Automatic Partitioning: A Promising Approach to Prototyping Ubiquitous Computing Applications

Nikitas Liogkas
University of California—Los Angeles

Blair MacIntyre, Elizabeth D. Mynatt, Yannis Smaragdakis, Eli Tilevich, Stephen Voida
Georgia Institute of Technology

Submitted to *IEEE Pervasive Computing*,
Special Issue on Building and Evaluating Ubiquitous System Software
Submission date: 15 March 2004

Abstract. One of the main challenges facing ubiquitous computing research and development is the difficulty of writing software for complex, heterogeneous distributed applications. In this paper, we evaluate automatic application partitioning as an approach to rapid prototyping of ubiquitous computing systems. Our approach allows developers to largely ignore distribution issues when developing their applications, by providing tools for generating distribution code automatically, under user guidance. We claim that automatic partitioning is promising for a large class of ubiquitous computing applications and discuss an example ubicomp application re-engineered using our approach.

Keywords: ubiquitous computing; distributed systems programming; automatic partitioning

The software engineering goal of removing obstacles to human creativity is one of the main challenges in several areas of computing. One area in which the need for software engineering support has been clearly identified[1] is that of *ubiquitous computing.*[2] Proponents of ubiquitous computing envision a future in which computers are cheap and plentiful, and can be used together effortlessly. Unfortunately, while hardware continues to become smaller and less expensive, the corresponding software tools that would make the vision of ubicomp possible have not matured at the same rate. In particular, few languages and tools are available for exploratory programming of distributed interactive ubicomp applications.

One major feature of the ubicomp domain (distinguishing it from, for example, traditional desktop applications) is the inherent distributed nature of the software. Ubiquitous computing environments are naturally distributed over multiple computers, connected via a wired or wireless network. These computers come in many shapes and sizes, from hand-held to wall-sized,[2] and applications are typically designed under the assumption that computing resources come and go in ever-changing combinations of light- and heavyweight, predefined and ad-hoc groups. Thus, developers of ubicomp applications typically have to suffer all the complexities of distributed systems programming.

The engineering difficulties encountered when building ubicomp applications are more pronounced during research and prototype development. Ubicomp application prototypes are typically exploratory in nature: the structure of the application, the kind of data being shared, and the distribution characteristics of these data will all be modified frequently as the application undergoes iterations through the design-build-deploy-evaluate-redesign cycle. In order to facilitate rapid application prototyping in this domain, developers need to be able to modify the underlying distribution characteristics of the data structures with very little effort. Unfortunately, more often than not, the developers of ubiquitous computing applications are not distributed systems experts. As a result, ubicomp researchers need simple, automated techniques that support rapid prototyping of applications in such domains.

In this paper, we report on an evaluation of *automatic application partitioning* as a technology for developing and deploying an important class of ubiquitous computing applications—those that are distributed to best take advantage of unique or unusual computing resources. Automatic partitioning is the process of adding distribution capabilities to an existing centralized application without needing to rewrite the application's source code or needing to modify existing runtime systems (OSes or virtual machines). An automatic partitioning system allows users to describe the location of system resources, and performs a rewrite of the application binaries to introduce appropriate distribution mechanisms. This contrasts sharply with traditional distribution middleware, which automates the mechanics of distributed communication, but requires the application designer to explicitly encode decisions about the distribution structure of the application in the source code itself.

We believe automatic partitioning has the potential to greatly simplify the development of ubicomp applications: no distributed systems programming is required; applications can easily be re-partitioned and re-deployed without modifying the source code; and the applications run on standard run-time environments, enabling easy deployment in a variety of devices. For example, it has been relatively easy for us to take straightforward centralized Java programs, partition them, and deploy them on any device supporting a Java VM, including PDAs and computers of many different sizes and architectures. Nevertheless, the approach has some drawbacks that limit its general applicability to ubicomp development: there is limited flexibility with respect to the communications/middleware mechanisms that the application may use (for instance, it may not be possible for the system to be reconfigured dynamically); it is not easy to introduce fault-tolerance, unless the partitioning infrastructure supports it; and the communication patterns may be limited by the structure of the application. These problems present exciting research challenges which would not have been apparent without attempting to apply this nascent technology to a realistic ubicomp application. Overall, we found the value of automatic partitioning to be clear for ubicomp application prototyping tasks, but not yet mature enough for final system development.

