MAUI: Making Smartphones Last Longer with Code Offload

Eduardo Cuervo[†], Aruna Balasubramanian[‡], Dae-ki Cho[*],
Alec Wolman[§], Stefan Saroiu[§], Ranveer Chandra[§], Paramvir Bahl[§]
[†]Duke University, [‡]University of Massachusetts Amherst, [*]UCLA, [§]Microsoft Research

ABSTRACT

This paper presents MAUI, a system that enables fine-grained energy-aware offload of mobile code to the infrastructure. Previous approaches to these problems either relied heavily on programmer support to partition an application, or they were coarse-grained requiring full process (or full VM) migration. MAUI uses the benefits of a managed code environment to offer the best of both worlds: it supports fine-grained code offload to maximize energy savings with minimal burden on the programmer. MAUI decides at runtime which methods should be remotely executed, driven by an optimization engine that achieves the best energy savings possible under the mobile device's current connectivity constrains. In our evaluation, we show that MAUI enables: 1) a resource-intensive face recognition application that consumes an order of magnitude less energy, 2) a latency-sensitive arcade game application that doubles its refresh rate, and 3) a voice-based language translation application that bypasses the limitations of the smartphone environment by executing unsupported components remotely.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/Server*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

code offload, partitioning, smartphones, energy management

1. INTRODUCTION

One of the biggest obstacles for future growth of smartphones is battery technology. As processors are getting faster, screens are getting sharper, and devices are equipped with more sensors, a smartphone's ability to consume energy far outpaces the battery's ability to provide it. Unfortunately, technology trends for batteries indicate that these limitations are here to stay and that energy will remain the primary bottleneck for handheld mobile devices [34]. Given the tremendous size of the mobile handset market, solving the energy impediment has quickly become the mobile industry's foremost challenge [14].

One popular technique to reduce the energy needs of mobile devices is *remote execution*: applications can take advantage of the resource-rich infrastructure by delegating code execution to remote servers. For the last two decades, there have been many attempts to make mobile devices use remote execution to improve performance and energy consumption. Most of these attempts took one of the following two approaches to remote execution. The first approach is to rely on programmers – to specify how to partition a program, what state needs to be remoted, and how to adapt the program partitioning scheme to the changing network conditions [9, 10, 1, 3]. This approach leads to large energy savings because it is fine-grained – applications can remote only the sub-parts that benefit from remote execution. For example, an application that both decodes and plays video would remote only the decoder, which is the CPU-intensive part, without remoting any of the screen-intensive parts. The second approach is to use full process [31] or full VM migration [6, 37] in which individual applications (or entire OS's in the case of VMs) can migrate to the infrastructure. This approach reduces the burden on programmers because applications do not need to be modified to take advantage of remote execution; instead, all their code and program state is automatically sent to the remote infrastructure.

We present MAUI, an architecture that combines the benefits of these two approaches: it maximizes the potential for energy savings through fine-grained code offload while minimizing the changes required to applications. MAUI achieves these benefits by using several properties of today's managed code environments (we use the Microsoft .NET Common Language Runtime (CLR) [36] for MAUI, although Java would offer the same properties). First, MAUI uses *code portability* to create two versions of a smartphone application, one of which runs locally on the smartphone and the other runs remotely in the infrastructure. Managed code enables MAUI to ignore the differences in the instruction set architecture between today's mobile devices (which typically have ARM-based CPUs) and servers (which typically have x86 CPUs). Second, MAUI uses *programming reflection* combined with *type safety* to automatically identify the remoteable methods and extract only the program state needed by those methods. Third, MAUI profiles each method of an application and uses *serialization* to determine its network shipping costs (i.e., the size of its state). MAUI combines the network and CPU costs with measurements of the wireless connectivity, such as its bandwidth and latency to construct a linear programming formulation of the code offload problem. The solution to this problem dictates how to partition the application at runtime to maximize energy savings under the current networking

MobiSys'10, June 15–18, 2010, San Francisco, California, USA.

conditions. Because serialization can be done at runtime, MAUI's application profiling is run continuously to provide up-to-date estimates of each method's costs. This continuous profiling makes MAUI's program partitioning scheme highly dynamic. The combination of code portability, serialization, reflection, and type safety allows MAUI to offer dynamic fine-grained code offload with minimal burden on the programmer.

In implementing MAUI, we discovered a number of unforeseen challenges to implementing program partitioning for mobile applications. One such challenge is that using power-save mode (PSM) when transferring state remotely can *hurt* the overall energy consumption of the application when the latency to the server is low. Moreover, PSM mode helps save energy but only when latencies approach the sleep interval (today's hardware uses 100 ms). Another unforeseen challenge is related to using profiling to estimate the energy savings of code offload. On one hand, profiling the state transfer overhead of a method each time it is called can provide the freshest estimate; on the other hand, the cost of this profiling is not negligible and it can impact the overall application's performance. Throughout our presentation of the MAUI architecture, we provide in-depth descriptions of the lower-level implementation challenges discovered in an effort to guide future implementations of remote execution to avoid them.

Our evaluation shows that MAUI offers significant energy and performance benefits to mobile applications. With MAUI, we partitioned four smartphone applications that have different latency and computational requirements. Our results show that (1) MAUI improves the energy consumption of resource-intensive applications, such as face recognition, by almost one order of magnitude (i.e., a factor of eight); (2) MAUI allows latency-sensitive applications, such as games, to more than double their screen refresh rates; and (3) MAUI enables the development of applications that bypass the limitations of today's smartphone environments; we developed a real-time voice-based Spanish-to-English translator even though today's smartphones do not provide enough RAM to run our advanced speech recognition engine.

The remainder of this paper presents the design, implementation, and evaluation of MAUI. We start by raising four questions that investigate the benefits of remote execution given today's mobile landscape (Section 2). Next, we present a high-level description of MAUI's architecture (Section 3). The next three sections describe MAUI's program partitioning mechanism (Section 4), its ability to profile mobile applications and the wireless environment (Section 5), and its dynamic optimization framework (Section 6). The paper's final sections evaluate MAUI's performance (Section 7), summarize the work related to remote execution for mobile applications (Section 8), and conclude (Section 9).

