Automatic Partitioning for Prototyping Ubiquitous Computing Applications

A major challenge facing ubiquitous computing R&D is the difficulty of writing software for complex, distributed applications. Automatic application partitioning can help development teams rapidly prototype distributed ubiquitous computing systems.

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

The software-engineering goal of removing obstacles to human creativity is a primary challenge in several areas of computing research. In particular, ubiquitous computing is one area in which researchers have clearly identified the need for software-engineering support.[1,2] Proponents of ubicomp envision a future in which computers are inexpensive and plentiful and seamlessly interoperate. Unfortunately, although hardware continues to become smaller and less expensive, the corresponding software tools that would make the vision of ubicomp possible haven't matured at the same rate. Few languages and tools are available for exploratory programming.

One major feature of the ubicomp domain—distinguishing it from traditional desktop applications, for example—is the software's inherent distributed nature. Ubicomp environments are naturally distributed over multiple computers connected via a wired or wireless network. These computers come in many shapes and sizes, from handheld to wall-sized. Applications in these environments are typically designed under the assumption that computing resources come and go in ever-changing combinations of lightweight and heavyweight, predefined and ad hoc groups. So, ubicomp application developers typically must suffer all the complexities of distributed systems programming.

The difficulties that developers encounter when building ubicomp applications are more pronounced during research and prototype development. Ubicomp application prototypes are typically exploratory: The application's structure, the kind of data being shared, and the data's distribution characteristics will change frequently as the application undergoes iterations through the design-build-deploy-evaluate-redesign cycle. To facilitate application prototyping in this domain, developers must be able to modify the data structures' underlying distribution characteristics with little effort. Unfortunately, ubicomp developers often aren't expert at distributed systems. As a result, ubicomp researchers need simple, automated techniques that support rapid prototyping in such domains.

Automatic application partitioning is a technology for developing and deploying an important class of ubiquitous computing applications—those that are distributed to 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 exist-

Published by the IEEE CS and IEEE ComSoc ■ 1536-1268/04/$20.00 © 2004 IEEE

ing runtime systems. 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 technique 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.

Motivation and challenges

Consider the traditional approaches to evolving a regular, centralized application into a distributed one. The programmer typically employs conventional middleware—such as CORBA, DCOM (Distributed Component Object Model), or Java Remote Method Invocation—so that program entities can communicate with each other. However, middleware programming has many complications. Although traditional middleware mechanisms typically implement a remote-procedure-call paradigm, RPCs don't behave the same as local calls.

For example, a local procedure call can accept by-reference arguments, but an RPC usually supports by-copy semantics. That is, the client and server work on two different copies of the data. Similarly, the implementation of several familiar actions is radically different—for example, 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 (for example, Java) don't fully support distributed garbage collection. In general, a programmer must perform many changes to distribute an existing application. Most of these changes require thorough knowledge of the application structure. An application evolution might also require major changes in distribution implementation. For example, new objects might have to become remotely accessible, or some data structures might become too large to be copied and might therefore have to be split.

Automatic application partitioning makes distribution transparent and doesn't require changes to the runtime system, which is the fundamental difference from distributed-shared-memory systems. The distinction here is especially crucial in the ubicomp domain. Supporting exploratory programming is very hard when a specialized runtime system must be deployed together with the application. For instance, it's much easier to deploy an application with a standard Java VM (which supports a variety of third-party libraries and can often be found precompiled 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 changes that a human programmer would have to perform by hand. The partitioning tool can handle several aspects of distribution automatically.

The main idea informing automatic partitioning is simple: An automatic tool takes as input a regular program and user-supplied location information for the program's data and code; the tool rewrites the program so that the code and data divide into parts that can run in the desired locations. Any data exchange between parts of the program at different locations automatically becomes remote communication. This simple idea produces significant complications for realistic programs with data shared through pointers.

Because centralized programs assume a single, shared address space, the same abstraction must be maintained over a network. Data shared through pointers in the centralized version must continue to be shared in the distributed program. Many pointers—or "references" in Java—must be transformed into indirect references (that is, references to a "proxy" object) that could point either to local objects or to objects over the network. We use Java in all examples, which include the following transformations that must take place:

- Access to fields of other objects must be transformed to method calls.
- Constructor calls must be transformed to calls of a factory method.
- References to Java objects might have to become references to special wrapper objects that are remotely accessible.
- Synchronization requests must be transmitted over the network while maintaining thread identity over remote calls.

