Automatic Application Partitioning: The J-Orchestra approach

(position paper)

Eli Tilevich and Yannis Smaragdakis
Center for Experimental Research in Computer Science (CERCS), College of Computing
Georgia Tech
Atlanta, GA, 30332 USA
{tilevich, yannis}@cc.gatech.edu

Abstract. 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. Many distributed applications are created by partitioning their centralized versions. Traditional application partitioning entails re-coding the application functionality to use a middleware mechanism for communication between the different entities. This process is tedious and error-prone. Automating the partitioning process while preserving correctness and ensuring good performance of partitioned applications can greatly facilitate development of a large class of distributed applications. We review the main advantages and challenges of automatic application partitioning and present the J-Orchestra system. J-Orchestra is an automatic partitioning system for Java programs. J-Orchestra takes as input Java applications in bytecode format and transforms them into distributed applications, running on distinct Java Virtual Machines.

The present paper is a high-level supplement of our paper in the ECOOP 2002 technical program [12]. Here, we do not describe specific technical contributions, but instead we concentrate on the high level design decisions for an automatic partitioning system and argue that the J-Orchestra decisions make sense.

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 in 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, a traditional business 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 [9], or Java RMI [11] middleware). Perhaps, objects should migrate on demand, or according to an application-specific pattern.

Such circumstances give 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., [8]) as a way to use resources more efficiently. For instance, a scientific application, collecting data from several sensors,

storing the data, and processing them, can be partitioned so that the computer handling the data collection (sensor manager) is different from the computer storing the data (database manager) which in turn is different from the computer processing the data. This enables significant flexibility: the sensor manager could have little computing or storage capabilities, allowing it to be deployed more easily and in more varied environments (polar or space measurements are a good extreme example). At the same time, all three machines need to perform some processing of the data: the sensor manager may need to perform pre-processing to determine what data need to be stored. This pre-processing may result into elimination of redundant data, thus dropping the communication requirements. Similarly, the database manager will need to organize the data for fast retrieval.

Traditional application 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 order to partition the aforementioned scientific application, programmers have to identify the parts of code that correspond to each partition and understand how data are exchanged among different partitions. Then, re-coding of the application will need to be performed so that data are passed to remote sites instead of used in local calls. Integration with some middleware mechanism has to be done, and code has to be written that will simulate the results of local changes to the data using remote calls. The latter part is particularly challenging: local routines may often operate on the same data (through pointer aliases) and simulating this result across different memory spaces requires thorough understanding of the data dependencies among partitions.

As in all complex programming tasks, automation is highly valuable for application partitioning. In this paper, 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 *automatic* approach to application partitioning has significant potential. It can simplify drastically the process of partitioning applications. As application partitioning is becoming the main reason for distributed programming, the automatic partitioning approach can change the way many distributed applications are developed.

Our ideas are realized in the J-Orchestra[2] automatic partitioning system for Java programs. J-Orchestra takes a regular Java application in bytecode format and converts it into a distributed application. Regular objects become mobile objects and complex mobility scenarios can be specified. Serious correctness issues when dealing with unmodifiable code (e.g., code in the Java system classes) are addressed in novel ways. A thorough technical presentation of J-Orchestra can be found in Reference [12]. In this paper, we will not argue so much that J-Orchestra is technically interesting but that J-Orchestra is potentially very useful in practice. We will argue that the time for automatic partitioning is right and that J-Orchestra makes the right choices in order to maximize the potential impact.

---

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 automatic partitioning approach has nothing to do with concurrency discovery (e.g., work on automatic parallelization).
2. The name “J-Orchestra” suggests the analogy between partitioning centralized applications and the way orchestral pieces are often 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.

2. Can It Be Done and Does It Matter?

The main motivation questions about automatic partitioning are:

- Is there a real need for it?
- Can it be done correctly and efficiently for a large class of applications?

