# Adaptive Offloading for Pervasive Computing

*Delivering a complex application on a resource-constrained mobile device is challenging. An adaptive offloading system enables dynamic partitioning of the application and efficient offloading of part of its execution to a nearby surrogate.*

**Xiaohui Gu and Klara Nahrstedt**
*University of Illinois at Urbana-Champaign*

**Alan Messer, Ira Greenberg, and Dejan Milojicic**
*Hewlett-Packard Laboratories*

Running complex applications on mobile devices is challenging due to their strict constraints on resources such as memory capacity, CPU speed, and battery power. A brute-force approach to accommodate device diversity is to rewrite applications for each individual device. But this can be expensive and even impossible if the application's source code is proprietary. Various projects have addressed this problem using application- or system-based adaptations.[1–4] However, these approaches often require significant degradation to an application's fidelity to fit that application into a mobile device. (See the "Related Work" sidebar for additional work in this area.) To deliver pervasive services without modifying the application or degrading its fidelity, we propose an adaptive offloading system that includes two key parts: a distributed offloading platform[5] and an offloading inference engine.[6]

## System architecture

Figure 1 shows our offloading system's architecture. Let's say a user wants to access a memory-intensive application on a resource-constrained mobile device, such as a PDA. The application can be either a distributed application, such as content retrieval from a remote server, or a local application such as an image editor. When the application memory requirement reaches or approaches the mobile device's maximum memory capacity, the system initiates offloading. The system partitions the application's program objects into two groups, offloading some to a powerful nearby surrogate to reduce the device's memory requirement. The offloading platform transparently transforms method invocations to offloaded objects into remote invocations. Our assumptions are that the application is written in an object-oriented language such as Java or C# and that the user's environment contains powerful surrogates and plentiful wireless bandwidth. A surrogate can be the user's personal laptop, an embedded server, or some other environmental host. With the proliferation of computing devices and wireless networks, such as IEEE Std. 802.11 for wireless LANs, we believe these assumptions are realistic.

There are two important decision-making problems for adaptive offloading: adaptive offloading triggering and efficient application partitioning. Figure 2 shows the architecture for the offloading inference engine. The engine doesn't require any prior knowledge to make offloading decisions. Instead, it acquires execution and resource information from the offloading platform's execution and resource monitors. To simultaneously meet multiple user requirements for offloading, the offloading inference engine uses a composite partition cost metric to select the best partition plan. The selected application partition plan indicates which program objects to offload to the surrogate and which to pull back to the mobile device during the new offloading


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

## Related Work

Our work is related to the Spectra project, which proposed a remote execution system for mobile devices used in pervasive computing.[1] Spectra can generate a distributed execution plan that balances the competing goals of performance, energy conservation, and application quality. The Puppeteer project supports adaptations without modifying applications.[2] The MONET research group proposed a dynamic-service-composition and distribution framework for delivering component-based applications in pervasive computing environments.[3] To support application-specific adaptation, this framework provides application developers with metalevel programming tools for deploying applications in pervasive computing environments.[4] However, this work assumes that applications are component based and that they've exported component interfaces to the system. The Coign project proposed a system to statically partition binary applications built from component object model (COM) components.[5] Unlike Coign, our approach performs dynamic runtime partitioning without any offline profiling. Furthermore, we don't assume applications are component based or written in COM components.


### REFERENCES

1. J. Flinn, S. Park, and M. Satyanarayanan, "Balancing Performance, Energy, and Quality in Pervasive Computing," *Proc. 22nd Int'l Conf. Distributed Computing Systems* (ICDCS 02), IEEE CS Press, 2002, pp. 217–226.
2. E. de Lara, D.S. Wallach, and W. Zwaenepoel, "Puppeteer: Component-Based Adaptation for Mobile Computing," *Proc. 3rd USENIX Symp. Internet Technologies and Systems* (USITS 01), Usenix Assoc., 2001, pp. 159–170.
3. X. Gu and K. Nahrstedt, "Dynamic QoS-Aware Multimedia Service Configuration in Ubiquitous Computing Environments," *Proc. 22nd Int'l Conf. Distributed Computing Systems* (ICDCS 02), IEEE CS Press, 2002, pp. 311–318.
4. X. Gu et al., "An XML-Based QoS Enabling Language for the Web," *J. Visual Language and Computing*, vol. 13, no. 1, 2002, pp. 61–95.
5. G.C. Hunt and M.L. Scott, "The Coign Automatic Distributed Partitioning System," *Proc. 3rd USENIX Symp. Operating System Design and Implementation* (OSDI 99), USENIX Assoc., 1999, pp. 187–200.


action. Extensive trace-driven simulations and prototype experiments show that our adaptive offloading system can efficiently support memory-intensive applications on a mobile device in a pervasive computing environment.