Motivation

Consider the traditional approaches to evolving a regular, centralized application into a distributed one. The programmer typically employs conventional middleware (e.g. CORBA, DCOM, Java RMI) in order to allow different program entities to communicate with each other. However, middleware programming has many complications. Although traditional middleware mechanisms typically implement a Remote Procedure Call (RPC) paradigm, remote procedures do not behave the same as local procedures. For example, whereas a local procedure call can accept arguments by-reference (i.e. the client and server share data through a pointer), a remote procedure call usually supports by-copy semantics (i.e. the client and server work on two different copies of the data). Similarly, the implementation of several familiar actions is radically different—e.g. object construction needs to happen through calls to a remote factory; such objects need to be registered with a special service to be remotely accessible; synchronization of threads over a network requires extra support not offered directly by middleware; and garbage-collected languages (e.g., Java) do not fully support distributed garbage collection. In general, a programmer needs to perform a large number of changes to distribute an existing application, and most of these changes require thorough knowledge of the application structure. Application evolution may also require major changes in distribution implementation: new objects may need to become remotely accessible, some data structures may become too large to be copied and may need to be split, etc.

Automatic Application Partitioning is an approach that makes distribution transparent. Additionally, the approach *does not require changes to the runtime system.* This goal is the fundamental distinction between automatic partitioning and Distributed Shared Memory (DSM) systems. The distinction is crucial in the ubicomp domain. Supporting exploratory programming of multiple, diverse, distributed devices is very hard when a specialized runtime system needs to be deployed together with the application. For instance, it is much easier to deploy an application with a standard Java VM (which supports a variety of third-party libraries and can often be found pre-compiled for handheld devices such as PDAs and cell phones) than with a specialized VM that supports distribution. Instead of having specialized runtimes, automatic partitioning only modifies the application binaries, essentially imitating many of the changes that a human programmer would have to perform by hand. Several aspects of distribution (and especially correctness aspects—e.g. maintaining by-reference semantics over the network, enabling correct distributed synchronization for the partitioned application, etc.) can be handled automatically by the partitioning tool in use.

Automatic Partitioning and J-Orchestra

The main idea of automatic partitioning is quite simple: an automatic tool takes as input a regular program and user-supplied location information for the program's data and code. The tool re-writes the program so that the code and data are divided into parts that can be run in the desired locations. Any data exchange between parts of the program on different locations automatically becomes remote communication.

This simple idea results in significant complications for realistic programs with data shared through pointers. Since centralized programs are written to assume a single, shared address space, the same abstraction must be maintained over a network. Data that are shared through pointers in the centralized version must continue to be shared in the distributed program. Therefore, many pointers (or "references" in Java) need to be transformed into indirect references—i.e. references to a proxy object—that could be pointing either to a local object or to an object over the network. Additionally, many other transformations need to take place. To name a few (we use the Java language in all our examples):

- Access to fields of other objects (e.g. `obj.field = new_val`) should be transformed to method calls (e.g. `obj.set_field(new_val)`).
- Constructor calls should be transformed to calls of a factory method.
- References to Java system objects may need to become references to special wrapper objects that are remotely accessible (through normal proxies).
- Synchronization requests need to be transmitted over the network. Thread identity should be preserved over remote calls.

