Adaptive Offloading Inference for Delivering Applications in Pervasive Computing Environments [*]

Xiaohui Gu[†], Klara Nahrstedt, Alan Messer[‡], Ira Greenberg[§], Dejan Milojicic[¶]

Abstract

Pervasive computing allows a user to access an application on heterogeneous devices continuously and consistently. However, it is challenging to deliver complex applications on resource-constrained mobile devices, such as cell phones and PDAs. Different approaches, such as application-based or system-based adaptations, have been proposed to address the problem. However, existing solutions often require degrading application fidelity. We believe that this problem can be overcome by dynamically partitioning the application and offloading part of the application execution to a powerful nearby surrogate. This will enable pervasive application delivery to be realized without significant fidelity degradation or expensive application rewriting. Because pervasive computing environments are highly dynamic, the runtime offloading system needs to adapt to both application execution patterns and resource fluctuations. Using the Fuzzy Control model, we have developed an offloading inference engine to adaptively solve two key decision-making problems during runtime offloading: (1) timely triggering of adaptive offloading, and (2) intelligent selection of an application partitioning policy. Extensive trace-driven evaluations show the effectiveness of the offloading inference engine.

[*]The work was supported by a NASA grant under contract number NASA NAG 2-1406, an NSF grant under contract number 9870736, 9970139, and EIA 99-72884EQ. 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 U.S. Government.

[†]Xiaohui Gu and Klara Nahrstedt are affiliated with Department of Computer Science, University of Illinois at Urbana-Champaign. Email: {xgu, klara} @ cs.uiuc.edu

[‡]Alan Messer was affiliated with Hewlett Packard Laboratories when much of the work was done. He is now affiliated with Samsung R&D Labs, USA. Email: alan_messer@yahoo.com.

[§]Ira Greenberg was affiliated with Hewlett Packard Laboratories when much of the work was done. Email: ibg@earthlink.net

[¶]Dejan Milojicic is affiliated with Hewlett Packard Laboratories, Palo Alto, CA. Email: dejan @ hpl.hp.com

1 Introduction

Computer systems have evolved from the era of the mainframe, which is shared by many people, and the era of the personal computer, which is used by one person, to the era of pervasive computing where a single user possesses multiple heterogeneous mobile devices ranging from a laptop, to a personal digital assistant (PDA), to a cell phone. To accommodate device diversity, platform-independent language runtime systems, such as the Java Virtual Machine [10], and the Common Language Runtime in Microsoft .NET [2], have been developed and used for pervasive computing. Unfortunately, these solutions are often associated with high levels of resource.

Different approaches have been proposed to solve the problem by using application-based or system-based adaptations [4, 12, 13, 8]. However, these approaches often require degrading an application's fidelity to adapt it to resource-constrained mobile devices. Moreover, adaptation efficiency is often limited by coarse-grained approaches. However, it is expensive to rewrite an application according to the capacity of each mobile device. Hence, a fine-grained runtime offloading system, called adaptive infrastructure for distributed execution (AIDE) [11], has been proposed to solve the problem *without modifying the application or degrading its fidelity*. The key idea is to dynamically partition the application during runtime, and migrate part of the application execution to a powerful *nearby* surrogate device. In order to ensure efficient program execution under runtime offloading, two key decision-making problems must be addressed: (1) *when to offload*, and (2) *what policy to use to select objects to offload*.

In this paper, we present the *offloading inference engine* (OLIE), which makes intelligent offloading decisions to enable AIDE to deliver applications on resource-constrained mobile devices with minimum overhead. We identify two important decision-making problems solved by OLIE: (1) *timely triggering of adaptive offloading*, and (2) *intelligent selection of an application partitioning policy*. To solve the first problem, OLIE decides when to trigger the offloading action. If an offloading action is triggered, OLIE decides the new level of memory utilization to employ on the mobile device given the current resource conditions (e.g., wireless network bandwidth) in the pervasive computing environment. To solve the second problem, OLIE selects a proper application partitioning policy that decides which program objects should be offloaded to the surrogate and which program objects should be pulled back to the mobile device during an offloading action. To achieve both flexibility and stability, OLIE employs the Fuzzy Control model [9] for making offloading decisions. The Fuzzy Control model has previously been applied to coarse-grained application adaptations. The novelty of our approach is to apply the model to fine-grained application adaptation via runtime offloading.