2. THE NEED FOR REMOTE EXECUTION IN TODAY'S MOBILE LANDSCAPE

This section's goal is to examine remote execution through the lens of today's mobile landscape. Today's mobile users expect their smartphones to run sophisticated applications, they have almost ubiquitous access to 3G connectivity, and their computing applications are moving to the cloud. How do these trends affect the remote execution problem? The remainder of this section examines these issues by raising three questions:

1. MAUI's goal is to mitigate the energy problem for mobile handhelds, the mobile industry's foremost challenge. *How severe is the energy problem on today's handheld devices?*

2. Mobile handheld devices are increasingly equipped with wireless wide-area network interfaces, such as 3G. Unlike Wi-Fi, 3G offers ubiquitous connectivity. However, 3G connections are known to suffer from very long latencies and slow data transfers [15]. For code offload, this poor performance may lead to increased energy consumption. *How energy efficient is using 3G for code offload?*

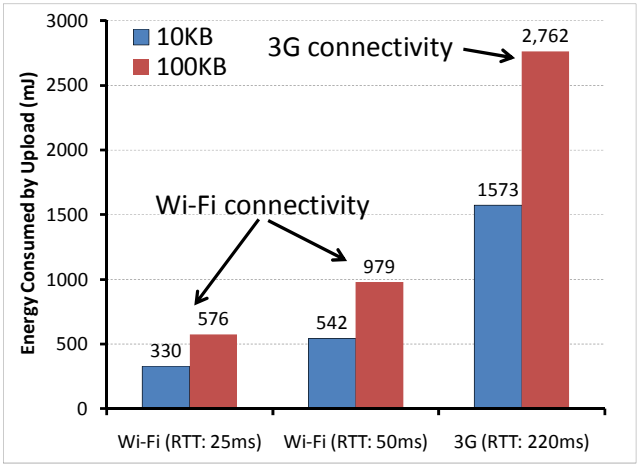

Figure 1: The Energy Consumption of Wi-Fi Connectivity vs. 3G Connectivity ***We performed 10 KB and 100 KB uploads from a smartphone to a remote server. We used Wi-Fi with RTTs of 25 ms and 50 ms (corresponding to the first two sets of bars) and 3G with an RTT of 220 ms (corresponding to the last bar).***

3. With cloud computing, industry is building massive infrastructure to offer highly available resources. Offloading mobile code to the cloud is an attractive possibility for systems like MAUI. However, typical round-trip times (RTTs) to cloud servers are on the order of tens of milliseconds. An alternative for MAUI is to use *nearby* servers [37] (with RTTs less than 10ms), such as a server co-located with a Wi-Fi access point or a user's personal desktop in the home. *How sensitive is the energy consumption of code offload to the RTT to the remote server?*

2.1 How Severe is the Energy Problem on Today's Mobile Handheld Devices?

The two main contributors to the energy problem for today's mobile devices are: 1) limited battery capacity, and 2) an increasing demand from users for energy-hungry applications.

To illustrate the severity of the battery limitations of today's handheld devices, we performed the following experiment. We created a synthetic resource-intensive application, and measured how long it takes to completely drain the battery of a new smartphone (the HTC Fuze with a 1340 maH battery, released in November 2008). Our synthetic application performs a large bulk-data transfer over the Wi-Fi interface, consumes the entire CPU, and keeps the display backlight on. Although it is synthetic, this application is not unrealistic: a streaming video decoder would also heavily utilize the CPU and the Wi-Fi network with the screen on. When running this application, we found that the fully charged battery drained after only one hour and twenty minutes.

In today's mobile handheld market, user demand is increasing for three categories of resource intensive applications. First, video games are very popular (for example, many of the top applications on Apple's AppStore are video games) and they have large energy demands. Second, mobile users are increasingly watch-

ing streaming video. For example, one third of iPhone users accessed YouTube [24] and the number of mobile uploads to YouTube increased 400% during the first six days after iPhone 3GS's release [21]. Finally, mobile devices are increasingly equipped with new sensors that produce continuous streams of data about the user's environment. New applications that rely on continuous processing of sensor data are emerging, such as car navigation systems, pedometers, and location-based social networking. Today, developers restrict their applications by making judicious use of sensors due to energy concerns (e.g., obtaining a GPS reading is expensive).

The trends in battery technology make it unlikely that the energy problem will disappear in the future. Although the chemical composition of batteries has changed significantly over the years, these changes have had less impact on battery capacity than on other aspects of battery technology, such as recharging and "memory effects". While recent research has suggested using newer chemicals [32], these changes are unlikely to significantly improve battery lifetime. Another proposed technology is the use of "fuel cells", yet even the most optimistic predictions of their performance do not suggest that energy concerns are likely to disappear [27].

2.2 How Energy Efficient is 3G for Code Offload?

Handheld devices have two alternatives for code offload: use either 3G or Wi-Fi. The primary benefit of 3G over Wi-Fi is the near-ubiquitous coverage it provides. However, recent studies have shown that round-trip times over 3G are lengthy and bandwidth is limited. RTTs are consistently on the order of hundreds of milliseconds and they even reach seconds [15]. Such long latencies may make code offload slow and expensive. For example, sending just a few tens of kilobytes to the cloud requires multiple RTTs, and thus the transfer time may take on the order of seconds.