## Distributed offloading platform

Here, we introduce the distributed offloading platform, which includes application execution monitoring, resource monitoring, application-partitioning candidate generation, and transparent remote-procedure-call (RPC) platform support.

### Application execution monitoring

In the rest of the article, we use Java programs to illustrate our approach. The offloading platform characterizes application execution with a weighted directed graph called the *application execution graph* (AEG). Each graph node represents a Java class. We choose a class as the graph node for several reasons. First, a class represents a natural component unit for all object-oriented programs. Second, a class granule enables more precise offloading decisions than do coarser component granules. Third, classes don't require manipulation of a

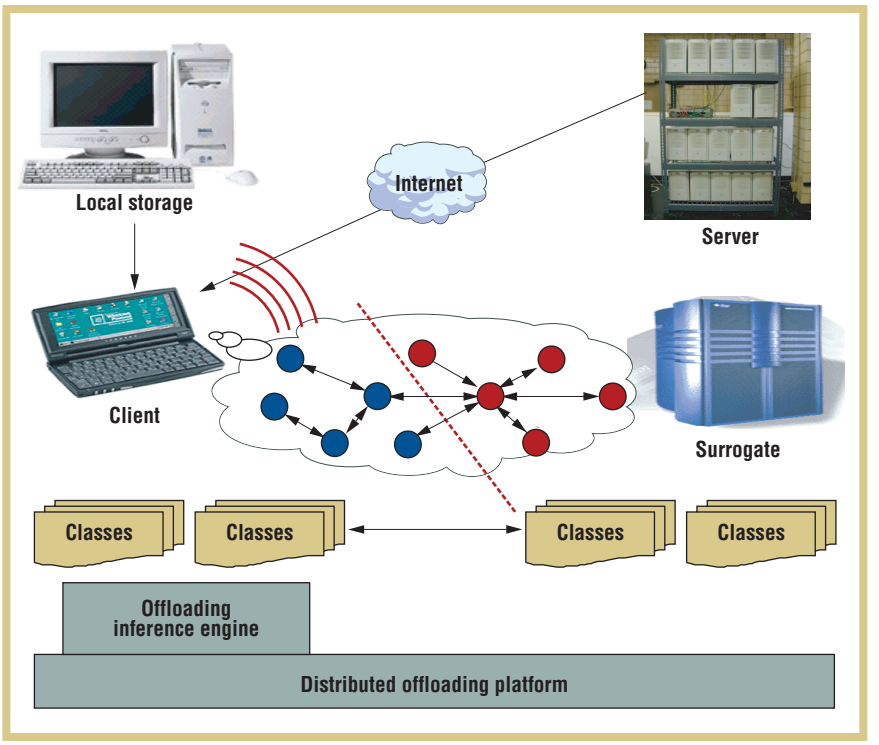



Figure 1. Adaptive offloading system architecture.

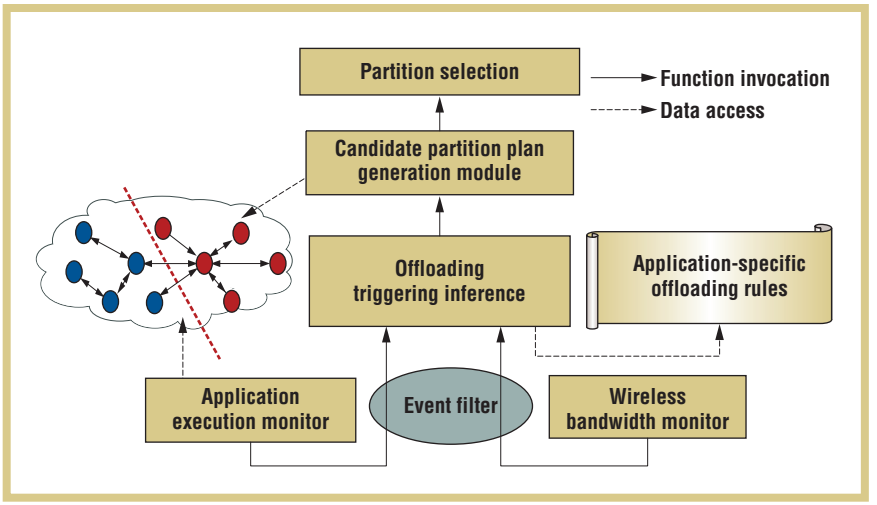



Figure 2. Offloading inference engine architecture.

large execution graph with too many fine-grained Java objects. For example, a simple image-editing Java program created 16,994 distinct Java objects during 174 seconds of execution.