One of the critical resource constraints of a mobile device is its strict memory limitation. In this paper, we focus on relieving the memory constraint of a mobile device by offloading application objects at runtime. This will allow a memory-intensive application to be used on a mobile device that otherwise would not be able to support the application without rewriting or fidelity degradation. Although runtime offloading can be used to relieve other types of resource constraints, such as CPU and energy constraints, these issues are outside the scope of this paper. Using extensive trace-driven experiments, we show that OLIE can effectively direct AIDE to support resource-intensive applications on a mobile device in a pervasive computing environment with minimum overhead.

This paper is organized as follows. Section 2 presents the system architecture and model. Section 3 describes the design and algorithms used by OLIE. Section 4 presents the performance evaluations. Section 5 discusses related work. The paper concludes in Section 6.

2 System Overview

In this section, we introduce the overall architecture of the distributed runtime offloading system, which is illustrated in Figure 1. The user wants to access a memory-intensive application on a resource-constrained mobile device, such as a PDA. The application might be a distributed application such as a content retrieval/editing application from a remote server or a local-area storage device, or simply a local application such as a graphic image editor. When the application memory requirement reaches or approaches the maximum memory capacity of the mobile device, an offloading action is triggered. The program objects on the mobile device are partitioned into two groups[1]. Some of the program objects are offloaded to a powerful nearby surrogate to reduce the memory requirement on the mobile device[2]. AIDE is responsible for properly transforming method invocations to objects that were offloaded to the surrogate into remote invocations [11]. OLIE does not require any prior knowledge about an application's execution or the resources of the system and the network to make offloading decisions. OLIE collects and analyzes all of the execution and resource information it needs at runtime.

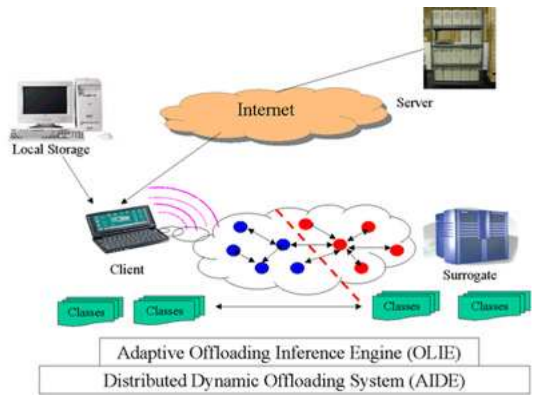
Internet
Server
Local Storage
Client
Surrogate
Classes
Classes
Classes
Classes
Adaptive Offloading Inference Engine (OLIE)
Distributed Dynamic Offloading System (AIDE)

Figure 1. Distributed dynamic offloading system architecture.

We now introduce the system models that are used to describe the runtime execution and resource consumption of the object-oriented application programs. Without loss of generality, we use Java programs in the rest of the paper as examples. Program execution information is represented as a connected weighted execution graph, as illustrated in Figure 2. Each node represents a *Java class* and is annotated with the amount of *memory* occupied by the objects of that class. We chose a *class* as the graph node because: (1) classes represent a natural component unit for all object-oriented programs, (2) classes enable more precise offloading decisions than coarser component granules such as JavaBeans, and (3) classes enable us to avoid manipulating a large execution graph with finer granules such as objects. (For example, a simple image-editing Java program that we examined created 16,994 distinct objects during 174 seconds of execution.)

Each class is annotated with an *AccessFreq* field, which represents how many times the methods or data fields of the class have been accessed. Currently, OLIE's offloading decisions are centralized. An application's execution graph is maintained as a whole on either the mobile device or the surrogate. After the first partitioning, the execution information on the remote side will be periodically collected and merged into the local execution graph. Hence, each node is also annotated with a *location* field to describe the

[1]Currently, we assume only a two-way cut between a mobile device and a surrogate device.

[2]We assume that the surrogate can be discovered in the local environment by some discovery service such as the Jini lookup service[1].

