Towards a Distributed Platform for Resource-Constrained Devices

Alan Messer, Ira Greenberg, Philippe Bernadat, Dejan Milojicic,
Deqing Chen, T.J. Giuli, Xiaohui Gu

HP Labs, Uni. of Rochester, Stanford University, Uni. of Illinois Urbana-Champaign

[messer, iragreen, bernadat, dejan]@hpl.hp.com,
lukechen@cs.rochester.edu, giuli@stanford.edu, xgu@students.uiuc.edu

Abstract

Many visions of the future predict a world with pervasive computing, where computing services and resources permeate the environment. In these visions, people will want to execute a service on any available device without worrying about whether the service has been tailored for the device. We believe that it will be difficult to create services that can execute well on the wide variety of devices that are being developed because of problems with diversity and resource constraints.

We believe that these problems can be greatly reduced by using an ad-hoc distributed platform to transparently offload portions of a service from a resource-constrained device to a nearby server. We have implemented a preliminary prototype and emulator to study this approach. Our experiments show the beneficial use of nearby resources to relieve both memory and processing constraints, when it is appropriate to do so. We believe that this approach will reduce the burden on service developers by masking many of the details of device diversity, resource limitations, and resource fluctuations.

Keywords – Distributed platform, Java, Resource constraints, Mobile computing.

Technical Area: Middleware and distributed platforms.

Contact Author: *messer@hplhp.com*

1 Introduction

Many visions of the future describe a world of pervasive computing, where computers permeate our environment and way of life [21, 34]. In these visions, computing resources in our surroundings will provide plentiful resources to support services in the locale. People will use a multitude of devices to access numerous resources and services from their environment, any where and at any time. Numerous projects are exploring this vision in industry [13, 15, 19, 28, 29], academia [5, 6, 9, 24], and government [3, 4].

Recently, there has been a growing adoption of early pervasive devices including mobile phones, personal digital assistants (PDA), and internet appliances. From these earlier devices we have identified two interesting problems in realistically supporting pervasive computing visions: diversity and resource constraints in mobile devices. One of the most striking aspects of these early devices is their diversity, which is caused by differing requirements and markets. Even in this embrionic market, these devices cover a wide range of processing powers, memory capacities, operating systems, network capacities, power supplies, and so forth. For example, enterprise PDAs such as the HP Jornada and the Compaq iPAQ are much more powerful than personal PDAs such as those made by Palm Inc. The amount of diversity increases if we also consider devices purchased over time.

Resource constraints are becoming a problem as many of these devices display increased generality. Perhaps spurred by competitive forces and limited physical space, many of these devices are becoming more generic with each revision. For example, observe the push to add Java virtual machines into mobile phones and PDAs. Unfortunately, for the foreseeable future, size, weight, power, and heat factors limit the amount of resources that can be placed into mobile devices when compared with their stationary, powered counterparts.

We believe that diversity problems will make it difficult for software vendors to support all of the different platforms. At the same time, resource constraints will make it difficult to provide the same full-featured service to all devices. These issues raise the question of whether these devices will become truly pervasive if users must continually upgrade them or must have a detailed understanding of their limitations in order to be able to use services on them.

We propose an approach that allows diverse, resource-constrained devices to execute the same full version of an application. The idea is to enhance a mobile device's run time platform so that it can dynamically and transparently establish a distributed platform with the other computing resources in its environment. If a device becomes resource constrained at run time and believes it can beneficially use nearby resources, it automatically and transparently offloads part of the service to them. By monitoring execution needs and resource availability, the platform can dynamically decide how much of the surrounding resources to use. As a result, the proposed distributed platform increases the level of abstrac-

tion at which services view the actual resources of a device.

In this paper, we first introduce our vision of the desirable properties for a transparent, distributed platform for diverse, resource-constrained devices. In Section 3, we describe AIDE, our approach to realizing an instance of this conceptual platform for experimentation. We then discuss the implementation of the AIDE modules in Section 4. In Section 5, we describe our preliminary experiments we have undertaken with the platform on some sample applications. Section 6 covers the lessons that we have learnt from our initial experiments. Section 7 presents a detailed comparison of our work with similar related research. Finally, Section 8 describes the future work that we are considering, and we conclude in Section 9.

2 Distributed Platform

In the future, many environments are expected to contain a multitude of computing devices. These devices will come in many forms, including desktops, embedded servers and computers (such as a meeting room server), personal computing devices, and so forth. Each device may contain many resources, such as processing or memory, resulting in an enviroment full of computing resources.

In this environment, we refer to a device that could provide the use of some or all of its resources to another device as *a surrogate*. Devices which may choose to use resources from surrogates using wired or wireless networks we refer to as *clients*. A device can perform the role of a surrogate with respect to a client even though it may be used independently for other purposes. Also in general, we view surrogates as having more computing power and memory than clients, but this isn't necessary.

We believe that client diversity and resource constraints can be allievated by transparently using surrogate resources in this environment. We propose that a *distributed platform* can be used to achieve this transparent resource usage. We refer to the transparent use of surrogate resources by a client as *offloading, i*n the sense that computation and data is offloaded from the client to the surrogate. Not only does this allow network and memory resources to be used, this approach also provides a the capability for using processing resources from the surrogate. If the necessary resources for a client are not available at the closest surrogate, multiple surrogates could be used by the client, or surrogates could offload to other surrogates to provide access to suitable resources for the client.

Distributed platforms cover a wide range of types of platform support and technologies to achieve a full platform for applications. In this paper we use distributed platform to refer to a system-level layer that provides a shared execution environment across two or more machines. This enables us to focus on the aspects of the platform that we believe directly relate to the problems of client diversity and resource constraints. Aspects such as security, fault tolerance, discovery, and so forth are not covered in this paper.

We believe that a distributed platform to support transparent offloading for diverse, resource-constrained clients requires the following features.

Transparent, distributed execution – It should be possible to execute an application on multiple machines without the application code being aware that multiple machines are being used. In addition the platform should give the application the appearance of executing only on the client device. These features allow the platform to hide the complexities of remote execution and to allow applications to be written more independently of the underlying hardware.

Application partitioning – It should be possible to dynamically divide the application at run time into two (or more) partitions that can be placed on different devices. Partitioning may take place at any granularity suitable to the platform and/or application. The resulting partitioning should produce a partition for the client that it can execute under its constraints.

Adaptive offloading – To be effective, it should be possible for the partitioning algorithm to consider the available resources and the application's execution patterns. Based on either resource variation triggers or periodic re-evaluation, the platform should be able to adapt to load and execution changes to maintain a good partitioning/offloading decision.

Beneficial offloading – The platform should only offload a portion of an application if doing so would benefit the user. We define offloading as being beneficial if it improves the performance of the application (e. g., its speed or battery life), or if it allows the application to execute when it was not able to do so before (e. g., it overcomes a memory limitation). It should also be possible for the user to specify what is beneficial. For example, a user may choose to extend battery life at the cost of slower execution in order to allow the device to continue functioning during a long airplane flight.