The weight metrics associated with a graph node include the following:

- *Memory size* describes the amount of memory that the Java class's objects occupy.
- *AccessFreq* represents the number of times that other classes access the class's methods or data fields.
- *Location* describes whether the Java class's objects reside in the mobile device (local) or in the surrogate.
- *IsNative* indicates whether a class's objects can migrate from the mobile device to the surrogate. Some classes, such as those invoking device-specific native methods, must always execute on the mobile device.

Each graph edge represents the interactions and dependencies between two classes' objects. Two fields annotate each edge: *InteractionFreq* represents the number of interactions between the two classes' objects; *BandwidthRequirement* represents the total amount of information transferred between those objects. In the current prototype, the offloading platform execution monitor maintains the entire AEG on the mobile device. After the first application split, the system periodically collects execution information on the surrogate and merges it into the mobile device's execution graph.

### Resource monitoring

To adapt to resource changes, the offloading platform must monitor the resources of the mobile device, the surrogate, and the wireless network. The platform monitors the mobile device's and surrogate's available memory by tracking the amount of free space in the Java heap. The garbage collector of the Java Virtual Machine (JVM) provides this heap. We can estimate the wireless bandwidth and delay by passively observing ongoing traffic through the offloading platform or by actively measuring this information with measurement tools. Whenever some significant change happens (for example, the application consumes a certain amount of memory, or a sufficiently large wireless bandwidth fluctuation occurs), the offloading inference engine decides whether to trigger offloading.

### Application-partitioning candidate generation

The problem of optimal application partitioning (finding the minimum cut between two bounded sets) is nondeterministic-polynomial (NP)-complete.[7] However, we designed the offloading system for resource-constrained devices. Hence, the system uses an efficient partitioning heuristic derived from Frank Stoer's and Mechthild Wagner's MINCUT algorithm to generate a set of candidate partition plans.[8]

Let $G = (V, E)$ describe the current AEG, where $V$ is the set of nodes and $E$ is the set of edges. Each edge is associated with a cost value reflecting the interclass interactions and dependencies. Let $P_M$ represent the partition on the mobile device, and $P_S$ represent the partition on the surrogate. At first, the system initializes both $P_M$ and $P_S$ as empty. The partition candidate generation algorithm performs the following steps:

1. Merge all nodes that cannot migrate to the surrogate (those with their *IsNative* field set to true) into one node, $v_1$. Given $k$ edges from a node to these merged nodes, the system substitutes one edge for these $k$ edges. The new edge's cost equals the sum of the $k$ old edges' costs. If there are $n$ nodes after the merging, then let $P_M = \{v_1\}$, and $P_S = \{v_2, \ldots, v_n\}$.
2. Among the neighbors of $v_1$ representing the partition on the mobile device, select the one with the largest edge cost to $v_1$. If the selected node is $v_i$, merge $v_i$ with $v_1$, and move $v_i$ from $P_S$ to $P_M$. Next, consider the cut, $\langle P_M, P_S \rangle$, as one of the candidate partition plans. Then record this partition plan with the information about its partitioning cost and the two partitions.
3. Repeat step 2 until all nodes have merged with $v_1$.

The offloading inference engine can then select the best partitioning from these candidate partition plans according to its partition cost metric.

### Transparent RPC platform support

To support transparent partitioning, a mechanism for transparent RPCs between virtual machines is necessary. Java's existing support for remote exe-

Figure 3. Offloading rules for the offloading inference engine.

```
if (AvailMem is low) and (AvailBW is high) then NewMemSize := low;
if (AvailMem is low) and (AvailBW is moderate)
        then NewMemSize := average;
if (AvailMem is high) and (AvailBW is low) then NewMemSize := high;
```

cution, remote method invocation, doesn't provide transparent mapping of calls and objects into RPCs between machines. Hence, we modify Hewlett-Packard's Chai JVM to support transparent migration of objects between the mobile device and the surrogate. In a JVM, an object reference uniquely identifies each object. To support remote execution, we modify the JVM so that it flags object references to remote objects and intercepts accesses to those objects. Using these hooks, the offloading platform converts remote accesses into transparent RPCs between two JVMs. A JVM that receives an RPC request uses a pool of threads to execute that request on behalf of the other JVM. This approach doesn't migrate threads. Instead, invocations and data accesses follow the placement of objects.