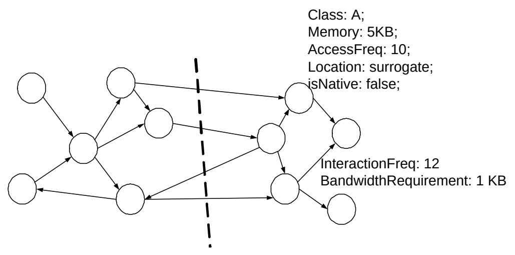

Figure 2. Illustration of our application program execution graph model.

class's current location. Some classes must always execute on the mobile device, such as classes that invoke device-specifi c native methods. Thus, each node is also annotated with an *isNative* fi eld to indicate whether the class can be migrated from the mobile device to the surrogate. Each edge represents the interactions between two classes. It is annotated with two fi elds, *InteractionFreq* and *BandwidthRequirement*. The *InteractionFreq* fi eld represents the number of interactions between two adjacent classes. The *BandwidthRequirement* fi eld represents the total amount of information transferred between two adjacent classes.

To make adaptive offloading decisions, OLIE monitors the mobile device, the surrogate device, and the network. The available memory on the mobile device is monitored by tracking the amount of free space in the Java heap, which is obtained from the Java Virtual Machine's (JVM) garbage collector. For simplicity, the wireless network conditions, including bandwidth and delay, are estimated by periodically invoking the ping system utility. Whenever some signifi cant changes happen (e.g., a big object is created or deleted, or a large wireless bandwidth fluctuation occurs), OLIE examines the current information about the memory utilization and available wireless network bandwidth to decide whether offloading should be triggered. If an offloading action is triggered, OLIE determines the new target memory utilization on the mobile device based on the current system/network conditions. Next, OLIE refers to the partitioning selection policy to decide which classes should be offloaded to the surrogate and which classes should be pulled back to the mobile device.

3 Design and Algorithms

In this section, we present the design details of OLIE. Although runtime offloading allows a memory-intensive application to be used on a mobile device, it also brings some overhead, such as: (1) migration costs, and (2) remote data access and function invocation delays caused by wireless communication. Hence, the major goal of OLIE is to make intelligent offloading decisions to relieve the memory constraint with minimum overhead. There are two major decision-making problems addressed by OLIE: (1) timely triggering of adaptive offloading, and (2) intelligent selection of an application partitioning policy.

3.1 Triggering of Adaptive Offloading

OLIE makes the offloading triggering decision based on the Fuzzy Control model [9]. The Fuzzy Control model includes: (1) a generic fuzzy inference engine based on fuzzy logic theory, and (2) decision-making rule specifi cations provided by system or application developers. For example, rules for making adaptive offloading triggering decisions can be specifi ed as follows.

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

The *AvailMem* and *AvailBW* variables are input linguistic variables that represent the current memory utilization and available wireless network bandwidth, respectively. The *NewMemSize* variable is the output linguistic variable representing the new memory utilization on the mobile device. *Low*, *moderate*, and *high* are linguistic values. The mappings between the numerical value (e.g., 500 KB) of a linguistic variable (e.g., available memory) and its linguistic values (e.g., low) are defi ned by the membership functions.

Figure 3 (a) illustrates a sample membership function for the linguistic variable *AvailMem*. In this example, if the numerical value of the *AvailMem* variable is within [0,800], the stochastic confi dence that linguistic variable *AvailMem* belongs to the set of linguistic value *low* is 100%. If the numerical value of the *AvailMem* variable is within [800,900], the stochastic confi dence that linguistic variable *AvailMem* belongs to the linguistic value *low* is the linear decreasing function from 100% to 0%. The intersection between different linguistic values (e.g., the values within [850,900], which are between *low* and *moderate*) represents uncertainty in stochastic confi dence and the result can belong to either linguistic value "low" or "moderate," but with different confi dence probabilities.

Membership functions are part of the rule specifi cations provided by the application developer. If the current system and network conditions match any specifi ed rule, an offloading action is triggered. In comparison to simple threshold-based offloading triggering, the Fuzzy Control model allows OLIE to implement more expressive and confi gurable triggering conditions.