To complicate matters even more, not all references in an application can be transformed into indirect (proxy) references. Some references have to remain direct because the code manipulating them cannot be modified. For example, a data type representing a disk file can be accessed by code inside the runtime system (in our case, the Java VM) as well as application code. The application code holds a reference to the file data type and passes this reference to the VM whenever a file-related task needs to be performed. Since the VM code cannot be modified, the VM reference to the file data type must remain direct. Furthermore, if files need to be used on two separate partitions, there is no guarantee that the partitioned application will behave in the same way as the centralized version. Without knowledge of the semantics of the partitioned application, there is no way to tell if file operations in the two partitions are distinct. Therefore, much of the complexity of automatic partitioning systems is due to such issues of dealing with unmodifiable code.[3,4,5]

The J-Orchestra automatic partitioning system that we discuss in this paper is a state-of-the-art partitioning system in terms of sophistication and scalability.[4,5] J-Orchestra is completely GUI-based, works on Java programs, and performs all transformations at the bytecode level. The user of J-Orchestra sees a view of all the classes (both application-level classes and Java system classes) involved in an application. The user's input consists of assigning groups of classes to network sites. The system then rewrites the application code to effect the partitioning. J-Orchestra's partitioning does not need to modify either the JVM or its runtime (JRE) classes, making deployment easy to manage. J-Orchestra is the first system to effectively address the problems of dealing with unmodifiable code (references that must remain direct) for industrial-strength applications. The solution consists of an analysis algorithm that tries to determine heuristically what references leak to unmodifiable code, and a sophisticated rewrite algorithm that injects code transforming indirect references to direct references (and vice-versa) at run-time. The role of the analysis algorithm is strictly advisory: the user can override analysis results and guide the J-Orchestra rewrite at will. The results of the analysis algorithm are reflected in the GUI as groupings of co-dependent classes.The user can override these restrictions and place the classes on different machines. Thus, the user of J-Orchestra manipulates the partitioned application at the level of individual classes or groups of classes (where the groupings are computed automatically). This yields a high degree of automation. J-Orchestra has been used to partition third-party industrial applications without any knowledge of their internals.

Despite its capabilities, the automatic partitioning approach is not a magic wand that removes all difficulties of implementing distributed systems. The main feature (and restriction) of automatic partitioning is that *the logic and*

structure of the distributed application remain identical to those of the centralized application. This limits the applicability of the approach. For example:

- Partitioned applications should have very clear communication and locality patterns. Since the application logic will remain the same, a large number of remote accesses will be detrimental to performance.

- Objects shared among partitions should not be used by unmodifiable code (e.g. OS or JVM code). Otherwise, the application structure needs to change for partitioning to be possible.

- The resulting distributed application should primarily have synchronous communication patterns. If good performance or reliability requires asynchronous communication, the application structure needs to change.

These and other limitations make automatic partitioning particularly well suited for what can be roughly described as *resource-driven distribution*. This is the case when the application has distinct parts, each dealing with different hardware or software resources that may exist throughout a network. For example, machines scattered around the network might each possess unique resources such as a large graphical display screen, high-quality speakers, a digital camera, etc. A centralized application, written without any distribution in mind, might want to access such resources located on one or more remote machines. As a specific example of resource-driven distribution, a user may decide to partition a sound application to be controlled and monitored remotely from the machine where the sound is actually produced or processed.

The Ubicomp Domain and Automatic Partitioning

Mark Weiser's original vision of computers "disappearing" and being integrated "seamlessly into the world" has always relied on leveraging a variety of computing form factors and connecting those devices—and the applications that run on them—using wired, and wireless, networks.[2] Thus, ubicomp systems are naturally distributed because they integrate many devices: sensors, displays, storage, etc.This is exactly the resource-driven kind of distribution that automatic partitioning excels at.

In many cases, the different parts of a ubicomp application are loosely coupled. Although network communication can be a bottleneck, most successful applications of automatic partitioning[4,5] achieve high performance for loosely coupled applications by putting the code near the resource it accesses.