However, to provide a single, transparent distributed platform between two JVMs, we must still address several issues. First, it's not possible to migrate Java native methods, because their implementations use languages other than Java and could have different implementations on different platforms. To solve this problem, the system directs native invocations back to the master JVM. This gives an application the appearance of executing on the mobile device even though part of its execution is on a surrogate. Second, application objects statically share some Java objects—for example, `System.properties`, which contains `<key, value>` pairs specifying information such as the host operating system's name. Therefore, to ensure consistency, the system directs all accesses to static data back to the master JVM. Third, each JVM has a private object reference name space and doesn't understand an object reference from another JVM. To overcome these name-space limitations, we modified the JVM so that it maps a reference from another JVM into its own name space. Each JVM keeps local references for remote objects as place holders. When one JVM invokes a method or accesses an object on the other JVM, the former uses its local object reference to send an operation referring to the object on the other JVM. The receiving JVM then maps the first JVM's local reference to its own real local reference for the object. Each JVM maintains its object reference mappings when objects and object references move between the two JVMs.

## Adaptive offloading inference engine

The offloading inference engine's two decision-making modules address the problems of triggering offloading and selecting partitioning. Offloading can add overhead to the application's execution. This overhead includes the cost of transferring objects between the mobile device and the surrogate, and the cost of performing remote data accesses and function invocations over a wireless network. One of the inference engine's functions is to minimize the offloading overhead while relieving the memory constraint on the mobile device.

### Offloading triggering inference

To perform offloading triggering inference, the offloading inference engine examines the application's current resource consumption and the available resources in the pervasive computing environment. It then decides whether to trigger offloading according to the user's offloading goals. If so, it decides what level of resource utilization to use on the mobile device—that is, how much memory to free up by offloading program objects to the surrogate. At first glance, it appears that we can solve the problem using a simple threshold-based approach. For example, we can hard-code threshold-based rules in the offloading inference engine, such as "If the mobile device's free memory is less than 20 percent of its total memory, trigger offloading and offload enough program objects to free at least 40 percent of the mobile device's memory." However, such a simple approach can't meet the challenges of adaptability, configurability, and stability.

The offloading inference engine addresses these challenges using the Fuzzy Control model,[9] which has proved effective for flexible, expressive, and stable coarse-grained application adaptations. Employing this approach in the offloading inference engine is novel because it applies the model to fine-grained application adaptation through runtime offloading. The Fuzzy Control model includes system- or application-developer-provided linguistic decision-making rules, membership functions, and a generic fuzzy inference engine based on fuzzy logic theory. This model lets us specify the offloading inference engine's offloading rules (see Figure 3).

The *AvailMem* and *AvailBW* variables are input linguistic variables representing the current available memory and available wireless bandwidth. The *NewMemSize* variable is the output linguistic variable representing the new memory use on the mobile device. If current system conditions match any of these rules, the offloading inference engine triggers offloading and derives the offloading memory size using the difference between current memory consumption and new memory use. If the difference is negative, then the system should pull back some program objects from the surrogate to adapt to low wireless bandwidth. The application developer or the user can easily configure the

TABLE 1
The benchmark applications we used in our experiments.

| Benchmark program | Description | Operation | Lifetime (s) | Peak memory requirement (Kbytes) |
|---|---|---|---|---|
| DIA | Java image editor | Open a 180-Kbyte picture image and drag it around | 174 | 8,949 |
| Biomer | Graphical molecular editor | Create three complex molecules | 261 | 10,668 |
| JavaNote | Java text editor | Open a 600-Kbyte text file | 268 | 7,972 |

offloading inference engine using the linguistic offloading rules.

However, to interpret these rules, the offloading inference engine must establish mappings between numerical and linguistic values for each linguistic variable. *Low*, *moderate*, and *high* are linguistic values. In fuzzy logic, a membership function defines the mapping between a linguistic variable's numerical value and its linguistic values.[9] The generic fuzzy inference engine implements the fuzzy-logic-based mapping and nonlinear adaptation process. It takes fuzzy sets' confidence values (low, moderate, and high) as inputs and generates outputs in a similar form. Hence, the offloading inference engine provides two functions to use the generic fuzzy inference engine:

- *Fuzzification* prepares input fuzzy sets for the generic fuzzy inference engine.
- *Defuzzification* converts the output fuzzy sets into actual offloading decisions, such as the new memory use on the mobile device.