Ad-hoc platform creation – It should be possible to create and tear down the distributed platform between a client and a surrogate at run time. Where clients are able to determine which surrogate(s) are the most appropriate to be used based on factors such as latency of access and resource availablility.

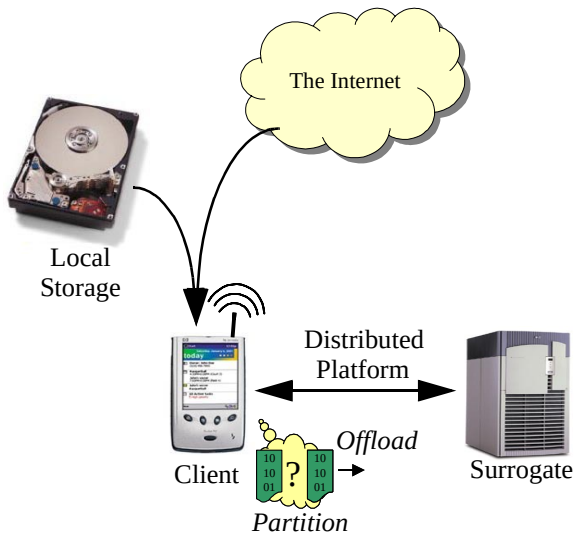

Figure 1: *A high-level view of the interaction and operation of the proposed distributed platform for diverse, resource-constrained devices.*

We envisage this platform being used as seen in Figure 1. A user locates, and obtains an application either over the network or from local storage and starts to execute the application on the client device. While the application executes, the client platform monitors the application's execution and the state of system resources such as memory and network bandwidth.

When a trigger event occurs, such as resources running low or periodical re-evaluation, the client platform analyzes the information it has collected and decides whether offloading should occur. If it is beneficial to offload, the client platform will select the components to offload and offload them to one or more surrogates running the distributed platform. The application will then continue to execute and monitoring will resume. If the application tries to access what is now remote data or to invoke what is now a remote method, the client side of the distributed platform transparently communicates to pass execution or access to the surrogate. Similarly, execution on the surrogate side will transparently refer back to the client for data accesses and method invocations on the client.

We believe that a distributed platform with these features would provide the following benefits.

- Applications can be written more generically for a more diverse set of devices because resource constraints would need less consideration.
- The division of responsibility between a client and a surrogate can be dynamically altered based on the available resources in the network, on the surrogate, and on the client.
- Component-oriented, monolithic, and client/server applications can all be supported, as long as components can be offloaded independently and information on component interactions can be gathered.
- Surrounding surrogate resources are used only when beneficial for the user and application.

Several of these features are available in particular platforms or in other research projects, but no system supports them all nor are they used in this context. Our research focuses on investigating the possibilities and benefits of a platform combining all of these features, particularly in the area of mobile computing devices.

In this paper, we make certain assumptions to act as a basis for our investigation and to limit its scope. Those assumptions are as follows.

- The reliability of communication connectivity between clients and surrogates is good enough that errors would not be a problem to users of our proposed system. Otherwise that reliability can be orthogonally added to the platform using existing techniques such as replication/redundancy.
- Computing resources in the surrounding environment will oversupply that required by the environment at any one time.
- Physical devices limitations such as screen size and user interface constraints will be handled by orthogonal techniques such as transcoding.

3 Approach

To investigate this vision, we built, AIDE, an example realization of our distributed platform for diverse, resource-constrained devices. AIDE consists of three platform modules to monitor application execution, partition the workload according to a policy, and support migrating and executing offloaded components transparently. We implemented both a prototype and a trace-driven emulation that share the AIDE modules. The prototype allows validation and demonstration for a limited set of applications and features. The emulator allows full-featured repeatable experimentation.

While transparent, distributed execution has been achieved in several settings [14], we decided to base our implementation on Java virtual machines (JVM). Using Java resolves many of the standard heterogeneity and library problems, and allows us to use many applications that have already been written in Java.

In this section, we describe how we realized our transparent, distributed platform. First, we describe our approach to deriving the types of Java components we selected for investigation. We then describe the steps we took to adapt two JVMs into a virtual transparent platform. Next, we present our approach for partitioning

monolithic Java applications. We end with a description of how we tackled extracting execution and resource information from the JVM at run time.

3.1 Componentization

When viewed at a high level, a Java application is written as a single monolithic unit or as a group of interacting components built from a component-oriented toolkit such as Sun's JavaBeans. Some research has been done into statically partitioning an application that was compased of high-level components [12]. However, for our approach we require dynamic partitioning at run time. In addition, we would also like to handle monolithic applications which account for most applications, in the Java language.

Looking more closely, all Java applications can be considered component oriented because they are all composed of objects and classes. Thus, there are three component granularities: objects, classes, and higher-level components such as JavaBeans. From our prespective each level influences the overhead of execution monitoring, the accuracy and flexibility of offloading, and the type of support required for remote execution.

For our current investigation, we selected *classes* as the application components because they are a middle ground among these options. As a result we hope to understand the influence of this choice on component granularity and the performance/storage overhead of monitoring. We also ruled out higher-level components for initial investigation because comparatively few applications are constructed using them. In the future, we plan to consider the ability to include their additional semantic information in the partitioning decision.

3.2 Transparent, Distributed Execution

Java's existing support for remote execution, RMI, requires applications to be written with explicit interfaces. This provides excellent support for client/server application programming, but client diversity would require new client/server applications to be written for each combination. This limitation prevents us from using RMI to support the necessary fully transparent execution of objects across multiple machines.

To overcome this limitation, we modified the JVMs so that objects can be transparently migrated between the client and surrogate. In a JVM, an object is uniquely identified by an object reference. To support remote execution, we modified the JVM to flag object references to remote objects and then intercept accesses to remote objects. Using these hooks, our modules convert remote accesses into transparent RPCs between two JVMs. Either JVM that receives a request uses a pool of threads to perform RPCs on behalf of the other JVM. Using this approach, threads are not migrated. Instead, invocations and data accesses follow the placement of objects.

This support brings out several issues that must be addressed to support the goal of providing our single, transparent distributed platform between the JVMs.