Additionally, ubicomp systems place a high premium on flexibility and configurability. Distribution decisions may change multiple times over the lifetime of the system. For example, even if a ubicomp application was originally designed to have sound processed on the same machine as some other part of the computation, a later version may need to change this assumption. Components should be able to be used together effortlessly and in a variety of configurations. Ease of programming by non-experts is paramount in ubicomp systems; hence, the low-effort distribution of automatic partitioning is highly desirable.

We have used the J-Orchestra infrastructure to partition multiple Java applications and deploy them on diverse platforms. A common case is that of taking a straightforward centralized Java program, partitioning it, and deploying it on many small mobile devices that communicate with a central server or a laptop machine. Examples that we commonly use for demonstration purposes include:

- A demo application where GUI actions cause the production of synthesized speech: speech is produced on a central machine while the application GUI is running on a handheld (iPAQ).

- A smart controller for our PowerPoint presentations: we have written a small Java GUI application that controls MS PowerPoint through its COM interface. We partitioned this application into a GUI and a back-end part. We run the GUI on a Linux PDA equipped with a wireless card and use it to control PowerPoint running on a Windows laptop. We have given multiple presentations using this tool.

- A remote load monitoring application: machine load statistics are collected and filtered locally with all the results forwarded to a handheld (iPAQ) machine over a wireless connection and displayed graphically. The original application was written to run on a single Windows machine.

A Case Study: Kimura

As a larger case study of applying automatic partitioning to ubiquitous computing systems, we have used automatic partitioning in the development of the latest version of the Kimura system[6,7]—a realistic, complex ubicomp application. Kimura is part of a research project that seeks to explore and evaluate the addition of visual peripheral displays to human-computer interfaces. It uses large, projected displays as peripheral interfaces to complement an existing focal work area (i.e. the area surrounding a traditional desktop computer). It effectively utilizes peripheral displays to assist users in managing multiple activities—coherent sets of tasks typically involving the use of multiple documents, tools, and communications with others. Background activities are visualized on the peripheral displays as montages of images captured in desktop computer activity logs. Additionally, the montages serve as anchors for background awareness information collected from a context-aware infrastructure.

Kimura's source code consists of 98 Java application classes and over 4,400 non-comment source statements. These application classes use a large number of system and third party classes, including Swing and Java Advanced Imaging (JAI) library classes, as well as classes that facilitate two-way communication with an electronic whiteboard.

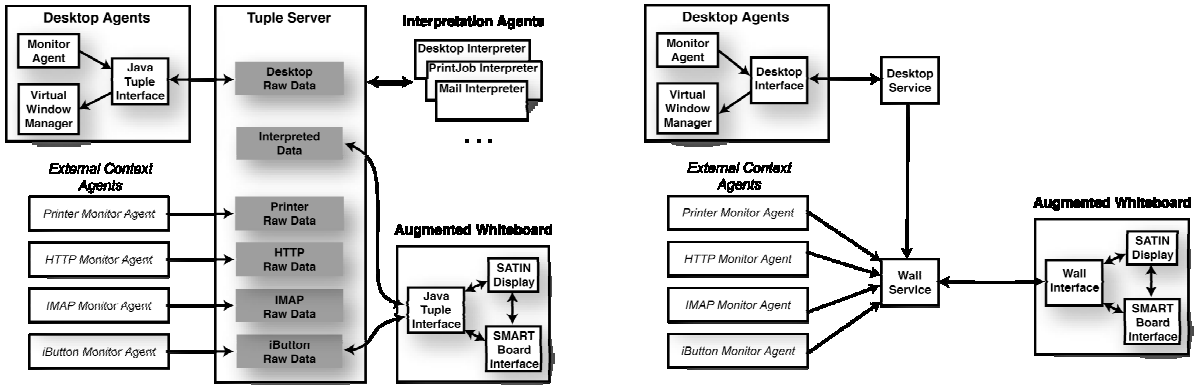

Figure 1. (a) The architecture of the original Kimura system. (*left*)
Figure 1. (b) The architecture of the re-engineered Kimura2 system. (*right*)

