how to extract "representative" information from profiling runs. Profiling can never capture the full complexity of all application executions, but can we get an acceptable approximation? Promising approaches include combining inferences from different profiling runs and concentrating on static elements (e.g., object creation statements) instead of specific objects.

Static Analysis. Properties of the partitioned application can be discovered automatically using static analysis. The static analysis space is fairly open ended, leaving many possibilities for exploration.

Specifically, we will:

- Implement published alias analysis algorithms as the first step to the required static analysis tasks. We will also explore a new alias analysis algorithm that we have designed and believe is appropriate for the domain.
- Implement escape and modifiability analysis algorithms on top of the alias analysis techniques. Test the difference in accuracy when different alias analysis information is used.
- Perform analysis of Java system classes. A challenge will be to see if safe and useful results can be extracted without analyzing native code. Some simple results of this kind should be easy to achieve (e.g., many system classes can be referenced only by very few other classes).

- Explore how static analysis results affect the profiling process and vice versa. To see why there is interaction, recall that part of the interesting information is whether an object can be aliased at a specific site. The answer to this question depends on what other objects are on the same site. An initial distribution of objects based on profiling information can make the static analysis less conservative. Similarly, static analysis information (e.g., the fact that a method does not change its arguments) affects how much data is transferred during a method invocation, which, in turn, affects the placement and migration decisions made in the profiling phase.

Supporting Technology. The performance of a partitioned application depends on having an efficient runtime system. In this direction we will:

- Explore what middleware technology is appropriate for no-programming partitioning. Compare different middleware technologies for performance and services offered. Application partitioning technologies, like JavaParty, have often used optimized middleware to improve performance [13].
- Explore optimizations specific to the rewriting algorithm proposed. Rewriting application classes to access other objects through indirect references imposes overhead. Several techniques can be used to limit the overhead to the absolutely necessary cases. Lazy conversion of direct to indirect references may be possible. Allowing direct field access to local objects seems promising. Efficient treatment of the `this` expression by changing the local variable array may be an option.

Applications and Evaluation. A large part of the proposed work has to do with evaluating the impact of no-programming partitioning. Although not all applications can be partitioned automatically, it will be beneficial to have a classification of applications that are amenable to no-programming partitioning. Clearly this evaluation process is closely coupled with the rest of the exploration. If some class of applications proves to be hard to partition well, this may motivate adding new capabilities to the back-end, profiler tools, or static analyzer.

Specifically we will do the following:

- Collaborate with colleagues implementing applications for naturally distributed environments. Georgia Tech has particularly active research groups designing and implementing applications for embedded systems and alternative computing environments. Applications in these domains operate in a heterogeneous, distributed environment with functional distribution constraints: cameras may be connected to one machine, sensors to another, while a database system runs on a central server. Java is often used to hide the platform specific elements of each environment. (Lately, Java has made great inroads to the embedded systems domain, in general—tens of millions of Java-enabled cell phones are in use in Japan [5].) Ease of application development is paramount, as the applications are not developed by systems experts. Therefore, the proposed approach has a lot of potential in these contexts. Lots of small machines (like Java-enabled cell phones) can be running parts of an application originally intended only for centralized execution, while the rest of the application runs on a central server, or even on other small machines.
- Experiment with interactive applications to make them receive input or produce output on sites other than where computation occurs. Example applications include command shells (e.g., the JShell—a Unix shell look-alike for the Java VM), and Swing applications (where the graphics will be displayed remotely). The goal is to enable a better partitioning than what would happen if individual keystrokes, or entire graphics windows were transferred over the network (as, for instance, in the telnet or X-Windows protocols). In the case of JShell, for example, the parsing of commands can be done on the client side and only their execution needs to take place on the server side. Eventually the system should scale to industrial-strength applications and examples should abound. A Georgia Tech colleague has already requested a partitioning of the JBits 2.5 FPGA simulator by Xilinx. The goal is to use this heavy-duty application over a network link without incurring as much overhead as X-Windows.