3.2 Intelligent Partitioning Selection

Once an offloading action is triggered, OLIE refers to its partitioning selection policies to decide which classes

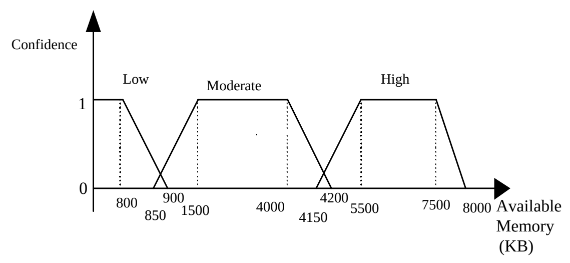

Figure 3. Illustration of a membership function for the linguistic variable *AvailMem*.

should be migrated to the surrogate and which classes should be pulled back to the mobile device given the new target memory utilization established by the triggering decision. OLIE solves this problem by comprehensively considering various inter-class dependencies and interactions, i.e., the *interactionFreq* and *BandwidthRequirement* fields annotating each edge in the execution graph illustrated in Figure 2.

OLIE first executes a coalescing process based on a MINCUT heuristic algorithm [14] to find all possible 2-way cuts of the execution graph. First, all the nodes that cannot be migrated to the surrogate (i.e., nodes with their *isNative* field set to true) are merged together into one node and placed in the first partition set, which belongs to the mobile device. The rest of the nodes form the second partition set, which belongs to the surrogate. Second, starting from the merged node, one of the neighbors of the first partition set is selected according to some inter-class dependency and interaction metrics, and merged into the first partition set. During the above coalescing process, each merging step generates a possible 2-way partitioning. This process is repeated until the first partition set contains all the nodes. OLIE then selects the 2-way partitioning that minimizes the predefined metrics.

We have designed several policies to select among different 2-way partitionings. For example, we can use the bandwidth requirement ($b_{i,j}$ from N_i to N_j) as the metric. In this case, we always merge into the first partition set the neighbor node with the largest bandwidth requirement in the second partition. This OLIE algorithm variation, called **OLIE_MB**, aims to minimize the wireless network bandwidth requirement caused by runtime offloading. We can also use the *interactionFeq* field in the execution graph ($f_{i,j}$) as the selection metric. This algorithm, called **OLIE_ML**, tries to minimize the interaction delay caused by remote data accesses and function invocations.

Finally, we consider a combined selection metric $C_k = < b_{i,k}, f_{i,k}, M_k >$ (where M_k is the current memory size of class k recorded in the execution graph). This algorithm, called **OLIE_Combined**, *comprehensively considers the*

```
$M_i$: memory size for Java class $i$;
$EG = \{N_0, N_1,... N_n\}$: execution graph;
$CMT$: the maximum memory size for a class node;
NewMemoryUtilization $\leftarrow -1$;

**while** offloading service is on
    **while** (no significant changes happen)
        perform executions and update *EG* accordingly;
        **while** ($\exists\, M_i > CMT$)
            create a new node to represent class $i$;
    //make the adaptive offloading triggering decision
    //set numerical values for all input linguistic variables
    SetLingVar();
    // map the numerical values to the linguistic values
    fuzzify();
    FuzzyInferencEngine();
    // map the linguistic values to the numerical values
    defuzzify();
    **if** (NewMemoryUtilization $== -1$)
    **then** offloading is not triggered;
    **else** //make the partitioning decision
        merge all non-offloadable classes into a node N;
        **while** (size(EG) $> 1$)
            merge (N, one of its neighbors $NB_j$);
            **if** ( current cut is better) bestPos $= NB_j$;
    $Partition_{mobiledevice} = \{ N_0,..., N_{bestPos}\}$;
    $Partition_{surrogate} = \{ N_{bestPos+1},..., N_n\}$;
```

Figure 4. Decision-making algorithm used by OLIE.

bandwidth requirement, interaction frequency, and memory size of the candidate neighbor during the coalescing process. We define the comparison of any two combined metrics C_k and C_l as follows. $C_k \geq C_l$ if and only if [3]:

$$w_1 \cdot \frac{b_{i,k} - b_{i,l}}{b^{max}} + w_2 \cdot \frac{f_{i,k} - f_{i,l}}{f^{max}} + w_3 \cdot \frac{M_l - M_k}{M^{max}} \geqslant 0 \quad (1)$$

where w_i ($1 \leq$ i ≤ 3) are nonnegative values such that $\sum_{i=1}^{3} w_i = 1$. Equation 1 states that we want to keep the classes that are most active (i.e., that have the largest interaction frequencies and bandwidth requirements) and occupy the smallest amount of memory on the mobile device, and offload the classes that are most inactive and occupy the largest amount of memory to the surrogate. To allow customization, we use $w_i (1 \leq i \leq 3)$ to represent