To complicate matters further, not all references in an application can be transformed into indirect proxy references. Some references must remain direct because the code manipulating them can't be modified. For example, a data type representing a disk file could be accessed by code inside the runtime system as well as by 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.

Because the VM code can't be modified, the VM reference to the file data type must remain direct. If files must be used on two separate partitions, there's no guarantee that the partitioned application will behave in the same way as the centralized version. Without knowledge of the partitioned application's semantics, we have no way to tell whether file operations in the two partitions are distinct.

Much of the complexity associated with automatic partitioning systems is related to dealing with unmodifiable code.[3–5]

Automatic partitioning with J-Orchestra

J-Orchestra, the automatic-partitioning system we've developed, is state-of-the-art in terms of sophistication and scalability. It's completely GUI-based, works on Java, and performs all transformations at the bytecode level. The J-Orchestra user 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 doesn't need to modify either the JVM or its runtime classes.

J-Orchestra is the first system to deal with unmodifiable code for industrial-strength applications. The J-Orchestra 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 to transform indirect references into direct references—and vice versa—at runtime. The analysis algorithm's role is strictly advisory. The user can override analysis results and guide the J-Orchestra rewrite at will. The analysis algorithm's results show up in the GUI as groupings of codependent classes. The user can override these restrictions and place the classes on different machines. So, the J-Orchestra user manipulates the partitioned application at the level of individual classes or groups of classes, a control level that can yield a high degree of automation.

Developers have used J-Orchestra even to partition third-party industrial applications without any knowledge of their internals. Despite this automatic-partitioning approach's capabilities, it isn't a magic wand that removes all difficulties associated with implementing distributed systems. The main element of automatic partitioning is that the distributed application's logic and structure must remain identical to the centralized application's logic and structure, which limits the approach's applicability. For example, partitioned applications should have very clear communication and locality patterns. Because the application logic will remain the same, a large number of remote accesses will decrease performance. In addition, unmodifiable code shouldn't use objects shared among partitions. Otherwise, the application structure must change to make partitioning possible. Also, the resulting distributed application should primarily have synchronous communication patterns. If good performance or reliability requires asynchronous communication, the application structure must change.

These and other limitations make automatic partitioning particularly well suited for *resource-driven distribution*, especially when the application has distinct parts that each deal with different hardware or software resources. 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, and so forth. 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 might decide to partition a sound application to be controlled and monitored remotely from the machine where the sound is actually produced or processed.

Ubicomp systems place a high premium on flexibility and configurability. Distribution decisions might change multiple times over the system's lifetime.

Ubicomp systems are naturally distributed because they integrate many devices—sensors, displays, storage, and so forth—which is exactly the resource-driven kind of distribution at which automatic partitioning excels. 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 achieve high performance for loosely coupled applications by putting the code near the resource it accesses.[4,5]

Ubicomp systems place a high premium on flexibility and configurability. Distribution decisions might change multiple times over the system's lifetime. For example, even if a ubicomp application were originally designed to process sound on the same machine as some other part of the computation, a later version might have to change this assumption. Components should be able to be used together effortlessly and in several configurations. Ease of programming is paramount in ubicomp systems.

We 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. One example is an application where GUI actions on a PDA produce synthesized speech on a central