The architecture of the original version of Kimura, shown in **Figure 1a**, is structured around three distinct components. A desktop interface module runs on the user's PC, monitoring all window and application activity through a native library and providing virtual desktop functionality. A context interpreter module acts as an intermediate layer, aggregating the incoming messages from the desktop and the context-aware infrastructure (providing email, printer, and location awareness services) and conveying them to the peripheral display module (informally, "the wall"). The wall is directly connected to several projectors and a SMARTBoard device, and is responsible for maintaining two-way communication with the whiteboard and providing up-to-date visualizations of the user's working contexts as montages projected on the SMARTBoard surface.

These three components in the original version of Kimura are connected together using TSpaces[8], a communication package designed to connect distinct distributed components. It is based on the well-known tuplespace paradigm, and incorporates database features such as transactions, persistent data, and flexible queries. TSpaces employs the publish-subscribe model: when one component adds or deletes a tuple on the TSpaces server, an appropriate callback method is called asynchronously in any other component that has registered to receive notifications matching that type of tuple. The creators of TSpaces aimed at "hitting the distributed computing sweet spot"[8]: the system allows programmers to ignore many of the hard aspects of distributed communication, such as naming, state (and persistent storage), and load balancing. The Kimura implementation did not use any of these

advanced features of TSpaces, but employed it as a convenient way to keep shared state and broadcast global events (e.g. activity changes) to all system components.

Developing Kimura2

In order to evaluate the applicability of automatic partitioning to the ubicomp domain, we re-engineered Kimura by removing the existing distribution code and re-distributing by automatic partitioning. The first step in the re-engineering process was to separate Kimura's main application tasks from its network communication. In this way, we could derive a simpler Kimura core, evolve it as necessary and automatically partition it with J-Orchestra. We first removed the code that supported distribution with TSpaces, and replaced it with a single shared data structure. The result was a single program that runs in one process and opens multiple windows (the "wall" and the desktop control panel) on a single machine. The TSpaces-related code—functions responsible for connecting to the TSpaces server and adding or deleting tuples—was spread over 11 of the 77 source files. While TSpaces dictated an event-based structure for the application, the centralized version could use direct method calls between components, resulting in simpler and cleaner code. Similarly, the interpreter component, which acted as an extra level of indirection between the desktop and the wall, was superfluous in the centralized version. We removed it as a distinct entity, preserving its functionality in two new modules that act as public interfaces of the desktop and the wall, respectively. These two new modules are essentially two singleton classes whose responsibilities are to handle incoming and outgoing messages from the other part of the application. Their coding and integration with the rest of the system was straightforward. The architecture of Kimura2 can be seen in Figure 1b. As illustrated, there is no longer a central server in the new version. Instead, the system components talk to one another directly and synchronously.

The partitioned version of Kimura2 consists of two partitions: one for the desktop and one for the peripheral display. The user interaction takes place through the peripheral display, while the desktop machine does the core of the processing (e.g. monitoring open applications). The peripheral display can be thought of as a "monitoring console" for the Kimura working environment. Altogether, out of the 64 classes automatically rewritten, 42 were Swing/AWT classes, and 6 were made Serializable so that they could be passed by-copy across different memory spaces, resulting in improved performance. 71 Kimura application, 4 third-party, and 12 JDK classes were excluded from the distribution process altogether as we determined (using an automatic heuristic analysis) that they never participate in the distributed communication. All in all, including testing, it took the programmers a few days to partition Kimura2 with J-Orchestra.

Benefits

Automatic partitioning turned out to be quite beneficial in the case of developing Kimura2. The main benefit is in the simplicity of the new software architecture, resulting in more understandable and maintainable code, without sacrificing any of the original functionality. The architecture of Kimura2 will enable planned additions to the system much more easily because the developers can focus on the desired functionality without having to worry about the distribution specifics. Furthermore, the new version is easier to deploy: we avoid the need to maintain a running TSpaces server.