### Application partition selection

The offloading inference engine selects the best application partitioning from a group of candidate partition plans generated by the offloading platform. First, the offloading inference engine considers the target memory use on the mobile device to rule out partition plans that don't meet this minimum requirement. Then the offloading inference engine selects the best partitioning from the remaining candidate partition plans by using a composite partition cost metric. To overcome mobile devices' memory constraints, users can stipulate multiple offloading requirements such as minimizing wireless bandwidth overhead, minimizing average response time stretch, and minimizing total execution time. The wireless bandwidth cost depends on two factors: migration of program objects during offloading, and remote function calls and remote data accesses. The total number of all remote invocations decides the average response time stretch. The total execution time stretch caused by offloading includes both migration delays and remote interaction delays.

To minimize the total offloading cost, the offloading inference engine comprehensively considers different interclass dependencies and interactions during application execution. For each neighbor node $v_k$ of $v_i$, $b_{i,k}$ denotes the total amount of data traffic transferred between $v_i$ and $v_k$, $f_{i,k}$ denotes the total interaction number, and $MS_k$ represents the memory size of $v_k$. Thus, the offloading inference engine uses a composite cost metric, $C_k = <b_{i,k}, f_{i,k}, MS_k>$ for the AEG edge between $v_i$ and $v_k$. We've shown that such a metric is effective for meeting different offloading requirements.[6] The partition cost of a candidate partition plan is the aggregated costs of all edges whose endpoints belong to different partitions. The offloading inference engine then selects the best partition plan that minimizes the partition cost.

## Trace-driven simulation experiments

We conducted trace-driven simulation experiments on an IBM Thinkpad T22 running Red Hat Linux 7.1. We first collected the execution traces using the three benchmark applications described in Table 1. The execution trace collector records method invocations, data field accesses, and object creations and deletions in the trace file by querying the instrumented JVM. For the JVM, we used HP's ChaiVM for embedded and real-time systems. Using the bandwidth measurement tool, we collected the wireless network traces on the same laptop equipped with an IEEE 802.11 WaveLAN network card. A desktop in an office room acted as the surrogate.

Our mobile roaming scenario took place in the Computer Science Department building of the University of Illinois at Urbana-Champaign. We obtained the wireless network trace by having a person with a mobile device begin in the office room, enter an elevator and ride it to the basement, and then exit the elevator and walk through the basement toward a stairway. The measured wireless bandwidth maintained around 4.8 Mbps until the person entered the elevator, when it dropped to about 2.4 Mbps. It then rose to about 3.6 Mbps when the person walked through the basement toward the stairway. Because the interface parameters used for function interactions and data accesses was small (less than 64 bytes for all execution traces), we measured only the round-trip time (RTT) for small data packets, which was about 2.4 ms.

The execution and network traces just described drove the simulator, which emulated a remote interaction by stretching the total execution time. The *remote data access delay* is the time duration between sending a request to the remote