We performed a simple experiment to contrast the energy consumption of 3G versus Wi-Fi. We setup the HTC Fuze smartphone do to two small code uploads (10 KB and 100 KB) to a remote server using either 3G or Wi-Fi. The server uses a driver that adds a controlled amount of queuing delay to the network path between the smartphone and the server. We then evaluated two scenarios: (1) the smartphone using Wi-Fi to reach the server (and adding an additional 25 ms or 50 ms of queuing delay); and (2) the smartphone using 3G (with a measured RTT of 220 ms). Figure 1 shows the energy consumed by the smartphone during the two uploads. Using 3G, the smartphone consumes three times as much energy as it does using Wi-Fi with a 50 ms RTT, and five times the energy of Wi-Fi with a 25 ms RTT. This implies that our HTC Fuze phone, with a 1340 mAH battery, will last less than 2 hours if the phone repeatedly downloads a 100 KB file over a 3G connection.

Such drastic discrepancies in energy consumption made us reconsider using 3G for MAUI. Despite the coverage benefits, 3G has poor performance and high energy consumption. For a system aimed at saving energy (like MAUI), the cost of using 3G today is almost prohibitive. In contrast, Wi-Fi has much better performance and lower energy consumption. MAUI can use either Wi-Fi or 3G for code offload.

2.3 How Sensitive is the Energy Consumption of Code Offload to the RTT to the Server?

As the previous experiment showed, the longer the RTT to the remote server, the higher the energy consumed by the code offload. To investigate this latency versus energy trade-off in more-depth, we performed a series of experiments where we offloaded code to a remote server via a link with increasing RTTs from 10 up to 150 ms. We used two configurations for Wi-Fi: with power-save mode (PSM) enabled, and without. With PSM, the smartphone's Wi-Fi radio wakes up only when it has data to transmit and once every 100 ms when it checks whether the access point has any incoming data.

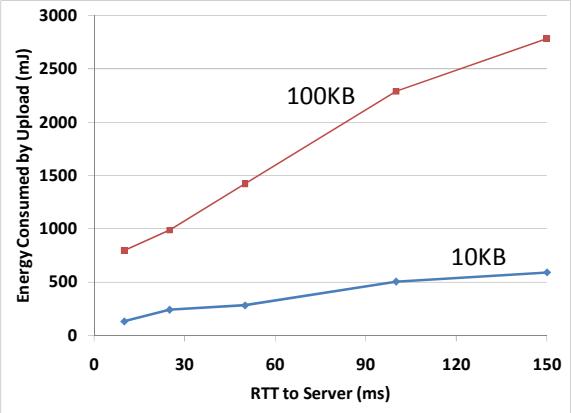

Figure 2: The Energy Consumed by Offloading 10 KB and 100 KB of Code to the Cloud for Various Round-Trip Times to the Server. ***We used power-save mode (PSM) for the results presented in this graph; we also conducted experiment with no PSM that show the same trends. High RTTs lead to significant increases in the energy consumption.***

Figure 2 shows the energy consumed when offloading 10 KB and 100 KB of data with Wi-Fi using PSM. We also measured the results when offloading pieces of code with different sizes (20 KB, 50 KB, 500 KB, and 1 MB) and using Wi-Fi with PSM disabled, but we omit showing these results to avoid cluttering the graph since the results show the same trends. As Figure 2 shows, the energy consumption of code offload grows almost linearly with the RTT. In fact, for offloading 10 KB of code, the energy consumption almost doubles when increasing the RTT from 10 ms to 25 ms only, and it doubles again when the RTT reaches 100 ms.

Two important consequences arise from these results. First, cloud providers should strive to minimize the latency to the cloud for mobile users. Shorter RTTs can lead to significant energy savings. Second, the benefits of remote execution are most impressive when the remote server is *nearby* (RTT of 10 ms), such as on the same LAN as the Wi-Fi access point, rather than in the cloud (RTT $\geq$ 25 ms). There are many scenarios where it is straightforward to deploy remote execution servers near smartphone users, such as in an enterprise to help increase the battery lifetimes for employees' smartphones, and in private homes to help home smartphone users. Others also pointed out the huge energy benefits of offloading code to nearby servers [37].

3. MAUI SYSTEM ARCHITECTURE

MAUI's goal is to maximize the benefits of code offload for today's smartphone devices. In this section, we present a high-level overview of MAUI's components on a mobile device and in the infrastructure in order to understand how they all integrate into one platform for developing mobile applications.

MAUI provides a programming environment where developers annotate which methods of an application can be offloaded for remote execution. Each time a method is invoked and a remote server is available, MAUI uses its optimization framework to de-

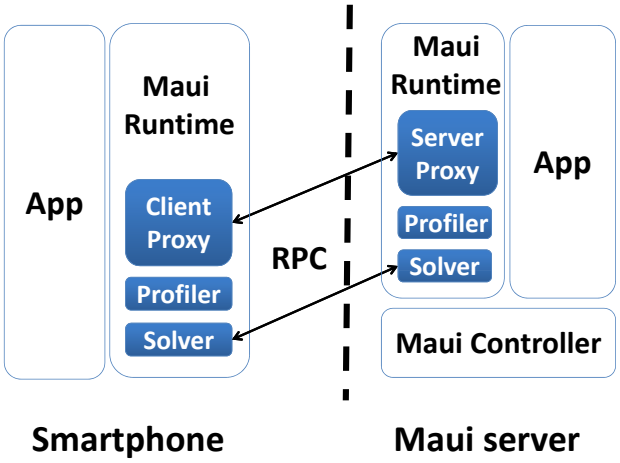

Figure 3: High-level view of MAUI's architecture.

cide whether the method should be offloaded. Once an offloaded method terminates, MAUI gathers profiling information that is used to better predict whether future invocations should be offloaded. If a disconnect occurs, MAUI resumes running the method on the local smartphone; in this case, the application's energy consumption only incurs a small penalty – the cost of shipping the control and program state to the server.

MAUI instruments each method of an application to determine the cost of offloading it, such as the amount of state that needs to be transferred for remote execution, and the benefit of offloading it, such as the number of CPU cycles saved due to the offload. Additionally, MAUI continuously measures the network connectivity to the infrastructure, estimating its bandwidth and latency. All these variables are used to formulate an optimization problem whose solution dictates which methods should be offloaded to the infrastructure and which should continue to execute locally on the smartphone. Because mobile users may move in and out of a MAUI server's range relatively rapidly, the optimization problem is re-solved periodically to enable MAUI to adapt to changes in the networking environment.