- **Native methods.** Java applications must ultimately call native methods to perform certain functions. These methods cannot be migrated because they are implemented with native code. In addition, native methods may have different effects on different platforms if they use local state, which cannot be accessed by the JVM. To solve this problem, native invocations are directed back to the client JVM. This gives applications the appearance of executing on the client even though part of their execution is on a surrogate. In some cases, as shown in Section 5.2, some of these constraints for native methods can be relaxed while still providing a transparent platform.
- **Static functions and data.** Static functions and data are shared between object instances. Like native methods, some static data in the JVM may also contain data that is specific to the implementation or host where it is located. For example, System.properties contains <key, value> pairs specifying information such as the name of the host operating system. Access to static functions may execute on either JVM as long as their implementations are equivalent. However, to ensure the consistency of static data, we chose to direct access back to the client JVM.
- **Object references.** Each JVM has a private object reference namespace and does not understand an object reference from another JVM. To overcome their namespace limitations, we modified the JVMs to map each others references into their own namespace. To acheive this each JVM keeps stub local references for remote objects as a placeholder. Then when one JVM invokes a method or accesses an object on the other JVM, it sends an operation referring to that object using its local object reference. The receiving JVM then maps these references to its own real local references for the objects. As a result, the two JVMs maintain object reference mappings are kept when objects and object references move between the VMs.

3.3 Partitioning

Our approach to determining whether a beneficial offloading exists is to reduce the problem to finding an appropriate partitioning of the graph of the application's execution history. The rational behind this idea is that if two components interact frequently (e. g., because of

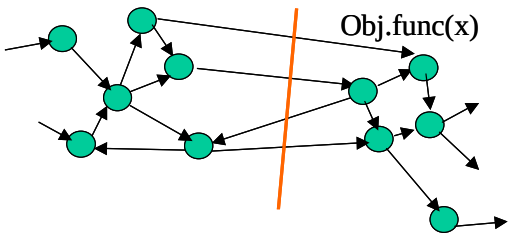

Figure 2: *Tightly coupled components should be placed together on either the client or surrogate device. Here, nodes represent components and the length of the arrow represents the component interaction frequency with a shorter arrow representing greater frequency.*

many method invocations), then the graph will reflect this situation with a high-weight edge. Any partitioning policy should have a high probability of placing frequently interacting components together on one machine or the other, because splitting them across the network could severely affect performance (see Figure 2).

Assuming that the execution history reflects future execution, we use the execution history to predict the application's future behavior. We believe that combining this information with resource availability information will allow a partitioning policy to effectively select a partitioning that balances the application's resource requirements, the system's resource availabililty, and the user's preferences.

Finding the best partitioning of an execution graph is an NP-Complete problem. Several approaches exist for splitting a graph to obtain a good solution based on the weights of the edges, but they do not necessarily result in a good partitioning. For example, one of the most popular heuristics is MINCUT, which bisects a graph along the cut with the fewest interactions or the smallest interaction weight. However, it may simply remove a single component, which may not free enough memory to satisfy the partitioning policy.

To solve this problem, we developed a heuristic that produces a group of approximate minimum cut partitionings. We then evaluate all of these partitionings and select the one that best satisfies the partitioning policy. The heuristic is derived from the MINCUT heuristic developed by Stoer and Wagner [27].

In their approach, they place one node in one partition and the rest of the nodes in another partition. They then move into the first partition the node in the second partition with the greatest connectivity to the first partition. This process is repeated until the first partition contains all but one of the nodes. The minimum cut is the partitioning with the lowest interpartition weight.

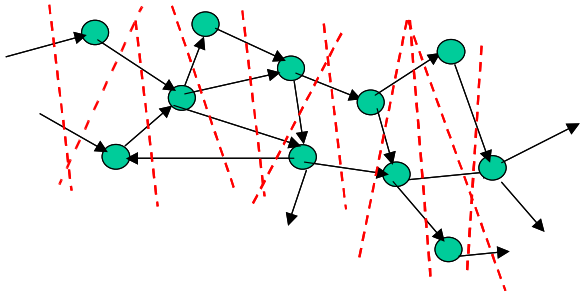

Figure 3: *Multiple partitionings of an execution graph. The partitionings are denoted by the dashed lines.*

Our heuristic begins by placing all of the nodes that represent classes that cannot be offloaded, such as classes that contain native methods, into the first partition. This is the partition that will remain on the client device. We then follow the MINCUT heuristic, moving one node at a time.

All of these intermediate partitionings are evaluated according to the partitioning policy (see Figure 3). The partitioning that is selected may not have the minimum interaction cost, but it will satisfy the overall policy best. The number of partitionings that will be evaluated is smaller than the number of components.

A particular intermediate partitioning is evaluated by applying a cost function that represents part of the partitioning policy. In this paper, we use a cost function that returns the historical amount of information transferred between the two partitions. The partitioning policy then selects among the candidate partitions to determine whether any part of the application can be beneficially offloaded. If so, it determines whether components can be offloaded without severely affecting other metrics and constraints. For example, in the prototype, we add the restriction that at least a certain amount of memory must be freed by any partitioning. If no such partitioning exists, then offloading does not occur. Conceptually, this policy offloads a sufficient amount of information while placing the smallest demand on network bandwidth.

3.4 Execution and Resource Monitoring

Information needs to be collected about an application's execution while it executes in order to be able to partition an application. This is accomplished by augmenting the JVM's code for method invocations, data field accesses, object creation, and object deletion. The information is obtained at the object level and aggregated to the class level. The prototype collects the amount of memory occupied by the objects of a class, the number of interactions between two classes, and the amount of information exchanged between two classes as repre-

sented by the parameters and return values used in inter-class interactions.

The execution information is represented as a fully connected weighted graph representing the execution graph, to reflect the application's execution history. Each node represents a class and is annotated with the amount of memory occupied by the objects of that class. Each edge represents the interactions between two classes and is annotated with the number of interactions between objects of the classes and the total amount of information transferred between objects of the classes.

For a partitioning to remain effective as the environment changes, the partitioning process also needs to adapt to be suitable to the underlying resource constraints. For the purposes of this paper we focus on memory as the primary constraint, although in Section 5.2, we also examine processor load. To monitor the constraints on memory the prototype tracks the amount of free space in the Java heap with information obtained from the JVM's garbage collector. The AIDE modules can then use this information to determine whether objects will fit into the VM, and to detect when memory is becoming constrained.

4 Implementation

To study the distributed platform, we implemented a prototype and emulation of the approach described in Section 3. The prototype executes under Linux on HP PCs and under Windows CE on HP Jornadas. It supports the distributed execution of monolithic Java programs, and performs a single offloading from a client device to a single surrogate server.

The prototype was built by modifying HP's Chai VM, version 5.1. The general architecture of the prototype is displayed in Figure 4. Three modules containing approximately 3,000 lines of C++ code were added. The monitoring module records execution monitoring information as a weighted execution graph. The partitioning module applies the modified MINCUT heuristic to the execution graph according to a built-in partitioning policy and offloads selected objects to the surrogate. The remote invocation module implements RPCs between the VMs and manages external object references. Several hundred additional lines of code, which are scattered throughout the Chai VM, extract execution and resource monitoring information and redirect remote invocations and data accesses. The modifications were designed and accomplished in reasonable time taking approximately five person-months.