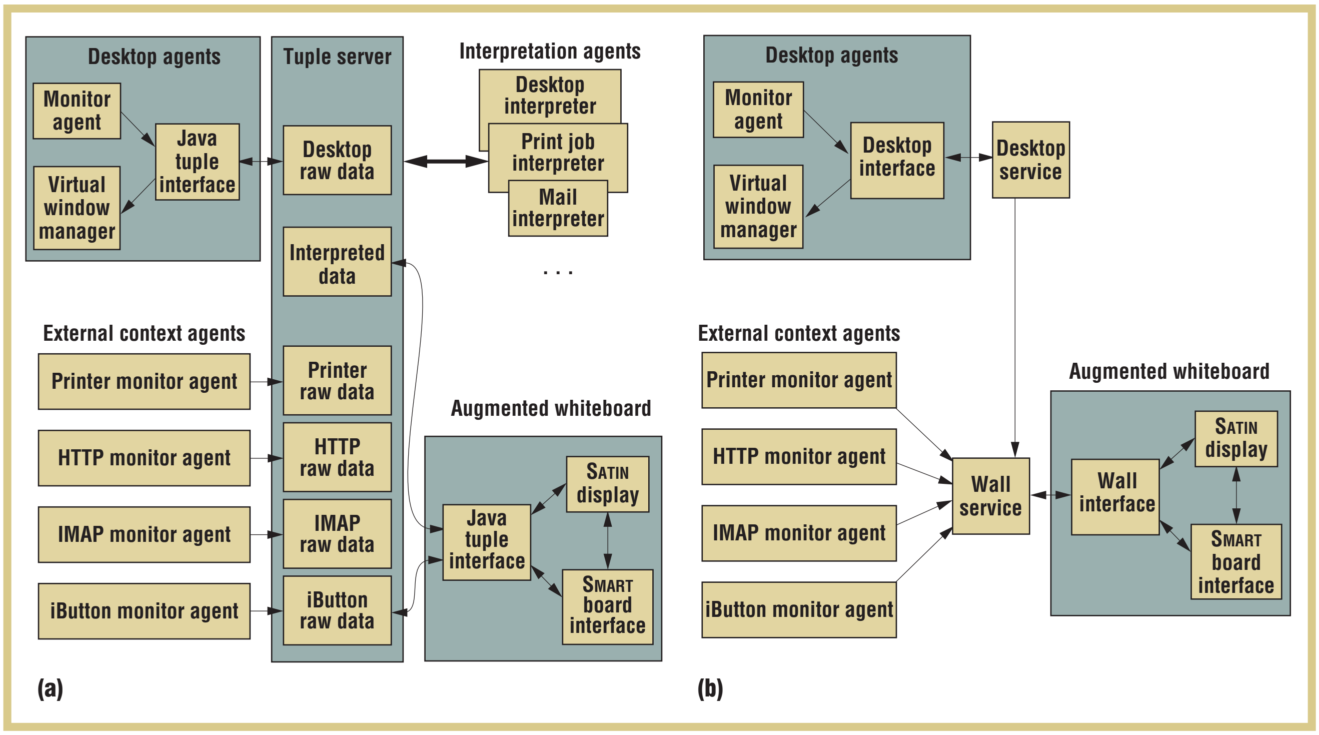


Figure 1. Kimura architecture: (a) the original system; (b) the reengineered Kimura2 system.

machine. Another example is a smart controller for our PowerPoint presentations. We wrote a small Java application that controls 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.

Kimura case study

As a larger case study of applying automatic partitioning to ubicomp systems, we used automatic partitioning in developing the latest version of the Kimura system, which is a realistic, complex ubicomp application.[6,7] Kimura is part of a research project that seeks to explore and evaluate the addition of visual peripheral displays to human-computer interfaces. Kimura uses large, projected displays as peripheral interfaces to complement an existing work area—the area surrounding a traditional desktop computer. Kimura uses these peripheral displays to help users manage multiple activities (coherent sets of tasks typically involving multiple documents, tools, or communications). Background activities are visualized on the peripheral displays as montages of images captured from activity on the desktop computer. These 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 source statements. These application classes use many system and third-party classes, including Swing and Java Advanced Imaging library classes, as well as classes that facilitate two-way communication with an electronic whiteboard.

The architecture of the original version of Kimura consists of three distinct components (see Figure 1a). 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*, which we informally call "the wall." The wall, which connects directly to several projectors and a SMART Board interactive whiteboard, maintains two-way communication with the SMART Board and provides up-to-date visualizations of the user's working contexts as montages projected onto the SMART Board.

These three components connect through TSpaces, a communication package designed to connect distinct distributed components.[8] TSpaces is based on the well-known tuplespace paradigm and incorporates database features such as transactions, persistent data, and flexible queries. It 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.