To quantify the simplicity benefits of Kimura2, we used a standard tool (JStyle 5 by Codework—http://www.codework.com/JStyle/product.html) to derive software metrics. The Software Engineering community is still divided on the value and meaning of software metrics; thus, the significance of our qualitative findings is subject to some degree of individual interpretation. We list below some of the more pronounced differences between the original Kimura and Kimura2. The new version exhibited better results under all metrics, including those not described in detail here.

The original application consisted of 4,436 source statements (including declarations, but not counting comments, empty statements, empty blocks, closing brackets or method signatures). Out of them, 3,836 (86% of the total) remained unchanged in the new version. The TSpaces-related code (486 statements, almost 11% of the total) was removed completely, and 134 statements were added. Finally, 114 statements were modified to adapt the application to the new communication paradigm.

Table 1. Software complexity metrics

	original version	new version	% more in original
total statements	4436	4084	8.6
number of classes	98	92	6.5
number of methods	693	682	1.6
program difficulty	3305	3124	5.8
development effort	2611	2235	16.8
LCOM	2395	2165	10.6
inter-package fan-out	881	822	7.2

The new version exhibits significant differences using the Halstead program difficulty metric,[9] Chidamber and Kemerer's *Lack of Cohesion of Methods* (LCOM),[10] and class fan-out (the number of classes a given class depends on). The summative values of these metrics can be seen in Table 1. The new version is significantly less complex than the original one. Of course, it is expected that a centralized architecture would be much less complex than a distributed one. However, it is interesting to quantify the difference.

In our evaluation of Kimura2, we also performed extensive measurements to evaluate the performance impact of the partitioning infrastructure. Most system operations (montage creation, montage switching, document manipulation, etc.) exhibited significant speedup relative to their counterparts in the original version, with only two of the measured operations (wall montage switching, document activation) showing a slowdown. We omit our performance measurements since they are not essential to our conceptual evaluation of automatic partitioning for ubicomp: they are merely the result of orthogonal, low-level concerns, such as the underlying middleware used in the case of J-Orchestra relative to TSpaces.

Limitations and Discussion

Our experiences of using automatic partitioning to develop ubicomp applications have been quite positive. The overwhelming advantages of the approach include the simplicity of coding for a single machine without the need for distributed programming, and the ease of re-partitioning and re-deployment. Furthermore, the ability to run on unmodified run-time systems (i.e., any Java VM) is invaluable when using a multitude of heterogeneous devices. Nevertheless, we have also identified several shortcomings of automatic partitioning in the context of ubicomp. Although some of them arguably result from limitations of the current state-of-the-art, we try to distance ourselves and to identify the general engineering issues that are difficult to address in an automated way. Note that we explicitly distinguish between "automatic" approaches, like ours, and "semi-automatic" ones. Recall that the J-Orchestra user works at the class or group-of-classes level of abstraction. Thus, our approach is quite automatic and involves no programming, just resource location assignment (e.g., graphics code should run on this machine, the main engine should run on that machine, etc.). In contrast, a semi-automatic approach could let the user annotate detailed parts of the code and data, e.g., to indicate what data should be replicated and where, how the copies remain consistent, where leases are used for fault tolerance, etc. Many of the issues with automatic partitioning can be resolved with a semi-automatic approach.