We believe that the need is there. The Internet has made most running programs part of a distributed system. Distributed computing is hard—any way to facilitate the development of distributed programs is desirable. The Ada 95 community, for instance, has long maintained that the Distributed Systems Annex of Ada 95 [7] represents a great advantage of the language. Using the Distributed Systems Annex, an Ada 95 user can develop a distributed application as a centralized Ada program and then partition it using external tools that do not modify the code—instead, the compiler transforms local procedure calls into remote calls and a different tool can assign objects to nodes without recompilation. (Even so, Ada 95 does not go nearly far enough because the centralized applications need to be written with distribution in mind. For instance, pointers are not supported, unless the user explicitly defines how pointer-based structures are marshalled and unmarshalled.)

Of course, it is utopian to expect that *all* applications can be distributed without code modifications and attain acceptable performance: typically a lot of human intelligence needs to be applied throughout the program code in order to get good performance over a high-latency/low-bandwidth medium. Nevertheless, there are good reasons to hope that the class of applications for which automatic 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. There is a conceptual similarity between an object interacting with its users through method calls and a server making its services available to its clients. This similarity often makes the conversion of objects into remote servers more straightforward. 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, significant ease of binary manipulation, and portability over different architectures. Therefore, the current increasing trend of writing applications in object-oriented languages with abstract runtime systems (like Java or C#) favors automatic 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.

In the spectrum of technologies aimed at facilitating distributed computing, automatic partitioning is among the most ambitious, because it imposes modest requirements. To elaborate this somewhat paradoxical statement, automatic partitioning is an ambitious approach on the technical front, but very modest on

the deployment front. Application partitioning is similar to a *distributed shared memory* approach (e.g., CJVM [1], Java/DSM [14]). The distinct element of application partitioning is that only the application changes—*no changes are allowed to the runtime environment where the applications are to be executed.* This offers significant deployment advantages, including 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).

3. Technical Issues and Design Choices

Our ultimate research objective is to advance automatic application partitioning to "industrial strength" levels, i.e., to the point where third-party, commercial applications can be partitioned and used successfully. The success criteria for our research will be whether J-Orchestra a) can handle the engineering complexity of commercial programs; b) enables convenient partitioning, where the user only needs to interact with a GUI for a few hours in order to partition a large application; c) achieves good performance for the resulting application.[3]

We will argue that J-Orchestra makes the right design decisions and represents a promising avenue to an industrial-strength automatic partitioning system.

The main elements of the J-Orchestra approach are as follows:

- the platform of experimentation is Java and we perform the partitioning through Java bytecode rewriting
- there is a test-case profiling phase for the application that will supply information to guide partitioning, placement, and mobility decisions
- there is a powerful rewriting engine allowing correct partitioning of more applications than prior approaches
- J-Orchestra generates source code proxies where failure handling code can be added by the user
- objects are able to move, whenever possible
- static analysis is used to enlarge the set of partitionings that are guaranteed to be correct and to perform optimizations
- a graphical interface presents the results of partitioning and static analysis to the user and allows the user to make distribution decisions
- J-Orchestra supplies heuristic algorithms for partitioning (data placement) based on data exchange information.

We examine some of these design decisions in detail.

Java Bytecodes as a Program Representation. The Java programming language [6] is the dominant language for Internet development and one of the most dominant programming languages overall. There is an 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 [5]). The accumulated Java expertise guarantees 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

3. Our attention is (for now) limited to the case where complex issues like crossing administrative domains, security, trust, routing, dynamic coordination, etc. can be handled orthogonally. This is not the case for all kinds of distributed computing, but it is a common scenario and it allows J-Orchestra to be simpler and more targeted.

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.). 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).

Graphical Front-End for User Interaction and Heuristics for Partitioning. The goal of our 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 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.

A General, Efficient Rewriting Engine that Allows Object Mobility. The most important part of an automatic partitioning approach is the rewriting it performs. The issues involved range from engineering considerations (supporting complex language features like inheritance, arrays, self-reference, etc.) to deep research problems (e.g., partitioning applications when they include unmodifiable code). The J-Orchestra rewriting algorithm is described in detail in [12]—here we will just sketch the main features.