Currently, graph partitioning is performed solely on the client JVM. Distributed monitoring of the application's

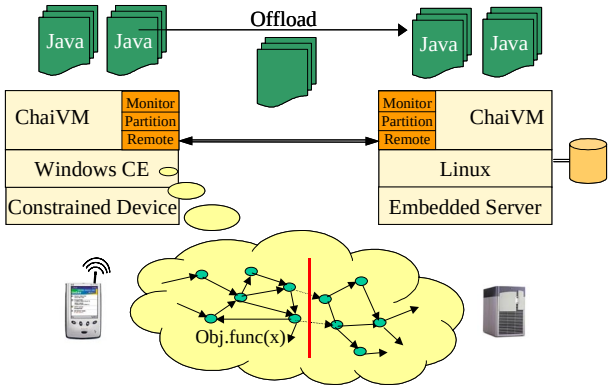

Figure 4: *The overall architecture of the distributed platform based on the Chai VM.*

execution and distributed partitioning of the execution graph would be more suitable in a real-world solution.

To simplify the platform, we assumed that both VMs have access to the application's Java bytecodes. This allows both VMs to have common knowledge about the application. In a real-world implementation, the surrogate would have to acquire the necessary bytecodes from the client or another device, or have them installed.

The emulator executes the same three modules that are used in the prototype. The Chai VM is replaced with a wrapper that is used to play back execution and resource traces into the modules. The traces for an application were extracted from the prototype while running the application to completion on a single PC. The emulation is able to repeatedly repartition an application, exploiting its ability to easily share the execution graph between the emulated VMs.

Distributed execution of an application trace by the emulator is assumed to be equivalent to serial execution of the trace. That is, after partitioning, execution moves between the two emulated VMs synchronously, and the two VMs do not execute application code simultaneously.

The emulator simulates remote communication by stretching simulated execution time to account for remote invocations and data accesses. In the emulator, remote communication is based on an 11Mbps WaveLAN link with a 2.4ms round-trip time for a null message. A communication trace was acquired by monitoring communication on a real WaveLAN link.

Unless otherwise stated, the prototype and emulator require static data to be accessed on the client VM and native methods to be executed on the client VM. Other data can be offloaded, and static methods written in Java (i. e., those associated only with a class) can execute locally on either VM. New objects are always created on

the VM that performs the creation operation. Finally, a simple distributed garbage collection scheme is supported to account for objects that are referenced from the other VM.

5 Experiments

We performed several experiments to gain an understanding of the operation and performance of the proposed platform. The first group of experiments focused on offloading processing and memory to alleviate memory constraints, while the second group of experiments addressed computing constraints. In the future, we plan to examine both types of constraints together.

Table 1 lists the applications that were studied. They represent a variety of application characteristics and resource demands.

Name	Description	Resource Demands
JavaNote	*Simple text editor*	*Content-based memory intensive*
Dia	*Image manipulation program*	*Content-based memory intensive*
Biomer	*Molecular editing application*	*Memory/CPU intensive*
Voxel	*Fractal landscape generator*	*CPU intensive, interactive*
Tracer	*Interactive Java Raytracer*	*CPU intensive, low interaction*

Table 1: Java applications used for experiments.

In Section 5.1, we examine the use of offloading to relieve memory constraints. First, our prototype is used to offload data and computation when the JavaNote application runs out of memory. We then use our emulator to study the cost of remote execution overhead for several applications. This is followed by a discusion of three important issues related to this overhead: the dynamic selection of partitioning policies, component granularity, and native method execution. Finally, we investigate the performance and storage overhead of execution monitoriing in the prototype. In Section 5.2, we employ the emulator to study the use of offloading to relieve computing constraints.

5.1 Offloading under Memory Constraints

Initially, we investigated the use of offloading when memory becomes constrained. This problem is interesting because memory is a finite resource (unlike processing) and can easily be exhausted on small devices.

Avoiding Memory Constraints

The scenario for the first experiment used JavaNote to load and edit a 600KB text file. When executed on an unmodified version of the Chai VM with a 6MB Java heap, the application runs out of memory and fails. When executed on our prototype, the lack of available memory is detected, data and computation is offloaded to the surrogate, and execution continues.

In the prototype, memory consumption is represented by the amount of free memory in the Java heap after each garbage collection cycle. Chai (and hence the prototype) uses an incremental mark-and-sweep algorithm that is triggered by space limitations, the number of objects created since the last collection, and the amount of memory occupied by objects created since the last collection. This causes the garbage collector to perform at least a partial sweep often, which produces frequent memory usage updates. In the policy that was used, partitioning is triggered when three successive garbage collection cycles indicate that additional memory cannot be freed or that less than 5% of memory is available.

While executing JavaNote, the prototype monitors the application's execution and constructs an execution graph. Figure 5 illustrates the complexity of an actual application execution graph.[1] Figure 5a shows the execution graph at the moment when the application runs out of memory.

The modified MINCUT heuristic is applied to the graph in Figure 5a to look for a suitable partitioning. The partitioning policy that was used required any acceptable partitioning to free at least 20% of the Java heap. The goal of the policy is to free a sufficient amount of heap while minimizing the network bandwidth caused by remote interactions.

Figure 5b illustrates the partitioning that is selected. Two subgraphs are formed based on offloading both data and computation. The objects in the classes represented by the left subgraph remain on the client, and the objects in the classes represented by the right subgraph are offloaded to the surrogate. The partitioning heuristic took approximately 0.1 seconds to compute on a 600MHz Pentium.

Interestingly, the partitioning policy offloaded more of the Java heap than the required 20% minimum because it detected that the bandwidth of the interactions was

1. In Figure 5, the length of an edge does not represent the strength of the coupling between classes. The lengths are assigned by the graph rendering algorithm as it flattens a multidimensional graph into two dimensions.

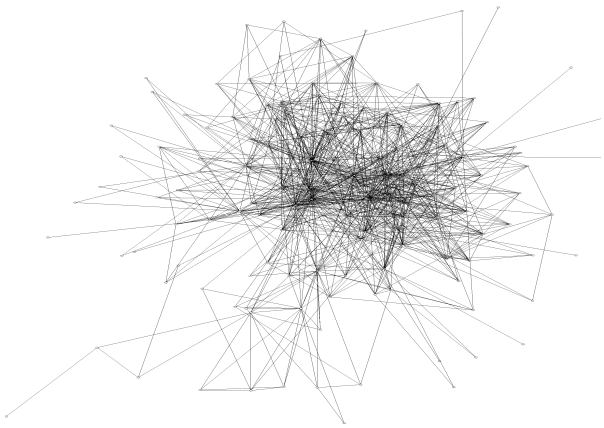

Figure 5a.