[3] b^{max}, f^{max}, and M^{max} represent the maximum values of the inter-class bandwidth requirement, inter-class interaction frequency, and class memory size, respectively.

the importance of the ith metric in making the offloading decision. These weights can be adaptively configured according to application requirements and user preference.

3.3 Class Granularity Problem

As mentioned before, classes were selected as the execution graph nodes. In practice, we found that the memory sizes of some classes are too large to be treated as single nodes. For example, the string class in the JavaNote application occupied 5.9MB during execution. If we offload these "big classes," they will cause large migration and remote invocation overhead. If we do not offload them, we cannot meet the memory constraint. Hence, if the memory size of a class exceeds a certain threshold, we create a new node in the execution graph to represent the class. All the objects belonging to the "big class" are distributed into two sets, each of which represents a node in the execution graph. Thus, the "big class" is split to enable more precise control of memory offloading. The complete decision-making algorithm used by OLIE is illustrated in Figure 4.

4 Performance Evaluation

In this section, we evaluate the performance of the dynamic offloading system under the direction of OLIE using extensive trace-driven simulations.

Evaluation methodology. The application execution traces are collected on a Linux desktop machine. The trace file records method invocations, data field accesses, and object creations and deletions by querying the instrumented JVM. Without loss of generality, we use ChaiVM, HP's personal JVM for embedded and real-time systems, for our experiments. The wireless network traces are collected using the Ping system utility on an IBM Thinkpad with an IEEE 802.11 WaveLAN network card.

The roaming scenario we selected for evaluation was conducted in the Digital Computing Laboratory building at the University of Illinois in Urbana-Champaign. The network trace was obtained by having a person start in a research lab on the second floor, enter an elevator and ride it to the basement, and then exit the elevator and walk to a stairway. The measured network bandwidth stays around 4.8Mbps until the person enters the elevator where it drops to about 2.4Mbps. It then rises to about 3.6Mbps when the person walks through the basement. Because the size of the parameters used for function interactions and data accesses is quite small ($<$ 64 bytes in all execution traces), we only measure the average round-trip time (RTT) for small packets, which is about 2.4 ms on average.

The simulator is driven by the execution and network traces described above. The simulator emulates a remote function invocation overhead by stretching the total execution time by $RTT/2$ because the application execution is delayed until the function on the remote site receives the invocation message. However, the simulator emulates a remote data access by stretching the total execution time by RTT because the application execution has to wait until the remote side receives the data request and then sends the data back to the local side. The migration overhead is simulated by increasing execution time using the equation, $\frac{\Sigma Memory_{\,classes\ to\ be\ migrated}}{current\ available\ bandwidth}$.

For comparison, we also implemented two other common approaches to making offloading decisions, *random* and least recently used (*LRU*). Unlike OLIE, which adaptively triggers offloading by comprehensively considering both memory and wireless network conditions, *random* and *LRU* adopt a simple fixed policy that triggers offloading when the available memory is lower than 5% of total memory and the new memory utilization constraint is less than 80% of total memory. In all of our experiments, OLIE uses the offloading triggering rules shown in Section 3.1. The weights $w_i (1 \leq i \leq 3)$ in Equation 1 are all set as $\frac{1}{3}$. Moreover, the *random* and *LRU* algorithms do not consider the "big class" problem, while OLIE splits the big class node into smaller ones with memory size smaller than 500KB. For the application partitioning problem, the *random* algorithm randomly selects some classes to keep on the mobile device and migrates the rest of the classes to the surrogate. The *LRU* algorithm offloads those classes that are least recently used according to the *AccessFreq* field of each class.