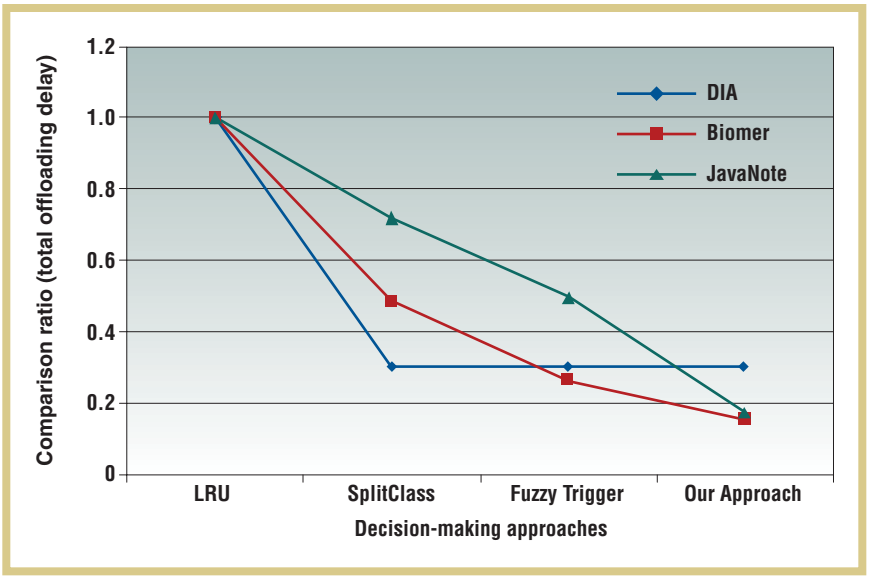



**Figure 4. Comparison of total offloading delay for four different decision-making algorithms.**

site and receiving the requested data, which approximately equals the RTT. The *remote function call delay* is the time to redirect a function request to the remote site, which is close to half of the RTT. We simulated the migration delay using the expression $MC/BW$ to increase execution time, where $MC$ is the memory size of all the classes to be migrated, and $BW$ is the current available bandwidth. We set the Java heap size to 8 Mbytes for the DIA and Biomer programs, and 7 Mbytes for JavaNote—according to their peak memory requirements.

We consider three performance metrics:

- *Total offloading delay* includes the delays for migration, remote data access, and remote function calls, which all extend an application's total execution time.
- *Average interaction stretch* represents the average interaction delay stretch caused by remote data accesses and remote function calls.
- *Total bandwidth requirement* is the sum of the migrated objects' total size and the total size of the parameters of the remote procedure calls that pass between two distributed JVMs during remote interactions.

For comparison, we implemented the least recently used (LRU) common-memory-management heuristic algorithm. The LRU algorithm adopts a simple offloading rule that triggers offloading when the available memory is lower than 5 percent of total memory, and sets a new target memory use of 80 percent of total memory. During application partitioning, the LRU algorithm offloads the classes that are least recently used according to each class's *AccessFreq* field. To enhance the LRU algorithm, we use the SplitClass algorithm, which splits large-class nodes into smaller-class ones with memory sizes smaller than 500 Kbytes. We then use the Fuzzy Trigger algorithm to enhance the SplitClass algorithm. Fuzzy Trigger replaces SplitClass's simple threshold-based rules with fuzzy-control-based offloading triggering. Finally, we run Our Approach, a complete offloading inference engine algorithm that enhances the Fuzzy Trigger algorithm with our composite-metric-based partition selection algorithm.

In an earlier work, we reported the performance comparisons between the composite and other simple metrics (for example, access frequency or bandwidth requirement alone).[6] We began by comparing the total offloading delay among the four different decision-making algorithms. We normalized the delay value of SplitClass, Fuzzy Trigger, and Our Approach to the LRU algorithm's value. The results in Figure 4 show that SplitClass can reduce the total offloading delay by as much as 60 percent compared to the simple LRU algorithm. The Fuzzy Trigger algorithm can further reduce this delay by up to another 44 percent. Finally, Our Approach, the offloading inference engine algorithm, consistently achieved the lowest offloading delay for all three applications.

We conducted a similar comparative study for the other two performance metrics: *average interaction stretch* and *total bandwidth requirement*. The results show that splitting large classes can reduce the average interaction stretch by as much as 90 percent and decrease the bandwidth requirement by as much as 54 percent, compared to the LRU algorithm. The Fuzzy Trigger algorithm reduced the average interaction stretch by up to 100 percent over that of SplitClass and decreased the bandwidth requirement by as much as 47 percent. Our Approach further reduced the average interaction stretch by as much as 62 percent over that of Fuzzy Trigger and decreased the bandwidth requirement by up to another 52 percent.

Finally, we compared the inference time of the naïve approach (that is, using simple, hard-coded rules) to our fuzzy-control offloading triggering approach. In the experiments just described, the fuzzy-control-based offloading inference required about twice the inference time used by the simple threshold-based triggering, which was about 0.06 ms.

## Prototype experiments

We developed a prototype of the distributed runtime offloading system. The prototype uses a modified version of HP's ChaiVM to monitor applications and resources, and uses that information to partition applications. In the prototype, we extended ChaiVM to support