TABLE 1
Software complexity metrics.

	Kimura	Kimura2	Percent more in original
Total statements	4,436	4,084	8.6
Number of classes	98	92	6.5
Number of methods	693	682	1.6
Program difficulty metric	3,305	3,124	5.8
Development effort metric	2,611	2,235	16.8
Lack of cohesion of methods metric	2,395	2,165	10.6
Interpackage fan-out (no. of classes)	881	822	7.2

The creators of TSpaces aimed at "hitting the distributed computing sweet spot."[8] The system lets programmers ignore many hard aspects of distributed communication, such as naming, state, and load balancing. The original Kimura implementation didn't use any of these advanced TSpaces features but employed TSpaces as a convenient way to keep shared state and to broadcast global events—such as activity changes—to all system components.

Developing Kimura2

To evaluate the applicability of automatic partitioning to the ubicomp domain, we reengineered Kimura by removing the existing distribution code and redistributing it with automatic partitioning. The first step of reengineering was to separate Kimura's main application tasks from its network communication. In this way, we could create 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 could run in one process and open 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, 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 designed to act as public interfaces of the desktop and the wall. These two new modules were two singleton classes whose responsibilities included handling incoming and outgoing messages from the application's other part. Coding and integrating them with the rest of the system was straightforward. As Figure 1b illustrates, Kimura's new version no longer has a central server. Instead, the system components talk to one another directly and synchronously.

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, such as monitoring open applications. You can think of the peripheral display as a "monitoring console" for the Kimura working environment. Altogether, of the 64 automatically rewritten classes, 42 were Swing and Abstract Window Toolkit classes and six were made serializable so that they could be passed by-copy across different memory spaces to improve performance. We excluded 71 Kimura applications—four third-party and 12 Java development kit classes—from the distribution process altogether because we determined that they never participate in the distributed communication. All in all, including testing, the programmers took only a few days to partition Kimura2 with J-Orchestra.

Quantifying the benefits

Automatic partitioning turned out to be quite beneficial in the case of developing Kimura2. The main benefit is in the new software architecture's simplicity, which resulted in more understandable and maintainable code without sacrificing any of the original functionality. Kimura2's architecture will facilitate planned additions to the system much more easily because the developers can focus on the desired functionality without worrying about the distribution specifics. The new version also is easier to deploy because we don't need to maintain a running TSpaces server.

To quantify this simplicity's benefits, we used Codework's JStyle 5 (www.codework.com/JStyle/product.html) to derive software metrics. The software engineering community is still divided on software metrics' value and meaning, so the significance of our qualitative findings is somewhat subject to individual interpretation. Table 1 lists some of the more pronounced differences between Kimura and Kimura2. The new version exhibited better results in all metrics, including those not described in detail here.

Kimura originally 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 percent of the total) remained unchanged in the new version. We completely removed the TSpaces-related code (486 statements, almost 11 percent of the total) and added 134 statements. Finally, we modified 114 statements to adapt the application to the new communication paradigm.

As Table 1 shows, the new version exhibited significant differences using the Halstead program difficulty metric,[9] Shyam Chidamber and Chris Kemerer's *lack of cohesion of methods*,[10] and class fan-out (the number of classes a given class depends on). The new version is significantly less complex. Of course, we'd expect a centralized architecture to be much less complex than a distributed one. However, it's interesting to quantify the difference.

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

Benefits and shortcomings

Our experiences using automatic partitioning to develop ubicomp applications have been quite positive. The approach's overwhelming advantages include both the simplicity of coding for a single machine without the need for distributed programming and the ease of repartitioning and redeployment. Furthermore, the ability to run on unmodified runtime systems—that is, any Java VM—is invaluable when using a multitude of heterogeneous devices. Nevertheless, we've also identified several shortcomings associated with automatic partitioning for ubicomp applications. Although some of these shortcomings arguably result from limitations of the current state-of-the-art technologies, we've tried to distance ourselves and to identify the general engineering issues that are difficult to address in an automated way.