First, we should point out that automatic partitioning is not a naïve end-user technique. Often automatic partitioning requires an understanding of the application's internal structure. For instance, J-Orchestra could not have partitioned Kimura2 without knowledge of its internals because Kimura2 uses Swing UI classes on what would become the wall and the desktop partitions. As discussed earlier, a major difficulty with automatic partitioning is dealing with unmodifiable code, such as the native Swing UI libraries in the Java runtime. Since the code handling these objects is unmodifiable, we need to be sure that the objects in one partition are not shared in the other; otherwise, the Swing code may try to access fields of a remote object directly, resulting in a crash. The heuristic analysis that J-Orchestra uses for determining what references can leak to what code conservatively determines that Swing classes cannot exist on two different partitions. However, we know that the Swing object partitioning in

Kimura2 is safe (i.e., that the Swing widgets on the desktop display are distinct from the Swing widgets on the wall display). Thus, we can explicitly direct J-Orchestra to produce appropriate code for Swing classes on both partitions.

Another issue with automatic partitioning in the context of ubicomp is that it does not offer any assistance in the problem of highly dynamic interactions between communicating entities. One of the common features of ubicomp applications is allowing for resources and services to come and go dynamically as users and devices enter and leave the environment. Since automatic partitioning does not change the logic or structure of the original centralized application, flexibility and configurability must be designed into the original application before it is partitioned. Nevertheless, although dynamic interactions cannot be supported by automatic modification of an unsuspecting application, they can be supported semi-automatically. A system offering tool support for ubiquitous computing development can let the user annotate the application code to express desired policies for data consistency under possible failures. These annotations form a domain-specific language for specifying properties of dynamic distribution. For instance, a certain data field can be annotated to indicate that there will be dynamically many instances of it. Another annotation can specify the leases that each client holds and the data that depend on each lease. The low-level code will then be generated from the annotations instead of having to be written by hand.

A similar observation holds regarding concurrency: automatic partitioning does not provide automatic parallelization. If the original application is single-threaded, the partitioned application will remain single-threaded: separate threads will exist on each machine, but only one of them will be active at any time. Of course, we can sometimes duplicate clients in identical configurations (i.e. replicate a partition on multiple devices). Nonetheless, every remote method remains single-threaded and multiple incoming remote calls will be queued and serviced in order. Fortunately, this is sufficient for many common communication patterns. Furthermore, most interactive Java programs make heavy use of threads when accessing resources and handling interactivity, due to the design of the Java run-time libraries. When such programs are distributed using automatic partitioning, the concurrency is maintained correctly (although remote transitions are more costly). A common scenario, however, is that in which threads execute almost entirely within one partition and handle distinct resources (disk files, sound devices, graphical interactions, CPU processing, etc.); this model is ideal for automatic partitioning.

In general, a partitioning system tries to automate many hard distribution tasks. Any automation effort, however, hinders complete control for users with advanced requirements. In ubicomp development such requirements may include replication for fault-tolerance; high-performance through load-balancing, caching, or asynchronous communication; security; persistence; and more. For instance, in an automatically partitioned application, it is not easy to use replication for redundancy and switch to a different server once a failure is detected. The conventional wisdom in the distributed systems community is that mechanisms for handling distributed failure are extremely application-specific and cannot be automated completely. Again, the appropriate solution may be to follow a semi-automated approach, providing tool support for replication, load-balancing, security, etc. In this way, the programmer will be relieved of the low-level complexity, but will still be responsible for annotating parts of the code in detail and for the conceptual consistency of the distribution, unlike in a fully automated approach. An example of the semi-automated approach is already supported by J-Orchestra: the user can enable complex schemes for object mobility (e.g., “move this object whenever it is reachable from an argument of a remote method”). Nevertheless, this is not a GUI-accessible feature. Instead, the user needs to write Java code that follows conventions of the J-Orchestra framework in order to enable such object mobility.

Because of these observations, we believe that the benefits of fully automatic partitioning for ubicomp are highest during prototyping. During prototyping, the benefits of avoiding distributed systems coding while being able to experiment with different partitioning schemes outweigh the potential inflexibility of the communication mechanisms. In contrast, the applicability of automatic partitioning for creating mature, deployable applications can vary substantially. If handling tough distribution issues (e.g. asynchrony, fault-tolerance, or load balancing) is essential for an application, the best option continues to be the use of a flexible middleware technology and a program design that exerts full control over the application structure. Further work is required to determine the ideal balance between automation and power in ubicomp development tools, but we believe that our study illuminates interesting aspects in the design spectrum.