Figure 3 provides a high-level overview of the MAUI architecture. On the smartphone, the MAUI runtime consists of three components: 1) an interface to the decision engine (to save energy, the solver actually runs on the MAUI server); 2) a proxy, which handles control and data transfer for offloaded methods; and 3) a profiler, which instruments the program and collects measurements of the program's energy and data transfer requirements. On the server side, MAUI provides four components: the profiler and the server-side proxy which perform similar roles to their client-side counterparts; the decision engine which periodically solves the linear program; and the MAUI coordinator, which handles authentication and resource allocation for incoming requests to instantiate a partitioned application. In Sections 4, 5, and 6, we describe in detail MAUI's mechanisms for program partitioning, the MAUI profiler, and the MAUI solver, respectively.

4. PROGRAM PARTITIONING

MAUI enables developers to produce an initial partitioning of their applications with minimal effort. The developer simply annotates as *remoteable* those methods and/or classes that the MAUI runtime should *consider* offloading to a MAUI server. Certain types of code should not be marked remoteable: 1) code that implements the application's user interface; 2) code that interacts with I/O devices where such interaction only makes sense on the mobile device; and 3) code that interacts with any external component that would be affected by re-execution (MAUI's failure handling is discussed in detail in Section 4.2). Examples of the second category include code that reads an accelerometer, or code that determines the location of smartphone by reading from the GPS device. An example of the third category is code that uses a network connection to perform an e-commerce transaction (such as purchasing an item from an online store).

Our intent for the remoteable annotations is that developers do not need to guess about whether or not a particular method makes sense to offload (from the perspective of energy and performance). Instead, they should mark as remoteable all code that meets the criteria specific above, and it is up to the MAUI solver (described in Section 6) to determine at runtime whether it makes sense from a performance and energy standpoint to execute it remotely.

We considered and then discarded an alternative approach: require the programmer to annotate those methods and/or classes that can only run on the smartphone as *local*, and then consider all the remaining code as remoteable. Although this approach may lead to fewer annotations, we decided against this approach because it favors performance over program correctness. If the programmer makes a mistake, such as forgetting to label a method as *local*, then the program may no longer behave correctly should MAUI decide to offload that method. Instead, with our approach such mistakes only affect the program's performance and not its correctness.

4.1 Partitioning .NET Applications

To support partitioning an application across multiple machines, MAUI needs to address the following challenges:

1. Because today's smartphones typically use a different instruction set architecture (ARM) than desktop and server machines (x86), MAUI needs the ability to execute the same program on different CPU architectures, preferably without access to the program source code.

2. MAUI must be able to automatically identify which methods are marked *remoteable* and which are not.

3. MAUI must be able to automatically identify and migrate the necessary program state from a running program on one machine to another.

4. MAUI must be able to dynamically select whether to run a method locally or remotely based on the current environment.

5. MAUI must be able to detect and tolerate failures without affecting the original program semantics.

In the rest of this section we discuss how our MAUI prototype handles these challenges.

4.1.1 *Executing the Same Code on Different CPU Architectures*

MAUI is currently designed only to support applications written for the Microsoft .NET Common Language Runtime (CLR) [36]. While the CLR supports a variety of programming languages, our MAUI applications are all written in C#[1]. All CLR applications are compiled to the CIL intermediate language, regardless of what source language they are written in. An "executable" for the CLR contains CIL instructions, and the CIL is dynamically compiled at execution time. Thus, by leveraging the CLR's capabilities, MAUI obtains independence from differences in the instruction set architecture between today's smartphones (which typically have ARM-

[1]Some include native dll's with C# wrappers.

based CPUs) and today's desktop and server machines (which typically have x86 CPUs).

To begin running an application, the MAUI runtime must ensure that the MAUI server has copies of the application executables. The MAUI runtime supports two options: 1) the MAUI server can obtain copies of the program executables directly from the smartphone (which incurs a delay and consumes energy), or 2) the MAUI runtime on the smartphone can send signatures of the executables to the MAUI server which then downloads the actual executables from a cloud service.

4.1.2 Extracting Remoteable Methods Using Reflection

To identify remoteable methods in a language-independent manner, we use the custom attribute feature of the .NET CLR. Attributes are meta-data that annotate specific code elements such as methods or classes, and they are included in the compiled .NET CLR executables. The application developer modifies the application's source code by adding the "[Remoteable]" attribute to each method that is safe to execute remotely. The MAUI runtime uses the .NET Reflection API [22, 36] to automatically identify which methods the developer has marked as suitable for remote execution, simply by searching through the executable for those methods tagged with the "[Remoteable]" attribute.

4.1.3 Identifying the State Needed for Remote Execution Using Type-Safety and Reflection

At compile time, MAUI generates a wrapper for each method marked as remoteable. This wrapper follows the original method's type signature, but with two changes: it adds one additional input parameter, and one additional return value. Figure 4 shows a snippet of the video game application's original interface and the MAUI wrapper interface. The additional input parameter is used to transfer the needed application state from the smartphone to the MAUI server, and the additional return value is needed to transfer the application's state back from the server to the smartphone.

Performing application state transfer leverages the type-safe nature of the .NET runtime. Type safety allows us to traverse the in-memory data structures used by the program, and to only send over the network data which is potentially referenced by the method being offloaded. To determine which state needs to be serialized beyond the explicit method parameters, we currently take the conservative approach of serializing all the current object's member variables, including not only simple types, but also nested complex object types. We also serialize the state of any static classes and any public static member variables. To perform the serialization, we use the .NET built-in support for XML-based serialization, which makes use of the .NET Reflection API. We are currently working on replacing this with our own custom binary serializer that will reduce the serialization overhead. Because the size of XML-based serialized state is potentially large compared to the in-memory representation, we also optimize the overhead of state transfer by only shipping incremental deltas of the application state rather than the entire state. We are also currently developing a static analysis tool that determines which variables are actually referenced in the remoteable method, to further limit the amount of state transferred.