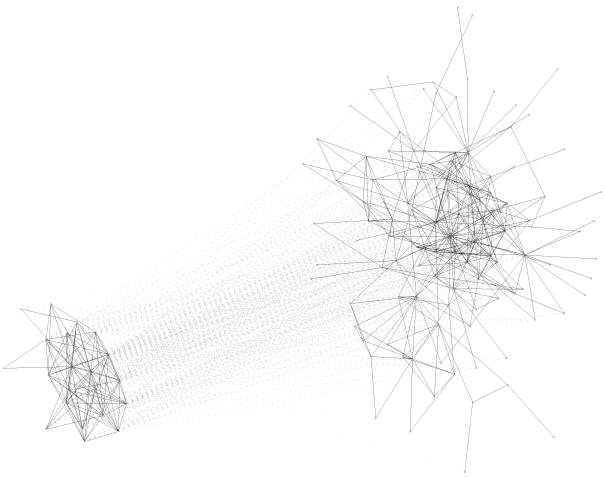

Figure 5b.

Figure 5: *The execution graph for the JavaNote application, when the Java heap is exhausted (Figure 5a) and immediately after the application is partitioned (Figure 5b). A node represents a class, and an edge represents interactions (method invocations, data accesses). In Figure 5b, dotted edges represent remote interactions and have been stretched to clearly separate the two subgraphs.*

minimized when approximately 90% of the heap was offloaded. Application data accounted for most of the objects that were offloaded. Based on the historical execution graph, the bandwidth caused by interactions between the two JVMs is predicted to be only 100KB per second.

Offloaded Performance

Using the emulator, we then compared the overhead of remote execution caused by offloading for several applications. The remote execution overhead was calculated as the offloading time plus the communication time for

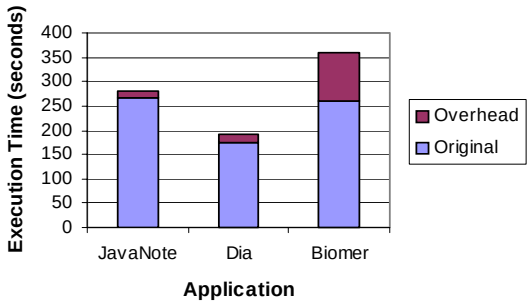

Figure 6: *Remote execution overhead caused by the initial partitioning policies, which have an offloading threshold of 300KB (5%) and free at least 20% of memory.*

remote interactions. For this experiment, the same processor speed was used for both the client and the surrogate, and communication was based on a WaveLAN link. We employed the same triggering and partitioning policies that were described in the previous section. In Figure 6, which shows the results of the experiment, the overhead for JavaNote and Dia seems reasonable at 4.8% and 8.5% respectively, but the overhead for Biomer is much worse at 27.5%.

We have identified three causes for the high overhead, especially for Biomer. First, the policy used may not be appropriate for the application. Attempting to free less memory or to free memory at a different trigger point might have produced a lower overhead. Second, because a class is the unit of placement, all of the objects in a class will be moved to the same site even though some of the objects are highly referenced only on the client while others are highly referenced only on the surrogate. This can be a problem for common generic types, such as String or Integer. Third, execution access patterns may cause objects on the surrogate to frequently reference objects that must remain on the client, such as native methods for screen updates.

Effect of Policy on Performance

To evaluate the effect of the triggering and partitioning policies on the remote execution overhead, we repartitioned the same execution traces under multiple policies. The partition triggering threshold was varied from when 2% to 50% of memory remained free, the tolerance to low-memory signals was varied from one to three events, and the minimum amount of memory to free was varied from 10% to 80%. Figure 7 illustrates the minimum overhead observed for the best policy for each application.

From our experiments, the remote execution overhead was reduced for Biomer and Dia by between 30% and

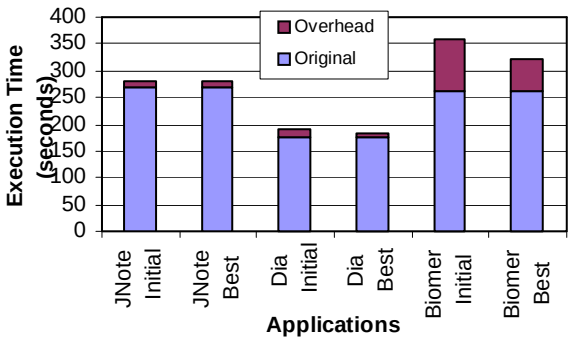

Figure 7: *Comparison of the effect of different partitioning policies on the remote execution overhead.*

43%, but remained the same for JavaNote. Both Biomer and Dia happened to perform best with a triggering threshold of 50%, one positive report, and a minimum of between 10% and 80% of memory freed. However, JavaNote performed best with a triggering threshold of 5%, three positive reports and a minimum of 20% memory freed. This indicates that the system needs to be able to select among policies and policy parameters to achieve the best performance, based on analysis, knowledge about the type of the application, and so forth.

Effect of Granularity on Partitioning

One consequence of using a class as the unit of placement is that the total amount of memory associated with the objects in a single class can represent a large percentage of the memory available for offloading. Because all of the class's objects must be placed on one site or the other, there will be fewer options available for the partitioning policy. For example, in JavaNote, the primitive character arrays (which contain the document being edited, menu items, etc.) account for a large percentage of the available memory. It might be desirable to be able to offload just a selected group of objects from a class.

A related situation occurs when one or a few objects represent a large percentage of the memory available for offloading. In this case, however, objects are the smallest natural component of a Java program.

A class can be composed of groups of unrelated objects that are used by the application in different ways. Another consequence of using a class as the unit of placement is that all of its objects are forced to reside on one site, even though some of the objects are highly referenced only on one site or the other. Forcing all of the objects to reside on one site can cause a high remote execution overhead. This issue is explored further in Section 5.2

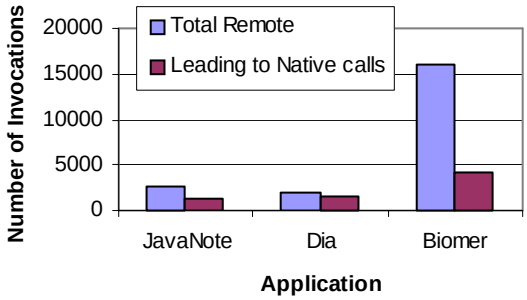

Figure 8: *Comparison of remote native method invocations to total remote invocations.*

Effect of Native Methods on Performance

Next, we studied how much the applications depend on code that must execute on the client device. We measured the number of references from code executing on the surrogate to native methods, which have been required to execute on the client. Figure 8 shows that for some applications native methods can account for quite a large percentage of remote accesses (JavaNote, Dia), while for others it is a relatively small percentage.

Interestingly, however, many of these native methods do not actually need to execute only on the client because they are stateless and/or idempotent operations such as string copy or mathematical functions. With some work, it may also be possible to allow other native methods such as file operations to move to the surrogate. However, some native methods will always have to be executed on the client, such as graphical framebuffer access. Given this, we believe that the number of remote calls caused by native methods can be significantly reduced by annotating the methods in the standard Java library according to their operation type.