Results and analysis. Table 1 lists the descriptions of three applications used in our experiments. DIA is a simple Java image editor. For the execution trace, we opened a 180KB picture image and dragged it around. Biomer is a graphical molecular editor that can be used to visualize and edit the chemical structure of various molecules. For the execution trace, we drew three complex molecules. The application is both very memory intensive and CPU intensive. JavaNote is a Java text editing program. Its execution trace is extremely memory intensive because we use JavaNote to read a very large text file (600KB). This causes JavaNote to keep creating and deleting objects of the string class. We set the maximum memory capacity of the mobile device (i.e., the Java heap size) to 8MB for DIA and Biomer, and to 7MB for JavaNote, according to their peak memory requirements. [4]

The first performance metric we use is the *total offloading overhead*, which consists of migration overhead, remote data access overhead, and remote function call overhead.

[4]Although the memory capacity of mobile devices will continue to increase, the memory limitation will still exist when the user runs multiple applications or multiple instances of the same application (e.g., multiple editors).

Program	Description	Lifetime	Peak Mem	oversizing	#classes	#objects
DIA	simple image editor memory intensive	174 s	8,949 KB	11% (8MB)	100	16,994
Biomer	graphical molecular editor, both memory and cpu intensive	261 s	10,668 KB	34% (8MB)	105	32,118
JavaNote	text editor, open a 600 KB file, extremely memory intensive	268 s	7,972 KB	14% (7MB)	85	13,122

Table 1. *Descriptions of the application suite used in our experiments*

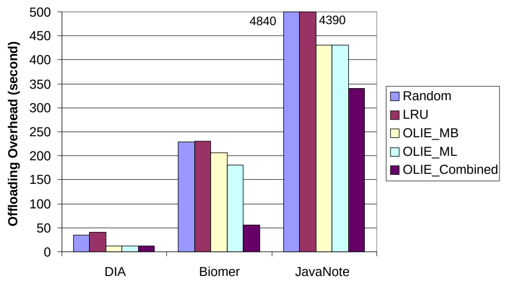

Figure 5. Total offloading overhead by the five different offloading algorithms. Random and LRU cause more than 4000 seconds of overhead for JavaNote.

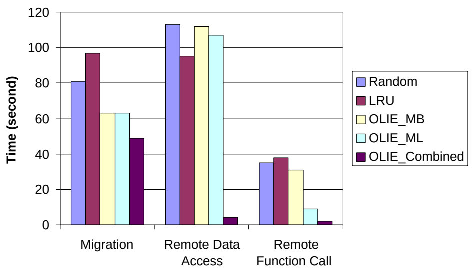

Figure 6. Offloading overhead breakdowns for the Biomer application by the five different offloading algorithms.

These overheads extend the total execution time of the three applications. Figure 5 illustrates the *total offloading overheads* of the three applications by *random*, *LRU*, and the three variations of the OLIE algorithm, respectively.

The results show that the OLIE algorithms consistently select classes more accurately, thereby achieving much less overhead than the *random* and *LRU* policies. This occurs because, unlike OLIE, both *random* and *LRU* do not consider inter-class dependencies and cannot adaptively trigger the offloading action according to fluctuations in the wireless network. Moreover, *random* and *LRU* do not solve the "big class" problem, which causes large migration overhead during offloading. Compared to the other two OLIE variations, OLIE_Combined further reduces the offloading overhead, especially for the very memory-intensive applications, Biomer and JavaNote. The performance improvements by OLIE_Combined can be as high as 66% for DIA, 73% for Biomer, and 94% for JavaNote when compared with *random* and *LRU*. This demonstrates that the combined selection metric (Equation 1) very effectively selects the proper application partitioning.

For a detailed analysis, Figure 6 shows the three different offloading overheads (migration, remote data access, and remote function call) for the Biomer application for the five different offloading algorithms. The first four algorithms achieve better or worse performance for the three different overheads because they only consider partial inter-classes depedencies/interactions. However, OLIE_Combined uniformly achieves the lowest overheads for both migration and remote interaction delay overhead, especially for the latter. (Remote interactions consist of remote data accesses and remote function calls). The reason is that the OLIE_Combined algorithm comprehensively considers all inter-class interactions (i.e., access frequency and bandwidth requirements), which guides the offloading system to properly split the application into two least-connected partitions.