4.1.4 Performing Code Offload

At compile time, MAUI generates two proxies, one that runs on the smartphone and one that runs on the MAUI server. The role of these proxies is to implement the decisions made by the MAUI solver (described in Section 6). The solver decides, based on input from the MAUI profiler, whether the method in question should

```
//original interface
public interface IEnemy {
  [Remoteable] bool SelectEnemy(int x, int y);
  [Remoteable] void ShowHistory();
  void UpdateGUI();
}

//remote service interface
public interface IEnemyService {
  MAUIMessage<AppState, bool> SelectEnemy (AppState state, int x, int y);
  MAUIMessage<AppState, MauiVoid> ShowHistory(AppState state);
}
```

Figure 4: Local and remote interfaces.

be executed locally or remotely, and the proxies handle both control and data transfer based on that decision. The granularity of state and control transfer in MAUI is at the method level; we do not support executing only portions of a method remotely. For calls which transfer control from the local smartphone to the remote server, the local proxy performs state serialization before the call and then deserialization of the returned application state after the call. When a remoteable method which is currently executing on the MAUI server invokes a method which is not remoteable, the server-side proxy performs the necessary serialization and transfers control back to the smartphone. The MAUI runtime currently has only limited support for multi-threaded applications – we only offload methods from a single thread at any given point in time.

4.2 Handling Failures

MAUI detects failures using a simple timeout mechanism: when the smartphone loses contact with the server while that server is executing a remote method, MAUI returns control back to the local proxy. At this point, the proxy can either re-invoke the method locally, or it can attempt to find an alternate MAUI server and then re-invoke the method on the new MAUI server. Because program state is only transferred at the start and end of methods, re-executing a portion of the method will not affect the program's correctness. However, programmers should not label methods as remoteable if those methods interact with any external component that would be affected by re-execution (e.g., using a network connection to perform an e-commerce transaction).

4.3 Additional Program Modifications Can Bring Performance Benefits

One of the key benefits of MAUI's approach to program partitioning is the extremely low barrier to entry for developers. Once the program is working with MAUI, the programmer may be interested in further optimizing the program's performance and/or energy consumption. From our experience, we find that some restructuring of the video game application has a significant impact on its performance and energy characteristics. The information produced by the MAUI profiler as input to the MAUI solver can also benefit the application developer in terms of understanding the energy and state transfer characteristics of the program.

In the video game application, the program logic that is best offloaded determines the attack strategy for the enemies. In the unmodified application, the method that implements this logic is called once per enemy during each frame refresh. If there are 60 enemies, this leads to 60 remote calls per frame. We modified the game to perform all the enemy update operations in a single call, which leads to a significant improvement in latency (i.e., the frame rate), and a reduction in energy consumption. We found that our

Figure 5: The hardware power meter used for energy measurements.

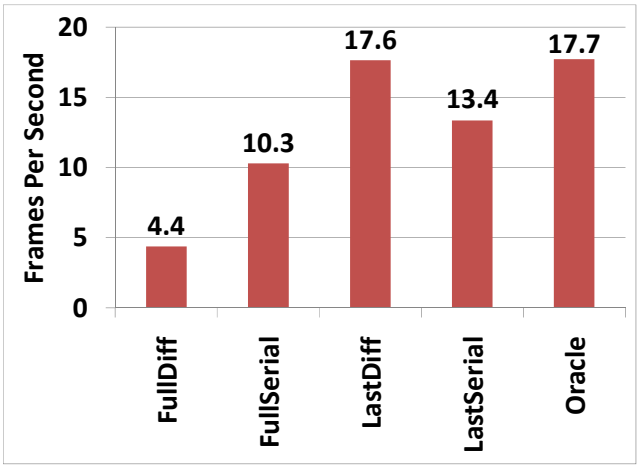

Figure 6: Performance overhead of MAUI profiling. ***Effect on the game's performance of five different strategies for the MAUI profiler.***

legacy face recognition application and the chess game did not require any code restructuring to enable partitioning, and we wrote the voice translator application from scratch and therefore we deliberately separated out the logic that implements the graphical UI and I/O on the smartphone (e.g., the translator uses the microphone and the speaker) from the bulk of the application that implements the translation functionality.

5. MAUI PROFILER

At runtime, before each method is invoked, MAUI determines whether the method invocation should run locally or remotely. Offload decisions depend on three factors: 1) the smartphone device's energy consumption characteristics; 2) the program characteristics, such as the running time and resource needs of individual methods; and 3) the network characteristics of the wireless environment, such as the bandwidth, latency, and packet loss. The MAUI profiler measures the device characteristics at initialization time, and it *continuously* monitors the program and network characteristics because these can often change and a stale measurement may force MAUI to make the wrong decision on whether a method should be offloaded. The current implementation of the MAUI profiler does not explicitly incorporate the CPU load on the MAUI server. In the rest of the section, we provide an in-depth description of MAUI's techniques for device, program, and networking profiling.

5.1 Device Profiling

Today's smartphones do not offer a way to obtain fine-grained energy measurements of an application. Instead, they just offer a simple "how much battery is left" API that is very coarse-grained and often unreliable. Instead, to measure a device's energy consumption, we attach a hardware power meter [25] to the smartphone's battery (see Figure 5). This power meter can provide fine-grained energy measurements; it samples the current drawn from the battery with a frequency of 5000 Hz.