Monitoring Overhead

Finally, we measured the performance and storage overhead of operating the platform without partitioning. JavaNote was executed on a PC with a 600MHz Pentium using our prototype with monitoring on and with monitoring off. In both cases, a 600KB file was opened, and then a small amount of editing and scrolling was performed on the file. By using an 8MB Java heap, the application was able to execute without running out of memory.

When executed without monitoring, the scenario executed in an average of 31.59 seconds. When executed with monitoring, it executed in an average of 35.04 seconds, resulting in a monitoring performance overhead of

11%. We believe that this value can be reduced by adding optimizations to our initial prototype.

Table 2 displays the average execution metrics for several runs of the scenario described above. The class and object events are creation and deletion, and the interaction events are invocation and access. The 1.2 million interaction events represent interactions between two classes and are almost evenly divided between invocations and accesses.

	average	maximum	total events
classes	134	138	138
objects	1,230	2,810	6,808
interactions	1,126	1,190	1,186,532

Table 2: Execution metrics for JavaNote.

Information is recorded only for interactions between two different classes. For each such link in the execution graph, we keep a running total of the number of interaction events and the number of bytes passed between the two classes. Note that the average number of links (interactions) is much smaller than the number of interaction events. Based on these values, the execution graph occupies a relatively small amount of storage. Any additional information needed for a partitioning decision is obtained from the Java heap and other JVM data structures.

5.2 Offloading under Processing Constraints

So far, we have considered offloading when memory becomes constrained. However, just relieving memory constraints can be achieved by other methods such as virtual-memory paging or swapping. The real value of the intelligent, distributed platform approach proposed here is the ability to offload processing, too. In this section, we cover some results related to offloading with processing constraints as the goal.

To evaluate the effect of partitioning policies on performance, we needed to determine the amount of execution associated with each class. This time was obtained by extending execution monitoring to calculate the approximate execution time in each method invocation. The Unix `gettimeofday()` system call was used to approximate the time spent in an invocation. The time spent in a class is the total of the time spent in any of that class's methods, and is assigned to the node that represents the class in the execution graph. When combined with the interaction frequencies already in the execution graph, there is enough information to approximate the execution time of pieces of the application.

The mapping of execution time to the execution graph is illustrated in Figure 9. The execution time assigned to a

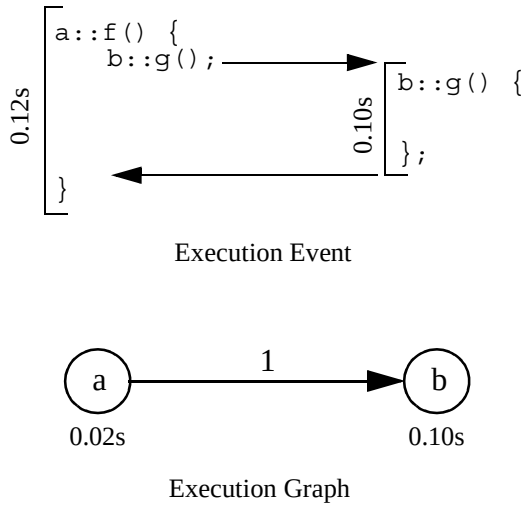

Figure 9: *The mapping of method execution times to an execution graph. Note that node 'a' accounts only for time spent in method f( ) of class 'a' and does not include time spent in method g( ) of class 'b'.*

class is the time spent in one of that class's methods minus the time spent in nested calls to methods of other classes. Here, it takes 0.12 seconds to execute method f( ) in class a, but 0.10 of those seconds are spend in a nested call to method g( ) in class b. As a result, only 0.02 seconds are assigned to class a. The '1' over the edge in the execution graph indicates that there has been one interaction between class a and class b.

For these experiments, the emulator was configured to have the surrogate execute 3.5 times faster than the client. This figure was obtained by comparing the execution of our applications on a Jornada 547 and a PC. Communication costs were based on a WaveLAN link.

The results for the applications Voxel, Tracer, and Biomer are shown in Figure 10. The bar marked "Original" represents the application executing only on the client. The bar marked "Initial" represents the initial result for executing the application with offloading. In all three cases, offloading caused performance to increase in spite of the faster CPU on the surrogate.

As might be expected, analysis shows that offloading to improve processing experiences the same problems as offloading to improve memory usage. In particular, using a class as the unit of placement and executing native methods only on the client can lead to high communication overhead.

To better understand the effects of these issues, we added two simple enhancements to the emulator. Because many of the native method calls were to math functions, which are stateless, we modified the emulator

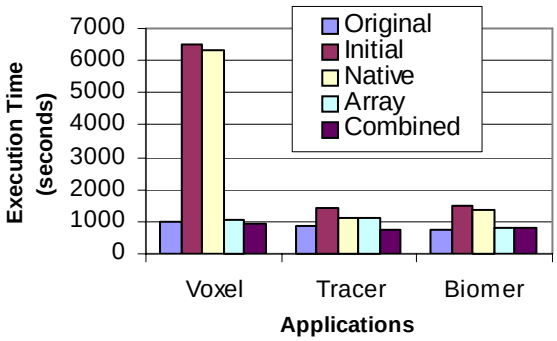

Figure 10: *Effect of offloading on application performance. Improvements were achieved by using enhancements for stateless native methods alone, primitive integer arrays alone, and both features combined.*

to treat calls to math functions as local. To address the component granularity issue, we allowed the emulator to consider primitive integer arrays at an object granularity instead of at a class granularity. These arrays were fairly common in the applications and were often used for a variety of unrelated purposes.

The results that were obtained after employing these enhancements are also shown in Figure 10. The bar labeled "Native" represents executing the application with offloading and the native method enhancement, the bar labeled "Array" represents executing the application with offloading and the integer array enhancement, and the bar labeled "Combined" represents executing the application with offloading and both enhancements.

These results demonstrate that with these two simple enhancements it is possible to use offloading to improve the execution speed of some applications (Voxel and Tracer). For Biomer, the system determined that there was no beneficial partitioning, and correctly decided not to offload any objects. Its best partitioning was predicted to take 790 seconds while the unpartitioned application took 750 seconds. However, by partitioning the application manually, we were able to find a beneficial partitioning of 711 seconds. More work is needed to understand and improve offloading in this type of application.

While the speedup achieved is modest and the applications have fairly good computational locality, we believe that these results can be improved with further investigation. This may be especially true for frequently mentioned pervasive computing tasks such as voice and image recognition.

6 Lessons Learned

We learned the following lessons from our initial investigations.

- **Feasible to offload without prior knowledge.**

  Based on our initial experiments, we believe that it is feasible to adaptively offload data and computation from a resource-constrained device to a surrogate device without prior knowledge about the application. Our experiments were performed without modifying the application programs, without hints from the application developer or the user, and with applications that weren't designed for distributed execution. All of the execution and resource information used by the partitioning policy was gathered and analyzed at run time, and this was accomplished with reasonable performance and storage overhead. In addition, objects were selected and offloaded at run time with reasonable overhead.