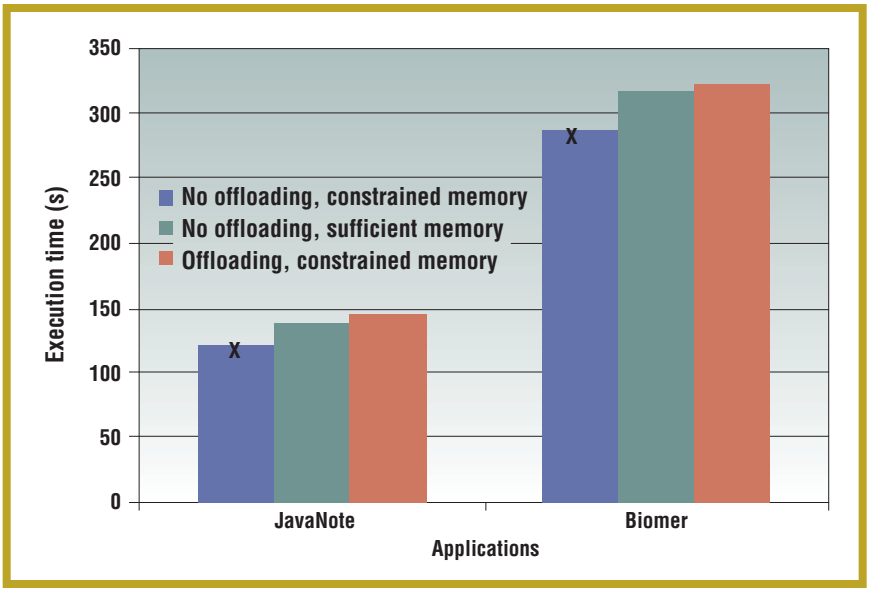



Figure 5. Execution time of two applications under the offloading prototype. The bars show the time until the executions exited. The X's indicate that the applications failed before completion.

transparent RPCs. We used an old 266-MHz HP laptop with an 11-Mbps IEEE 802.11b wireless card operating on a shared network to emulate a PDA-style mobile device. We used a 733-MHz HP Kayak PC workstation with 128 Mbytes of memory for the surrogate server. We attached the surrogate at 10 Mbps to the corporate wireless network using a network switch leading to the access point. Both machines ran Red Hat Linux 6.2 with the Linux 2.4.16 kernel. We used JavaNote and Biomer as our case study applications. We performed the same operations as in the simulation study described in Table 1.

To investigate the monitoring overhead, we evaluated the JavaNote application running on our prototype in a single-site mode that performs only monitoring. Because the system uses an 8-Mbyte Java heap, the application can execute without running out of memory. We also ran this configuration without enabling monitoring. Our unoptimized implementation using hashed arrays indicates that the monitoring overhead is about 7 percent of the total execution time.

Next, we evaluated the additional memory footprint required by the offloading platform. Because the virtual machine already maintained many of the states we needed, this memory footprint growth was mainly the result of the AEG. In several runs of the JavaNote application, we observed that a memory footprint of 15 Kbytes was necessary for the JavaNote application. We believe such results are acceptable for the offloading system but can be optimized further.

To evaluate memory-offloaded applications' runtime performance, we compared JavaNote's and Biomer's execution times on the offloading prototype against normal execution on an unmodified JVM. First, we used the prototype to determine the baseline memory that lets each application successfully run to completion. Then we selected a smaller memory size to emulate the memory constraint.

Figure 5 shows the experiment results. Without offloading, the constrained memory caused the application to crash when the heap became full. When running with enough memory, the applications took longer because they required more processing and were a better baseline for comparison to an offloaded run. When running with offloading under a constrained heap, the system ran between 1.5 percent and 5.7 percent more slowly than the baseline heap size without offloading. This increased execution cost is the difference between improved performance by borrowing memory from the surrogate and the cost of monitoring, remote accesses, and partitioning. Nevertheless, this low-performance cost is well worth the benefit of allowing applications to execute normally rather than crash because of insufficient memory.

Our extensive trace-driven evaluations show that with the offloading inference engine, runtime offloading can effectively relieve memory constraints for mobile devices with far lower overhead than other common approaches. Our prototype experiments indicate that the execution and memory overheads introduced by the adaptive offloading system are acceptable. Future research directions for the adaptive offloading system include applying the offloading approach to relieve other resource constraints on mobile devices, such as constraints related to CPU speed and battery lifetime, and supporting the use of multiple surrogates for offloading. P

## ACKNOWLEDGMENTS

We completed most of this work when Xiaohui Gu was a summer intern at HP Labs. NASA grant NAG 2-1406 and National Science Foundation grants 9870736, 9970139, and EIA 99-72884EQ partially supported this work. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, NASA, or the US government.

## REFERENCES

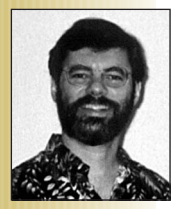
1. A. Fox et al., "Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives," *IEEE Personal Comm.*, Aug. 1998, pp. 10–19.