We use the power meter to build a simple energy profile of the smartphone. For the CPU, our approach is inspired by JouleMeter [19]. We constructed a synthetic benchmark that uses both integer and floating point arithmetic. We ran our synthetic benchmark multiple times, varying the run-time of the benchmark and instrumenting it to record the number of CPU cycles it required for each run. We then used the collected samples of CPU utilization and the corresponding smartphone energy consumption to build a simple linear model, using least-squares linear regression. This model lets us predict the energy consumption of a method as a function of the number of CPU cycles it requires to execute. We validated our model by comparing its prediction of energy consumption with the actual measurements produced by hardware power monitor for the applications we have built using MAUI. We found that the median error produced by the model is less than 6%, while the mean error is less than 8% and the standard deviation is 4.6%. Finally, we also used the energy meter to characterize the energy consumption of using the smartphone's Wi-Fi and 3G wireless radios.

5.2 Program Profiling

The MAUI profiler instruments each method to measure its state transfer requirements, its runtime duration, and the number of CPU cycles required for its execution. The amount of state needed to be sent to the MAUI server to execute the method includes the size of all data potentially referenced by the method, as well as the amount of state required to be returned to the smartphone once the method is completed (Section 4.1.3 describes in detail how MAUI identifies which program state needs to be transferred).

MAUI uses the method's duration and CPU cycles to estimate the energy consumed by running the method on the smartphone. Characterizing an application's energy consumption is very challenging for two reasons. First, applications are not deterministic: each subsequent invocation of a method can take a different code path leading to a different running duration and energy profile than the previous invocation. While more sophisticated program instrumentation can help mitigate this problem, such techniques may be prohibitively expensive. Instead, MAUI uses past invocations of a method as a predictor of future invocations; we found this assumption to work well for the phone applications we used to experiment with MAUI. Second, today's smartphones scale the CPU's voltage dynamically to save energy (i.e., dynamic voltage scaling) without informing the software running on the phone. While the dynamic voltage scaling mechanisms could drastically change the energy profile of an application, our experiments validating the CPU energy model discussed earlier did not show this to be a significant problem. Once the application terminates and the user does

not interact with the phone, the smartphone often lowers the CPU's voltage to save energy.

5.2.1 Profiling Overhead

As mentioned in Section 4.1.3, the MAUI proxies ship incremental deltas rather than the full application state to reduce the network and energy overhead of state transfer. To implement this, the local MAUI proxy keeps track of what state it has previously sent to the remote proxy, and just before each remote invocation, the proxy calculates the delta between the local state and the remote state. If the size of the delta is smaller than the full state, it sends the delta instead. This optimization, which seems straightforward, turns out to introduce unintended complexity for the MAUI runtime.

The MAUI profiler observes the program behavior over time as it executes, and the MAUI solver uses past program behavior as a predictor of how the application will behave in the future. The current MAUI profiler does not persist the profiling data across multiple runs of the same program. When a method A calls method B, the MAUI profiler measures the size of state that would need to be transferred over the network to enable B to run remotely, and this measurement is performed regardless of whether B is actually remoted. The performance overhead of taking this measurement is simply the cost of using the XML serializer to transform the state into XML format, and then measuring the size of the buffer. The more times the profiler observes A calling B, the better the estimate it obtains of future behavior.

The introduction of deltas changes this picture. When A calls B, the size of the state that needs to be transferred is now not only a function of the program's previous behavior, but also a function of the MAUI runtime behavior. In other words, MAUI has perturbed the very thing that it is measuring about the program. If MAUI has offloaded a method recently the delta will likely be small, whereas if MAUI has not offloaded in a long time then the delta is likely to be substantially larger.

Another unintended consequence of calculating deltas is the performance impact it has on interactive applications. To characterize this impact, we performed the following experiment. We used five different strategies for profiling an interactive application (a video game) and we measured the video game's performance (in frames per second) when offloading code to a nearby server. These strategies are:

1. *FullDiff* – the profiler uses delta calculation and serialization on every call to a remoteable method.

2. *FullSerial* – the profiler uses serialization only on each call to a remoteable method.

3. *LastDiff* – the profiler uses delta calculation and serialization on the remoteable method's *first* call, but then re-uses this estimate without recomputing it on subsequent calls to the method. However, whenever MAUI chooses to offload the method, MAUI updates these estimates.

4. *LastSerial* – the profiler uses serialization on the remoteable method's first call similar to LastDiff.

5. *Oracle* – the profiler knows in advance exactly how much state is transfered for each remoteable method without any computation.

Figure 6 shows the effect of MAUI's profiling on the performance of the video game under each of these five scenarios. The additional overhead of calculating deltas reduces the game's frame rate from 10 fps to 4 fps, and the two heuristics, *LastDiff* and *LastSerial*, both provide a significant benefit in terms of interactive performance of the game. *LastDiff* provides more benefit than *LastSerial* because it enables the decision engine to offload certain methods that are not offloaded with *LastSerial*. These results show that our simple heuristic can balance the trade-off between the cost of performing profiling frequently and the freshness of its estimates of application's behavior.

5.3 Network Profiling

Initially, we believed that profiling the wireless networking environment would be relatively straightforward. We planned to use Wi-Fi's power-save mode (PSM) to maximize the energy savings when offloading code, and we planned to use network measurement tools to estimate the wireless link's round-trip time, bandwidth [35], and packet loss [38]. Instead, we discovered that both design choices were either wrong or unnecessarily complicated.

First, we discovered that Wi-Fi's power-save mode (PSM) can hurt the energy consumption of code offloading. With PSM, the smartphone's Wi-Fi radio is put to sleep, waking up just before the access point (AP) sends a beacon (every 100ms) to check if the AP has any incoming data destined for the phone. If there is no data sent by the MAUI server available at the AP, then the radio goes back to sleep for another 100ms. Because MAUI uses TCP, PSM's interactions with TCP have some surprising effects in terms of time to complete a transfer, and in terms of energy consumption.

We characterized these interactions by performing a suite of experiments transferring different amounts of data to and from a MAUI server varying the connection's RTT, both with PSM enabled and with it disabled. To illustrate our findings we select two specific examples of a 500 KB TCP transfer over a link with a 25 ms RTT and one with 75 ms RTT with PSM enabled.