The second considered performance metric is the *average interaction delay*, which is measured by $\frac{(\Sigma RemoteDataAccess*RTT+\Sigma RemoteFunctionCall*RTT/2)}{\Sigma(DataAccess+FunctionCall)}$. This metric represents the average interaction time stretch caused by remote data accesses and remote function calls. For each remote data access, the interaction delay is the time required to send the request to the remote site and to receive the requested data from the remote site, which

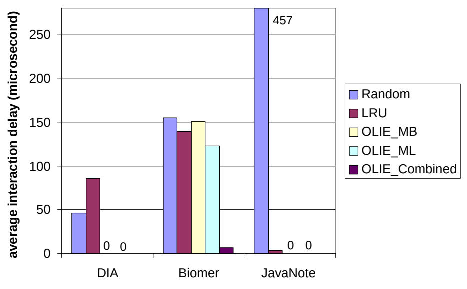

Figure 7. Average interaction delay of the three applications by the five different offloading algorithms. The random algorithm causes more than 400 microseconds of interaction delay for the JavaNote application.

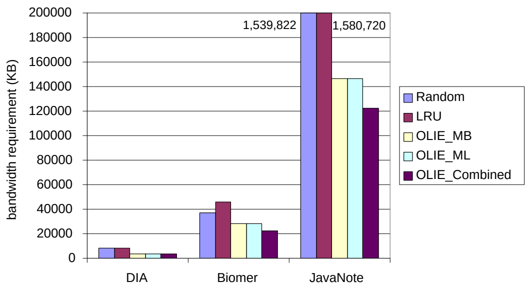

Figure 8. Total network bandwidth requirements of the three traces by the five policies. The random and LRU algorithms cause more than 1,500MB of total network traffic for the JavaNote application.

is close to the RTT of the wireless connection. For each remote function call, the interaction delay is the time required to send the function request and its parameters to the remote site, which is close to half of the RTT for the wireless connection.

The *average interaction delay* metric is very important for interactive applications because they are sensitive to the response time of each interaction. We do not want dynamic offloading to significantly compromise the responsiveness of the application. Figure 7 illustrates the average interaction delays for the five different offloading algorithms. The results again show that OLIE_Combined achieves the best performance. The delay reduction can be as high as 95% for Biomer and 100% for DIA and JavaNote. The reason for the delay reduction is that OLIE_Combined reduces remote data accesses and remote function calls to the minimum, even zero during certain time periods, by explicitly considering interactions between classes during offloading.

The third performance metric is the *total bandwidth requirement*, which is measured as the sum of the total size of the migrated objects and the total size of the parameters that are passed during remote interactions. Figure 8 shows the bandwidth requirements for the five different offloading algorithms. We observe that OLIE always requires much less bandwidth than the other two approaches, and that OLIE_Combined always requires less bandwidth than the other two OLIE variations.

5 Related Work

Recently, both application-based and system-based adaptations have been proposed to overcome resource constraints and environmental changes (e.g., wireless network fluctuations). The Odyssey project [12, 13] introduced an application-aware adaptation service within the end host to accommodate resource changes, such as wireless network bandwidth fluctuations. Fox et. al. proposed an application-based adaptation mechanism to meet client and network variations, called distillation. However, both solutions require modifying applications. The Puppeteer project [8] supports adaptation without modifying applications. However, they assume that the application is already written in a component-based fashion and has exported component interfaces to the system. In the Gaia project [5], we proposed a dynamic service composition and distribution framework for delivering component-based applications in a pervasive computing environment. To support application-specific adaptation, application developers can use meta-level programming tools for deploying their applications in pervasive computing environments [6, 16, 3].

One of the key differences between our dynamic offloading system and the above work is that pervasive application delivery can be realized without modifying the application or assuming that the application is developed in a component-based fashion and exports interfaces to control the components. Instead, a monolithic application that was not designed for distributed execution is dynamically partitioned at runtime based on its execution history and system/network resource information.

Other closely related work includes application partitioning under different contexts. The Coign [7] project proposed a system to statically partition binary applications built from COM components. Unlike Coign, our approach performs dynamic runtime partitioning without any offline profiling. Furthermore, we do not assume a component-based application and enable mobile delivery for any application, even a complex monolithic application. More recently, Teodorescu et. al. [15] presented a system to support mobile Java program deployment by partitioning

Java program execution between system nodes and mobile devices. The system nodes prepare a Java application for execution on a mobile device by generating device-specific native code using a Just-In-Time compiler and a customized Java runtime system. We believe that their approach is complementary to our work. However, they did not address the problem of runtime partitioning.