2. B.D. Noble, "System Support for Mobile, Adaptive Applications," *IEEE Personal Comm.*, Feb. 2000, pp. 44–49.

## the AUTHORS

**Xiaohui Gu** is a PhD candidate in computer science at the University of Illinois at Urbana-Champaign. Her research interests include pervasive computing, grid computing, and overlay networks. She received her MS in computer science from the University of Illinois at Urbana-Champaign. She is a student member of the IEEE. Contact her at the Thomas M. Siebel Center for Computer Science, 3101 SC, Univ. of Illinois at Urbana-Champaign, 201 N. Goodwin, Urbana, IL 61801; xgu@cs.uiuc.edu; http://cairo.cs.uiuc.edu/~xgu.

**Alan Messer** is a project manager at Samsung Electronics' Corporate Technology Operations. His research interests include pervasive computing, distributed systems, systems software, and consumer electronics. He received his PhD in computer science from City University, London. He is a member of the IEEE and the ACM. Contact him at Samsung Information Systems America, 75 W. Plumeria Dr., San Jose, CA 95134; alan.messer@samsung.com.



**Ira Greenberg** is an independent software consultant. His research interests include pervasive computing, distributed database systems, distributed middleware, and system security. He received his MS in computer science from the University of Massachusetts Amherst. Contact him at 953 Foxglove Dr., Sunnyvale, CA; ira.greenberg@comcast.net.

**Dejan Milojicic** is a senior scientist and a project manager at Hewlett-Packard Laboratories. His research interests include operating systems and distributed systems. He received his PhD in computer science from the University of Kaiserslautern, Germany. He is a member of the IEEE, the ACM, and Usenix. Contact him at HP Labs, MS 1183, Palo Alto, CA 94304; dejan@hpl.hp.com; www.hpl.hp.com/personal/Dejan_Milojicic.

**Klara Nahrstedt** is the Ralph and Catherine Fisher Associate Professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. Her research interests include multimedia middleware systems, quality of service, QoS routing, QoS-aware resource management in distributed multimedia systems, and multimedia security. She received her PhD in computer and information science from the University of Pennsylvania. She is a member of the IEEE and the ACM. Contact her at The Thomas M. Siebel Center for Computer Science, 3104 SC, Univ. of Illinois at Urbana-Champaign, 201 N. Goodwin, Urbana, IL 61801; klara@cs.uiuc.edu.


3. B.D. Noble et al., "Agile Application-Aware Adaptation for Mobility," *Proc. 16th ACM Symp. Operating Systems Principles* (SOSP 97), ACM Press, 1997, pp. 276–287.
4. E. de Lara, D.S. Wallach, and W. Zwaenepoel, "Puppeteer: Component-Based Adaptation for Mobile Computing," *Proc. 3rd USENIX Symp. Internet Technologies and Systems* (USITS 01), Usenix Assoc., 2001, pp. 159–170.
5. A. Messer et al., "Towards a Distributed Platform for Resource-Constrained Devices," *Proc. 22nd Int'l Conf. Distributed Computing Systems* (ICDCS 2002), IEEE CS Press, 2002, pp. 43–51.
6. X. Gu et al., "Adaptive Offloading Inference for Delivering Applications in a Pervasive Computing Environment," *Proc. 1st IEEE Int'l Conf. Pervasive Computing and Comm.* (PerCom 03), IEEE CS Press, 2003, pp. 107–114.
7. M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, 1979.
8. M. Stoer and F. Wagner, "A Simple Min-Cut Algorithm," *J. ACM*, July 1997, pp. 585–591.
9. B. Li and K. Nahrstedt, "A Control-Based Middleware Framework for Quality-of-Service Adaptations," *IEEE J. Selected Areas in Comm.*, Sept. 1999, pp. 1632–1650.


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

# How to Reach Us

### Writers
For detailed information on submitting articles, write for our Editorial Guidelines (pervasive@computer.org) or access www.computer.org/pervasive/author.htm.

### Letters to the Editor
Send letters to

Shani Murray, Lead Editor
*IEEE Pervasive Computing*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
pervasive@computer.org

Please provide an email address or daytime phone number with your letter.

### On the Web
Access www.computer.org/pervasive or http://dsonline.computer.org for information about *IEEE Pervasive Computing*.

### Subscription Change of Address
Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Pervasive Computing*.

### Membership Change of Address
Send change-of-address requests for the membership directory to directory.updates@computer.org.

### Missing or Damaged Copies
If you are missing an issue or you received a damaged copy, contact membership@computer.org.

### Reprints of Articles
For price information or to order reprints, send email to pervasive@computer.org or fax +1 714 821 4010.

### Reprint Permission
To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at copyrights@ieee.org.