Figure 7 shows the dynamics of the TCP transfer by plotting the TCP's byte sequence offsets over time. With an RTT of 25 ms, TCP experiences the PSM effects during slow-start, when the TCP sender has a small window size. The TCP sender finishes sending all the segments and then goes to sleep. It wakes up 100 ms later, receives the TCP ACKs and again sends a window worth of segments. However, after about one second, the window has become large enough that the TCP ACKs arrive back at the sender before it has run out of segments to send. This prevents the sender from going into sleep mode; at this point, the TCP transfer behaves similar to one without PSM enabled. Effectively, the transfer experiences an RTT of 100 ms (due to PSM) during the first second and an RTT of 25 ms during the last second. A similar effect occurs when the RTT is 75 ms. However, because the RTT is much closer to 100 ms (the PSM sleep interval), the discrepancy between the first part of the transfer and the second is not so large.

This PSM behavior raises an interesting trade-off. On one hand, with PSM, the sender saves energy by sleeping whenever it has no packets ready to send. On the other hand, PSM increases the duration of a transfer. The device consumes more energy overall when the transfer lasts longer. This trade-off has important consequences for the use of PSM with a system like MAUI. For example, for short RTTs (lower than 75 ms in our experiments), disabling PSM *saves energy* because the transfers are much shorter, and the device consumes less energy overall during the transfer duration. However, when RTTs approach 100 ms, enabling PSM saves energy because the wireless sender is sleeping while waiting for ACKs from the receiver. The same pattern occurs with RTTs higher than 100 ms – during slow start, the RTTs are effectively rounded up to 200 ms.

Unfortunately, we had to leave PSM disabled in our current implementation of MAUI, because Windows Mobile does not expose an API that allows regular applications to enable or disable PSM, we can only turn it on and off through the systems settings.

Second, we found that it was unnecessary to use specialized tools to individually measure the wireless link's round-trip time, bandwidth, and packet loss. Instead, our experiments showed that the following very simple technique works well: send 10 KB of data

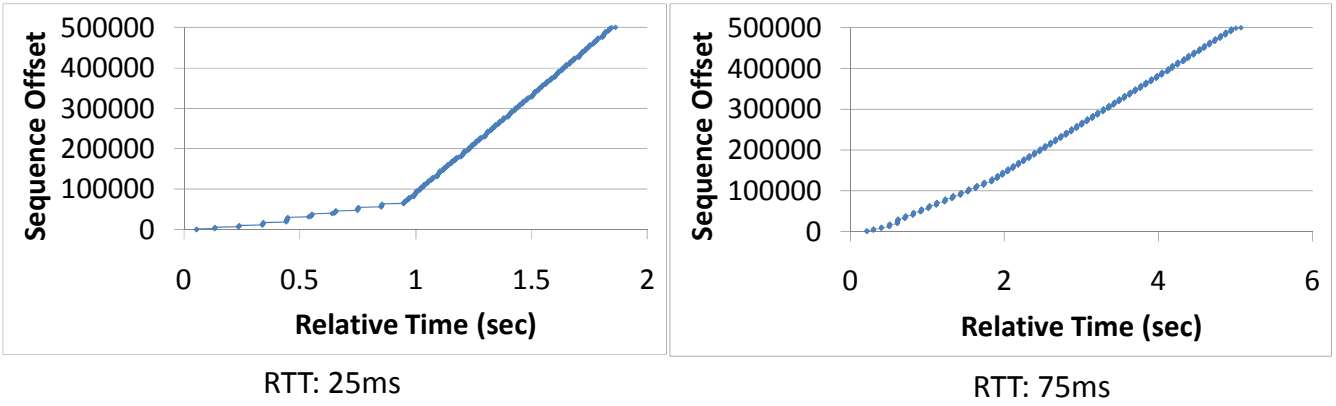

Figure 7: The dynamics of a 500 KB TCP transfer with PSM disabled over a two links, one with a 25 ms RTT and one with 75 ms RTT. ***During slow-start, for small sender window sizes, the sender goes to sleep after finishing sending its window causing the RTT of the transfer to effectively become 100 ms. As the size gets larger, the senders starts receiving TCP ACKs before finishing sending its window worth of segments. The discrepancy between these two regions of a transfer is less pronounced as the link's RTT increases.***

over TCP to the MAUI server and measure the transfer duration to obtain an average throughput. We use 10 KB because this is representative of typical transfers made to a MAUI server. This simple approach allows MAUI's network profiler to take into account both the latency and bandwidth characteristics of the network. Each time MAUI offloads a method, the profiler uses this opportunity to obtain a more recent estimate of the network characteristics; these recent estimates are then averaged (we use a sliding window) with the current estimate. Finally, if no transfers are performed for one minute, MAUI conducts another measurement by sending 10 KB to the server to obtain a fresh estimate.

6. MAUI SOLVER

The MAUI solver uses data collected by the MAUI profiler as input to a global optimization problem that determines which remoteable methods should execute locally and which should execute remotely. The solver's goal is to find a program partitioning strategy that minimizes the smartphone's energy consumption, subject to latency constraints.

Deciding where to execute each method is challenging because it requires a global view of the program's behavior. Figure 8 shows a simplified version of the call graph for a face recognition application. Each vertex represents a method and its computational and energy costs, and each edge represents the size of the method's state and the energy consumed to transfer this state remotely. For each individual method, its remote execution is more expensive than its local execution, and yet remote execution can save energy if *FindMatch()*, *InitializeFaceRecognizer()*, and *DetectAndExtractFaces()* are *all* remoted. Thus, the MAUI solver's decisions must be globally optimal (i.e., across the entire program) rather than locally optimal (i.e., relative to a single method invocation).