- **Component granularity is important.**

  In an object-oriented programming language, classes and objects are the two obvious units to use as the components for monitoring and offloading. We believe that it will be necessary to find a middle ground so that the overhead of execution monitoring and analysis can be kept at a reasonable level while allowing placement to better exploit the structure of the application. Perhaps classes should be used as the unit of monitoring and objects should be used as the unit of placement. Selective use of objects as the unit of placement to improve performance was shown with primitive arrays in Figure 10.

- **Memory constraints can be relieved.**

  We believe that our approach will increase the ability of clients to execute memory-constrained applications. For example, in the experiment described in Section 5.1, JavaNote would have failed with an out-of-memory error if executed on the client device alone. By using memory offloading, our prototype was able to execute the application using the client's Java heap and the surrogate's Java heap.

- **Processing constraints can be relieved.**

  We believe that our approach will enable some applications to be executed more quickly while using a particular client. As shown in Figure 10, our emulator was able to produce performance savings of up to 15%. However, to achieve these savings, it was necessary to use two enhancements — offloading primitive arrays on an object basis and executing math functions on the device where they were invoked.

- **Native methods can degrade performance.**

  We believe that the simple policy of executing native methods on the client device can lead to performance penalties. Further, we believe that significant gains can be achieved by identifying native methods that are

stateless and executing them on the device on which they are invoked. These observations were demonstrated by the experiments in Section 5.2. In order for this approach to work, the native methods must have the same interface and behavior on both devices.

- **Must select policies dynamically.**

  To achive satisfactory performance, we believe that it will be necessary for the system to dynamically select among partitioning policies and adjust policy parameters. This was illustrated in Figure 7 where some triggering and partitioning policies led to significantly lower performance overheads than others. Also, the best policies were very different from the policy that was initially selected.

7 Related Work

Offloading functionality from resource-constrained devices has been used for many years. Client/server applications and frameworks such as the X Windows System [25], CORBA [22], and the World Wide Web [1] provide the ability to write an application in a pre-partitioned fashion. Distributed agents, load-balancing and process-migration systems have investigated the dynamic placement problem for programs written as distributed components [18]. However, such distributed systems either migrate whole program entities, or require applications to be written for distributed execution. Also, they usually assume that the system has a fairly stable size and load configuration.

Scientific computing has addressed the partitioning of computation and data across parallel computers using interaction graphs and partitioning algorithms [10, 27]. We used this research as the basis for our dynamic partitioning algorithm in the context of partitioning Java applications. Systems such as MultiJav, JavaParty have investigated distributed Java platforms in the context of distributed shared memory and cluster computing [2, 23]. Our distributed platform leverages this work to support transparency for our partitioning and offloading support.

Coign tackled the problem of fixed client/server partitioning in their work on the profiling-based partitioning applications [12]. M-Mail saw the need for the adaptive placement of application modules in client/server applications based on usage patterns [17]. Work at AT&T and IBM's research labs has investigated the issues of graph reduction and optimization by alternative component replacement for distributed object applications [11]. Most recently, yBase proposed compiler and profiling techniques to consider partitioning for resource-constrained miniaturized computers [16].

Several different forms of offloading of applications between mobile clients and surrogates have been examined for resource-constrained devices, most notably with ParcTab, MPad, Infopad, and Wit [31, 32, 33]. Rover took these approaches further by proposing a general-purpose relocatable object system that can be used to help mask slow network links [14]. However, partitioning of the application is left to the programmer using a specific framework, with placement controlled by the application. Spectra is investigating self-tuning automated placement as part of the general Aura vision [7, 24]. Most recently, work at Cornell on MagnetOS proposes similar work in the context of sensor networks [26].

Odyssey investigated adapting content "fidelity" to support content delivery to clients through feedback from the underlying platform [20]. GloMop proposes mobile clients using transcoding proxies to match content to client format and size restrictions [8]. We believe that these techniques are complementary to those proposed in this paper and are useful for tackling issues such as screen size and the absolute level of resource consumption. However, they require applications and transcoding proxies to be rewritten or reconfigured for every device type. PocketLinux attempts to limit this by parameterize applications in XML [30].

The most notable method for dealing with resource constraint problems is to reduce application functionality. Typical examples are the lightweight versions of the popular Office applications included with the PocketPC platform. However, this increases the burden on the user to understand the features and limitations of yet another version of the application.

Our work differs from related work by proposing that applications need not necessarily be rewritten for a diverse set of devices. Instead, an adaptive, transparently distributed platform can be used to increase application support by automatically using local surrogate server resources. Transparency can greatly minimize the cost of developing software on diverse devices. Fine-grained adaptive application partitioning allows the system to respond to changes in load and the environment. Using these features we are working towards beneficially assisting resource-constrained, mobile devices without requiring application rewriting.

8 Future Work

Based on our initial work, we plan to explore a variety of research avenues.

- **Enhance the prototype and emulator.** We plan to add global placement strategies to our prototype with the goal of placing objects on the most appropriate

device for the current conditions. This approach will involve moving objects from the surrogate to the client device, and will require coordinated execution monitoring, resource monitoring, and placement analysis. We also plan to study additional partitioning heuristics besides the modified MINCUT approach that is currently being used.

- **Simultaneously consider multiple constraints.** Currently, processing and memory constraints are being addressed separately. In the future, we plan to study them together. In addition, we plan to examine constraints on other resources such as network bandwidth and power.
- **Study the effect of garbage collection.** We plan to investigate the effect of garbage collection on the distributed platform. Some garbage collectors are conservative and leave some garbage at the end of a collection cycle. If more memory is needed, should garbage collection be performed again or should offloading occur? We plan to study a variety of garbage collection approaches to determine whether any of them work better with offloading and whether partioning can be integrated with garbage collection. We also plan to examine the effects of distributed garbage collection and the implications of offloading garbage.
- **Combine offloading and mobility.** We plan to examine application offloading in the context of mobility. New strategies will be needed to handle the situation where a user moves from one surrogate's region to that of another. For example, should references continue to be sent to the first surrogate, or should the objects on the first surrogate be migrated to the second surrogate?
- **Consider additional semantic information.** We plan to consider the benefits of exploiting additional information about the applications such as hints from users and developers, previously gathered profiling information, and high-level components like JavaBeans.
- **Examine additional applications.** We plan to study the behavior of additional applications in order to understand a broader range of application structures and workloads. We would also like to identify a realistic mix of applications that people would really use.