6 Conclusion

We have presented an adaptive offloading inference engine (OLIE) for making intelligent decisions in the runtime offloading system. OLIE directs the runtime offloading system to efficiently enable pervasive application delivery without degrading application fidelity or expensive application rewriting. OLIE makes offloading decisions without assuming any prior knowledge about the application's execution or system/network conditions in pervasive computing environments.

The major contributions of this paper include: (1) identifying two key decision-making problems for dynamic offloading, namely timely triggering of adaptive offloading and intelligent selection of an application partitioning policy; (2) applying the Fuzzy Control model to OLIE to achieve both flexibility and stability in making the adaptive offloading decision; and (3) proposing three policies for selecting application partitions that consider various interclass dependencies and interactions. Our extensive trace-driven evaluations show that with OLIE, the runtime offloading system can effectively relieve memory constraints for mobile devices with much lower overhead than other common approaches.

Future research directions for the dynamic runtime offloading system include applying the idea to constraints on other mobile device resources such as CPU and power.

7 Acknowledgment

We wish to thank T.J. Guili at Stanford University for helping us collect execution traces. We also wish to thank Deqing Chen at the University of Rochester for his helpful discussions. Finally, we wish to thank the anonymous reviewers for their helpful suggestions.

References

[1] The Jini Network Technology. *See website at http://wwws.sun.com/software/jini/.*

[2] The .NET Common Language Runtime. *See website at http://msdn.microsoft.com/net.*

[3] V. Adve, V. Vi Lam, and B. Ensink. Language and Compiler Support for Adaptive Distributed Applications. *ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001), Snowbird, Utah*, June 2001.

[4] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives. *IEEE Personal Communications, Special issue on adapting to network and client variability*, August 1998.

[5] X. Gu and K. Nahrstedt. Dynamic QoS-Aware Multimedia Service Configuration in Ubiquitous Computing Environments. *Proc. of 22nd IEEE International Conference on Distributed Computing Systems (ICDCS 2002), Vienna, Austria*, July 2002.

[6] X. Gu, D. Wichadakul, and K. Nahrstedt. Visual QoS Programming Environment for Ubiquitous Multimedia Services. *Proc. of IEEE International Conference on Multimedia and Expo 2001(ICME2001), Tokyo, Japan*, August 2001.

[7] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. *Proc. of the 3rd USENIX Symposium on Operating System Design and Implementation (OSDI'99)*, February 1999.

[8] E. Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based Adaptation for Mobile Computing. *Proc. of 3rd USENIX Symposium on Internet Technologies and Systems*, March 2001.

[9] B. Li and K. Nahrstedt. A Control-based Middleware Framework for Quality of Service Adaptations. *IEEE Journal of Selected Areas in Communications, Special Issue on Service Enabling Platforms*, 17(9), September 1999.

[10] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. *Addison-Wesley, second edition*, 1999.

[11] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T.J. Guili, and X. Gu. Towards a Distributed Platform for Resource-Constrained Devices. *Proc. of IEEE 22nd International Conference on Distributed Computing Systems (ICDCS 2002), Vienna, Austria*, July 2002.

[12] B. Noble. System Support for Mobile, Adaptive Applications. *IEEE Personal Coomunications*, 7(1), February 2000.

[13] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. Eric Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. *Proc. of the 16th ACM Symposium on Operating Systems Principles (SOSP), Saint-Malo, France*, October 1997.

[14] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM, 44(4), pp.585-591*, July 1997.

[15] R. Teodorescu and R. Pandey. Using JIT compilation and configurable runtime systems for deployment of Java programs on ubiquitous devices. *Proc. of 3rd International Conference on Ubiquitous Computing (Ubicomp 2001), Atlanta, Georgia*, September 2001.

[16] D. Wichadakul, X. Gu, and K. Nahrstedt. A Programming Framework for Quality-Aware Ubiquitous Multimedia Applications. *Proc. of ACM Multimedia 2002, Juan Les Pins, France*, December 2002.