Acknowledgments

This work was supported by NSF grants IIS-9988712, CCR-0238289 and CCR-0220248, and by a seed grant from the Georgia Tech College of Computing. Nikitas Liogkas was also supported by a graduate fellowship by the Foundation “Lilian Voudouri.”

References

1. Abowd, G., “Programming Environments…literally. Ubicomp’s Grand Challenge for Software Engineering”, *ACM SIGSOFT 2002/FSE-10* Keynote Presentation.

2. Weiser, M., “The Computer for the 21st Century”, *Scientific American*, *265*(3): 94-104, September 1991.

3. Tatsubori, M., Sasaki, T., Chiba, S., and Itano, K., “A Bytecode Translator for Distributed Execution of ‘Legacy’ Java Software”, *European Conference on Object-Oriented Programming (ECOOP)*, June 2001.

4. Tilevich, E., and Smaragdakis, Y., “Automatic Application Partitioning: The J-Orchestra Approach”, *8th ECOOP workshop on Mobile Object Systems*, 2002. See http://j-orchestra.org.

5. Tilevich, E., and Smaragdakis, Y., “J-Orchestra: Automatic Java Application Partitioning”, *European Conference on Object-Oriented Programming (ECOOP*), 2002.

6. MacIntyre, B., Mynatt, E.D., Voida, S., Hansen, K.M., Tullio J., and Corso, G.M., "Support for multitasking and background awareness using interactive peripheral displays", in *Proc. of the ACM Symposium on User Interface Software and Technology (UIST ’01)*, ACM Press, 2001, pp. 41-50.

7. Voida, S., Mynatt, E. D., MacIntyre, B., and Corso, G. M., “Integrating virtual and physical context to support knowledge workers”, *IEEE Pervasive Computing*, *1*(3): 73-79, July-September 2002.

8. Lehman, T.J., Cozzi, A., Xiong, Y., Gottschalk, J., Vasudevan, V., Landis, S., Davis, P., Khavar, B., and Bowman, P., “Hitting the distributed computing sweet spot with TSpaces”, *Computer Networks*, *35*(2001): 457-472.

9. Halstead, M. H., *Elements of Software Science*, Operating and Programming Systems Series, Volume 7, 1977.

10. Chidamber, S.E. and Kemerer, C.F., “A metrics suite for object-oriented design,” *IEEE Transactions on Software Engineering*, *20*(6): 476-493, 1994.

The Authors

Nikitas Liogkas is a PhD student in the Computer Science department at the University of California - Los Angeles. His research interests include computer systems software, focusing on techniques to make such software more maintainable and easier to reason about.

Blair MacIntyre is an Assistant Professor in the College of Computing and the GVU Center at the Georgia Institute of Technology. His research interests include understanding how to create highly interactive augmented reality environments, especially those that use personal displays to directly augment a user's perception of his or her environment.

Elizabeth Mynatt is an Associate Professor in the College of Computing and the GVU Center at the Georgia Institute of Technology. She directs the research program in Everyday Computing within the Future Computing Environments Group, examining the implications of having computation continuously present in many aspects of everyday life.

Yannis Smaragdakis is an Assistant Professor in the College of Computing at the Georgia Institute of Technology. His research interests include object-oriented programming languages, systems software, and program generators.

Eli Tilevich is a PhD Candidate in the College of Computing at the Georgia Institute of Technology. His dissertation research deals with the issues of developing better programming language tools for distributed computing.

Stephen Voida is a PhD student in the College of Computing at the Georgia Institute of Technology. His research interests include ubiquitous computing, technology in the workplace, and augmented environments.