9 Conclusions

We have presented some of the problems that will be encountered as we move towards a world of diverse devices accessing applications as pervasive services. Based on these problems, we identified the need for a transparently distributed platform that can adapt to resource constraints by adaptively offloading computation and data to nearby machines. We believe that by using such a platform the burden on application developers can be reduced, while allowing acceptable service performance. We also believe that this approach is generally applicable to other situations such as wired and embedded networks where resources are also constrained.

To investigate our vision, we implemented both a prototype and an emulator to allow us to investigate the operation of a particular instantiation of our vision. Using a modified JVM, transparent remote execution, and execution statistics gathered at run time, we investigated the platform for memory and processing constraints. Our investigations show that for certain applications it is possible to transparently use remote resources to relieve resource constraints. Our results also show that using these resources can improve application performance and can allow applications to run that would have failed. We also believe that by intelligently adapting, the platform is able to use remote resources only when they improve the application's performance and provide benefit to the user.

Acknowledgements

We are indebted to Jason Flynn, Christos Karamanolis, Emre Kiciman, Todd Poynor, Erik Riedel, and Steve Richardson for reviewing this paper. Their comments significantly improved the content and presentation.

References

[1] Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H., and Secret, A., "The World-Wide Web," Communications of the ACM, 37(8):76–82, 1994.

[2] Chen, X., and Allan, V., "MultiJav: A distributed shared memory system based on multiple Java virtual machines", Proc. of the Conf. on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, June 1998.

[3] Composable High Assurance Trusted Systems (CHATS), www.arpa.gov/ito/research/chats/.

[4] DARPA ITO Ubiquitous Computing Program, www.arpa.gov/ito/research/uc/.

[5] Dertouzos, M. L., "The future of computing," Scientific American, July 1999.

[6] Esler, M., et al., "Next century challenges: data-centric networking for invisible computing: the Portolano project at the University of Washington," Proc. of 5th ACM/IEEE Conf. on Mobile Computing and Networking, Aug. 15–19, 1999, Seattle, WA.

[7] J. Flinn, D. Narayanan, and M. Satyanarayanan, "Self-tuned remote execution for pervasive computing,". Hot Topics on Operating Systems (HotOS-VIII), Schloss Elmau, Germany, May 2001.

[8] Fox, A., Gribble, S. D., Brewer, E. A., and Amir, E. "Adapting to network and client variability via on demand dynamic distillation," Proc. Seventh International ACM Conference on Architectural Support for Programming Languages and Operating Systems, pp. 160–170, Cambridge, MA, October 1996.

[9] Gribble, S., "The Ninja Architecture for Robust Internet-Scale Systems and Service," Special Issue of Computer Networks on Pervasive Computing, 2000. http://endeavour.cs.berkeley.edu/

[10] Hendrickson, B. and Kolda, T., "Graph partitioning models for parallel computing," Parallel Computing, 26:1519–1534, 2000.

[11] Högstedt, K., Kimelman, D., Rajan, V.T., Roth,T., Wegman, M., and Wang, N., "Optimizing Component Interaction," ACM Workshop on Optimization of Middleware and Distributed Systems (OM 2001), Snowbird, Utah, June 18, 2001.

[12] Hunt, G. C. and Scott, M. L., "The Coign Automatic Distributed Partitioning System." Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI '99), pp. 187–200, New Orleans, LA, February 1999, USENIX.

[13] IBM Pervasive Computing. http://www-3.ibm.com/pvc/.

[14] Joseph, A. D., et al., "Rover: A Toolkit for Mobile Information Access", in Proc. Fifteenth Symposium on Operating Systems Principles (SOSP), December 1995.

[15] Kindberg, T., et al., "People, Places, Things: Web Presence for the Real World," Proc. of the third WMCSA, 2000. see also HPL CoolTown, http://cooltown.hp.com/

[16] Li, Z., Wang, C., Xu, R., "Computation Offloading to Save Energy on Handheld Devices: A Partition Scheme," ACM CASES'01, Atlanta, November 16-17, 2001.

[17] Lo, H. Y., "M-mail: A case study of dynamic application partitioning in mobile computing," Master's thesis, Dept. of Computer Science, University of Waterloo, May 1997.

[18] Milojicic, D., Douglis, F. and Wheeler,R., "Mobility — Processes, Computers, and Agents," ACM Press. Addison-Wesley, Feb. 1999.

[19] Milojicic, D., Messer, A., Bernadat, P., Greenberg, I., Spinczyk, O., Beuche, D., Schröder-Preikschat, W., "Ψ — Pervasive Services Infrastructure," Technologies for E-Services, Second International Workshop, TES 2001, LNCS 2193, Rome, Italy, Sept. 14-15, 2001, pp. 187–200.

[20] Noble, B.D., et al, "Agile Application-Aware Adaptation for Mobility," Proc. of 16 SOSP, St. Malo, France, October 1997. Aura projects at CMU.

[21] Norman, Donald A., "The Invisible Computer," MIT Press, 1998.

[22] Object Management Group, "CORBA: Architecture and Specification," Aug. 1995.

[23] Philippsen, M., and Zenger, M., " JavaParty - transparent remote objects in Java," Concurrency: Practice and Experience, 9(11):1125--1242, November 1997.

[24] Satyanarayanan, M., "Research Challenges in Project Aura," keynote address at the Ninth IEEE International Symposium on High Performance Distributed Computing, Pittsburgh, PA, August 2000.

[25] Scheifler, R. W. and Gettys, J., "The X Window System," ACM Transactions on Graphics 16:8 (Aug. 1983), pp. 57–69.

[26] Sirer, E. G., Barr, R., Kim, T.W. D, Fung, I.Y.Y. "Automatic Code Placement Alternatives for Ad Hoc and Sensor Networks," Computer Science Technical Report 2001-1853, Cornell University, October 2001.

[27] Stoer, M. and Wagner, F., "A simple min-cut algorithm," Journal of the ACM, 44(4):585–591, July 1997.

[28] Sun Microsystems, "The .com Revolution Meets Consumer Appliances," available at: www.sun.com/990106/ces/.

[29] Sybase white paper, "Enabling e-Business Anywhere, Anytime: the Sybase Strategy."

[30] TransVirtual Technologies, PocketLinux, http://www.pocketlinux.com/.

[31] Truman, T., Pering, T., Doering, R., and Brodersen, R., "The infopad multimedia terminal: A portable device for wireless information access," IEEE Transactions on Computers, 47(10), Oct. 1998.

[32] Want, R., Schilit, B., Adams, N., Gold, R., Petersen, K., Goldberg, D., Ellis, J. R. and Weiser, M., "An overview of the ParcTab ubiquitous computing experiment," IEEE Personal Communications Magazine, 2(6):28–43, Dec. 1995.

[33] Watson, T., "Effective Wireless Communication through Application Partitioning," Fifth Workshop on Hot Topics in Operating Systems, Orcas Island, WA, May 1995.

[34] Weiser, M., "Some Computer Science Problems in Ubiquitous Computing," Communications of the ACM, July 1993, 75–84.