Briefly, J-Orchestra allows turning local objects in the application into mobile objects. Static analysis ensures that the partitioning is correct by finding all dependencies between objects that would prevent them from being placed on different network sites. Such dependencies arise only from unmodifiable code (e.g., native code in the Java system classes). Modifiable objects (i.e., instances of regular application classes) can always be turned into mobile objects and migrate at will. Unmodifiable objects are always remotely accessible but cannot migrate. When unmodifiable objects are passed into unmodifiable code, they are dynamically "unwrapped" so they can be accessed like regular Java local objects. Similarly, objects are "wrapped" when they are passed out of unmodifiable code so that all other objects can refer to them from anywhere on the network. (Strictly speaking, object references are wrapped and unwrapped, not the objects themselves.) The result is a very general rewriting algorithm that automatically guarantees the correctness of partitioning and allows good performance through object mobility. No previous automatic partitioning system offers these features.

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 [13]). 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, J-Orchestra 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 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 J-Orchestra 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.

4. Questions to Resolve

The main goal of our research to evaluate whether automatic application partitioning can become an industrial-strength technique, scaling to large third-party applications and providing acceptable performance. We believe that J-Orchestra makes the right general design choices. Many issues still remain to be resolved, however.

- *Profiling*: what kind of profiling information can lead to choosing good object migration strategies? Can we automate the discovery of good mobility strategies based on profiling information?
- *Static analysis*: how can more advanced static analysis help with optimization? 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 automatic 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 automatic partitioning? Is automatic partitioning suitable to high-performance applications? How does it compare to traditional Distributed Shared Memory systems?

A large part of our work has to do with evaluating the impact of automatic partitioning. Although not all applications can be partitioned automatically, it will be beneficial to have a classification of applications that are amenable to automatic partitioning.

Some concrete ideas in these directions follow:

- Naturally distributed environments, such as embedded systems, offer a promising domain for automatic partitioning. Consider 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 [4].) Ease of application development is paramount, as the applications are not developed by systems experts. Therefore, automatic partitioning 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.

- J-Orchestra is ideal for interactive applications that need to 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.

  We show a simple comparison of J-Orchestra partitioned applications with X-Windows output redirection in [12]. J-Orchestra can get better performance than X-Windows by placing the graphics-generating code on the same machine as the graphical output screen.

- Traditional Distributed Shared Memory systems have concentrated on high-performance applications and issues with parallel execution and contention for resources. J-Orchestra is not ideal for this domain because it is hard to support replication with good performance in an automatic partitioning approach. DSM systems often take advantage of virtual memory hardware mechanisms or highly optimized code in order to implement consistency protocols without slowing down local operations. In an automatic partitioning approach, where the runtime system cannot change, these solutions are not applicable. Nevertheless, it is interesting to see if some well behaved applications are suitable for automatic partitioning. It is also interesting to quantify the overhead of the approach relative to traditional Distributed Shared Memory systems.

- We should establish guidelines for application development with automatic partitioning in mind. J-Orchestra can yield great benefits when used simultaneously with the development of the application and not only after the complete centralized application has been completely developed. 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.

5. Summary and Conclusions

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. Distributed applications, where the distribution is dictated by functional constraints, constitute a domain amenable to partial automation. Such distributed applications can be developed from centralized applications using an automatic partitioning approach. As networking becomes ubiquitous and computing enters every field of life, the importance of automatic partitioning can only grow.

Automatic partitioning can reduce drastically the development time and effort required to deploy applications in a distributed environment. Additionally, automatic 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, automatic 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 automatic 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

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 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`.

5 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`.

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

7 ISO, *Information Technology-Programming Languages-Ada*, ISO standard, Feb. 1995, ISO/IEC/ANSI 8652:1995.

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

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

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

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

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

13 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.

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