- Experiment with high-performance applications to see how the approach can deal with parallel execution and contention for resources. Replication is limited to immutable objects in the proposed approach. Lifting this limitation seems hard without modifying the runtime system and without incurring a lot of extra overhead on each write operation. Furthermore, pre-written concurrent applications, designed to execute on a multiprocessor machine, are unlikely to achieve optimal performance in a distributed environment without extensive manual rewriting. Nevertheless, it is interesting to see if some well behaved applications are suitable for no-programming partitioning. It is also interesting to quantify the overhead of the approach relative to traditional Distributed Shared Memory systems.
- Explore the possibilities for application development with no-programming partitioning in mind. Although the approach is aimed at pre-written applications, perhaps it can yield great benefits when used simultaneously with the development of the application. In this way, the application writer will be shielded from distribution concerns, but all the testing of the application will be done in a distributed environment. Having distribution in mind when writing the application should reveal several opportunities for optimization. We will try to specify guidelines for application authors to help make their applications amenable to no-programming partitioning.

5. Impact of the Proposed Work

Ease of application development has emerged as a primary concern of the Computer Science community. Nevertheless, facilitating application development is a very hard problem, that has generally defied solution for many decades. The hope is now that select, domain-specific solutions can be developed, aiding in the development of particular classes of applications. No-programming partitioning is exactly one such solution, aiming to facilitate developing a class of distributed applications. As networking becomes ubiquitous and computing enters every field of life, this class of applications will only grow.

No-programming partitioning can reduce drastically the development time and effort required to deploy applications in a distributed environment. Additionally, no-programming partitioning can improve performance over traditional techniques that enable applications to accept remote input or produce remote output (e.g., X-Windows, Java applets, Java servlets). In some cases, no-programming partitioning may make the difference that will enable running the application in a distributed environment: traditional techniques may be too slow or heavyweight, and manual rewriting may be impossible or not cost-effective. For applications amenable to no-programming partitioning, the tedious details of programming for a distributed environment can be completely eliminated. This will enable application developers to concentrate on the more interesting aspects of distribution (e.g., handling partial failure) and produce higher-quality partitioned applications.

References Cited

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 Ben Charny, “Cell phone industry infiltrates JavaOne show”, Special to CNET News.com, June 4, 2001 http://news.cnet.com/news/0-1004-200-6163270.html .

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

7 M. Driver, “Where Are Java Programmers When You Need Them?”, Gartner Group research note, 4 April, 2000, http://gartner11.gartnerweb.com/public/static/hotc/hc00087599.html .

8 James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification, 2nd Ed.*, The Java Series, Addison-Wesley, 2000.

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

10 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.

11 Nelson King, “Partitioning Applications”, *DBMS and Internet Systems* magazine, May 1997. See http://www.dbmsmag.com/9705d13.html .

12 Donglin Liang and Mary Jean Harrold, “Efficient Points-to Analysis for Whole-Program Analysis”, in *7th ACM Symposium on Foundations of Software Engineering*, pp 199-215, Sept. 1999.

13 Christian Nester, Michael Phillipsen, and Bernhard Haumacher, “A More Efficient RMI for Java”, in Proc. *ACM Java Grande Conference*, 1999.

14 Object Management Group, “The Common Object Request Broker: Architecture and Specification, rev. 2.2”, Technical Report, February 1998.

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

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

17 Andre Spiegel, “Pangaea: An Automatic Distribution Front-End for Java”, 4th *IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '99)*, San Juan, Puerto Rico, April 1999.

18 Andre Spiegel, “Object Graph Analysis”, Technical Report B-99-11, FU Berlin, FB Mathematik und Informatik, July 1999.

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

20 Sun Microsystems, Remote Method Invocation Specification, http://java.sun.com/products/jdk/rmi/, 1997.

21 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.

22 Deepa Viswanathan, and Sheng Liang, “Java Virtual Machine Profiler Interface”, *IBM Systems Journal*, 39(1):82-95, 2000.

23 Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall, “A note on distributed computing”, Technical Report, Sun Microsystems Laboratories, SMLI TR-94-29, November 1994.

24 Cliff Young, Y. N. Lakshman, Tom Szymanski, John Reppy, David Presotto, Rob Pike, Girija Narlikar, Sape Mullender, and Eric Grosse, “Protium, and Infrastructure for Partitioned Applications”, *Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*. May 20—23, 2001, Schoss Elmau Germany, pp. 41-46, IEEE Computer Society Press, 2001.

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