We explicitly distinguish between automatic approaches, like ours, and semiautomatic ones. As we mentioned before, 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—for example, that graphics code should run on this machine, or the main engine should run on that machine. In contrast, a semiautomatic approach could let the user annotate detailed parts of the code and data to indicate, for example, what data should be replicated, how the copies should remain consistent, and where leases should be used for fault tolerance. A semiautomatic approach could resolve many of the issues associated with automatic partitioning.

Understanding structure

Automatic partitioning is not a naive end-user technique. Often, automatic partitioning requires understanding the application's internal structure. For instance, J-Orchestra couldn't 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 we mentioned earlier, a major difficulty with automatic partitioning is dealing with unmodifiable code, such as the native Swing UI libraries in the Java runtime. Because 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 might try to access a remote object's fields 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 can't exist on two different partitions. However, we know that the Swing object partitioning in Kimura2 is safe. We know, for example, that the Swing widgets on the desktop display are distinct from the Swing widgets on the wall display. 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 doesn't offer any assistance in the problem of highly dynamic interactions between communicating entities.

Another issue with automatic partitioning in the context of ubicomp is that it doesn't offer any assistance in the problem of highly dynamic interactions between communicating entities. A common feature of ubicomp applications is to allow for resources and services to come and go dynamically as users and devices enter and leave the environment. Because automatic partitioning doesn't change the original centralized application's logic or structure, flexibility and configurability must be designed into the original application before it's partitioned. Nevertheless, although dynamic interactions can't be supported by automatic modification of an unsuspecting application, they can be supported semiautomatically. A semiautomatic system offering tool support for ubicomp development could let the user annotate the application code to express desired policies for data consistency in the context of possible failures. These annotations form a domain-specific language for specifying properties of dynamic distribution. For instance, you could annotate a certain data field to indicate that many instances of it might exist. Another annotation could specify the leases that each client holds and the data that depend on each lease. The low-level code would then be generated from the annotations instead of having to be handwritten.

Threads and failures

Automatic partitioning doesn't 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 will be active at any time. Of course, we can sometimes duplicate clients in identical configurations, but every remote method remains single-threaded. Multiple incoming remote calls would still be queued and serviced in order, which is sufficient for many common communication patterns.

Most interactive Java programs heavily use threads when accessing resources and handling interactivity. When you distribute such programs using automatic partitioning, the concurrency is maintained correctly, although remote transitions are more costly. Often, threads execute almost entirely within one partition and handle distinct resources—disk files, sound devices, graphical interactions, CPU processing, and so forth. 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 might include replication for fault tolerance; high performance through load balancing, caching, or asynchronous communication; security; and persistence. In an automatically partitioned application, it's not easy to use replication for redundancy and switch to a different server once you detect a failure. The conventional wisdom in the distributed-systems community is that mechanisms for handling distributed failure are extremely application-specific and can't be automated completely.

Again, the appropriate solution might be to follow a semiautomated approach, providing tool support for replication, load balancing, security, and so forth. In this way, the programmer would be relieved of the low-level complexity but would still be responsible for annotating parts of the code in detail and for the distribution's conceptual consistency. J-Orchestra already supports one such semiautomated approach: The user can enable complex schemes for object mobility—such as "move this object whenever it's reachable from an argument of a remote method." Nevertheless, this isn't a GUI-accessible feature. Instead, the user must write Java code that follows J-Orchestra framework conventions to enable such object mobility.

In light of these observations, we believe that automatic partitioning's benefits for ubicomp are most pronounced during prototyping. During prototyping, the benefits of avoiding distributed systems coding while being able to experiment with different partitioning schemes outweigh the communication mechanisms' potential inflexibility. In contrast, the applicability of automatic partitioning for creating mature, deployable applications can vary substantially.

If handling tough distribution issues—such as 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. We plan to do further research to determine the ideal balance between automation and power in ubicomp tools, but we believe that our study illuminates interesting aspects in the design spectrum.

ACKNOWLEDGMENTS

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

REFERENCES

1. G. Abowd, "Programming Environments … Literally: Ubicomp's Grand Challenge for Software Engineering," *ACM SIGSOFT,* ACM Press, 2002; www.cra.org/Activities/grand.challenges/abowd.pdf.

2. M. Weiser, "The Computer for the 21st Century," *Scientific American*, vol. 265, no. 3, 1991, pp. 94–104.

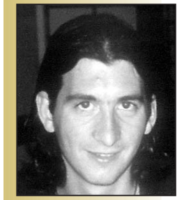
3. M. Tatsubori et al., "A Bytecode Translator for Distributed Execution of 'Legacy' Java Software," *Proc. 15th European Conf. Object-Oriented Programming* (ECOOP 2001), LNCS 2072, Springer-Verlag, 2001, pp. 236–255.

4. E. Tilevich and Y. Smaragdakis, "Automatic Application Partitioning: The J-Orchestra Approach," *Proc. 8th ECOOP Workshop Mobile Object Systems*, 2002; www.cc.gatech.edu/%7Eyannis/j-orchestra/jorch-position.pdf.

5. E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning," *Proc. 16th European Conf. Object-Oriented Programming* (ECOOP 2002), LNCS 2374, Springer-Verlag, 2002, pp. 178–204.

6. B. MacIntyre et al., "Support for Multitasking and Background Awareness Using Interactive Peripheral Displays," *Proc. ACM*

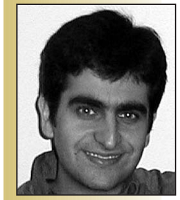
the AUTHORS



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. He received his MS in computer science from the Georgia Institute of Technology. Contact him at the Computer Science Dept., UCLA, Los Angeles, CA 90095-1596; nikitas@cs.ucla.edu.

Blair MacIntyre is an assistant professor in the Georgia Institute of Technology's College of Computing and the GVU (Graphics Visualization and Usability) Center. His research interests include understanding how to create highly interactive augmented-reality environments, especially those that use personal displays to augment a user's perception of his or her environment. He received his PhD in computer science from Columbia University. Contact him at the College of Computing, GVU Center, Georgia Tech, Atlanta, GA 30332-0280; blair@cc.gatech.edu.

Elizabeth D. Mynatt is an associate professor in the Georgia Institute of Technology's College of Computing and the GVU Center. She directs the Everyday Computing research program within the Future Computing Environments group, examining the implications of having computation continuously present in many aspects of everyday life. Her research interests include exploring how to augment everyday places and objects with computational capabilities. She received her PhD in computer science from the Georgia Institute of Technology in 1995. Contact her at the College of Computing, GVU Center, Georgia Tech, Atlanta, GA 30332-0280; mynatt@cc.gatech.edu.



Yannis Smaragdakis is an assistant professor in the Georgia Institute of Technology's College of Computing. His research interests include object-oriented programming languages, systems software, and program generators. He received his PhD in Computer Science from the University of Texas at Austin. Contact him at the College of Computing, Georgia Tech, Atlanta, GA 30332-0280; yannis@cc.gatech.edu.

Eli Tilevich is a PhD candidate in computer science at the Georgia Institute of Technology. His research interests include developing better programming language tools for distributed computing. He received his MS in computer science from New York University. Contact him at the College of Computing, Georgia Tech, Atlanta, GA 30332-0280; tilevich@cc.gatech.edu.

Stephen Voida is a PhD student in the Georgia Institute of Technology's College of Computing. His research interests include ubiquitous computing, technology in the workplace, and augmented environments. He received his MS in human-computer interaction from the Georgia Institute of Technology. Contact him at the College of Computing, GVU Center, Georgia Tech, Atlanta, GA 30332-0280; svoida@cc.gatech.edu.

Symp. User Interface Software and Technology (UIST 01), ACM Press, 2001, pp. 41–50.

7. S. Voida et al., "Integrating Virtual and Physical Context to Support Knowledge Workers," *IEEE Pervasive Computing*, vol. 1, no. 3, 2002, pp. 73–79.

8. T.J. Lehman et al., "Hitting the Distributed Computing Sweet Spot with TSpaces," *Computer Networks*, vol. 35, no. 4, 2001, pp. 457–472.

9. M.H. Halstead, *Elements of Software Science*, Elsevier, 1977.

10. S.E. Chidamber and C.F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, 1994, pp. 476–493.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.