We now describe our formulation of the global optimization problem. At a high level, the execution behavior of the program is modeled as an annotated call graph, and both the graph and all the annotations are provided by the MAUI profiler as inputs to the solver (Section 5 describes how these input parameters are estimated). We use a linear program solver to find the optimal partitioning strategy that minimizes the energy consumed by the smartphone subject to a set of latency constraints. To ease the burden on the programmer, we provide a default latency constraint: the total execution latency L must not exceed 5% more than the latency incurred if all the methods in the program are executed locally. There are some applications where absolute latency constraints are important to maintain the application's quality of service. To handle such cases, we optionally allow the application developer to specify additional latency constraints. As the application runs, we re-run the solver periodically for two reasons: to adapt to changing environmental conditions, and also to learn from the historical behavior of the program. The MAUI solver is invoked asynchronously from the mobile device to avoid affecting the interactive performance of the application.

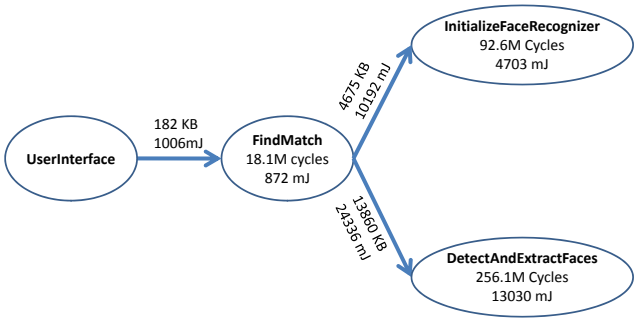

Figure 8: The call graph of a face recognition application. ***Each vertex represents a method and its computational and energy costs, and each edge represents the size of the method's state and the energy consumed to transfer this state remotely.***

In more detail, we start with the application's call graph $G = (V, E)$. The call graph represents the call stack as the program executes. Each vertex $v \in V$ represents a method in the call stack, and each edge $e = (u, v)$ represents an invocation of method v from method u. We annotate each vertex $v \in V$ with the energy it takes to execute the method locally E_v^l, the time it takes to execute the method locally, T_v^l, and the time it takes to execute the method remotely, T_v^r. We annotate each edge $e = (u, v)$ with the time it takes to transfer the necessary program state $B_{u,v}$ when u calls v, and the energy cost of transferring that state $C_{u,v}$. Each vertex v is also annotated with the parameter r_v that indicates if the method is marked remoteable. If a method is called from within a loop,

all these costs are scaled up by a factor that corresponds to the profiler's estimate of the number of loop iterations.

Formally, MAUI solves the following 0-1 integer linear programming (ILP) problem shown below. The solver solves for variable I_v. I_v is the indicator variable: $I_v = 0$ if method v is executed locally and is 1 if executed remotely.

$$\text{maximize} \sum_{v \in V} I_v \times E_v^l - \sum_{(u,v) \in E} |I_u - I_v| \times C_{u,v}$$

$$\text{such that:} \sum_{v \in V} ((1 - I_v) \times T_v^l) + (I_v \times T_v^r))$$

$$+ \sum_{(u,v) \in E} (|I_u - I_v| \times B_{u,v}) \leq L$$

$$\text{and} \qquad I_v \leq r_v, \, \forall v \in V$$

The first term in the objective function is the total energy saved by executing methods remotely – the savings are essentially the energy cost if the method had been executed locally. The second term in the objective function captures the energy cost of data transfer to execute a method remotely. Note that data transfer incurs an energy cost only if the two methods u and v are not both executed in the same location. The first constraint stipulates that the total time to execute the program be within L. The second constraint stipulates that only methods marked remoteable can be executed remotely.

7. EVALUATION

In this section, we evaluate MAUI's ability to improve the energy consumption and performance of smartphone applications, using a combination of macrobenchmarks (in Section 7.2) and microbenchmarks (in Section 7.3).

7.1 Methodology

We used an HTC Fuze smartphone running Windows Mobile 6.5 with the .NET Compact Framework v3.5. For the MAUI server, we used a regular dual-core desktop with a 3 GHZ CPU and 4 GB of RAM running Windows 7 with the .NET Framework v3.5. Note that the .NET Compact Framework running on the smartphone is more restrictive than the one running on the MAUI server. The server is equipped with an NDIS intermediate driver that inserts packet queuing delays to control the RTT of the path between the smartphone and the server (without adding any delay, the path between the smartphone and the server has an RTT of 10 ms in our setup). We measure energy on the phone using a hardware power meter [25] attached to the smartphone's battery. This power meter samples the current being drawn from the battery at 5000 Hz.

We evaluate MAUI's benefits on four applications. Three of these applications were already pre-built and running on Windows Mobile phones: a face-recognition application, a highly-interactive video game, and a chess game. Although these applications are relatively simple, they each contain on the order of 10 remoteable methods. We developed the fourth application from scratch – a real-time voice-based language translator (Spanish to English). The language translator must use remote execution to run on the smartphone because its memory resource requirements exceed those offered locally by the smartphone.

7.2 Macrobenchmarks

We now look at three macrobenchmarks that characterize MAUI's ability to reduce energy, improve performance, and bypass the resource limitations of smartphones.

1. MAUI's primary goal is to reduce the energy consumption of mobile applications. *How much does MAUI reduce energy consumption of mobile applications?*

2. In addition to saving energy, MAUI can also improve the performance of mobile applications. This is especially important for interactive applications, such as games. *How much does MAUI improve the performance of mobile applications?*

3. MAUI also allows developers to build applications that cannot currently be supported on mobile devices because of their resource requirements. *Can MAUI run resource-intensive applications?*

7.2.1 How Much Energy Does MAUI Save For Mobile Applications?

Figure 9 presents a comparison of the energy consumption of three applications (the face-recognition application, the video game, and the chess game) when executing in six different scenarios. In the first scenario, the applications are running standalone on the smartphone. In the next four scenarios, we use MAUI to offload code to a server over a link with different RTT values (10 ms, 25 ms, 50 ms, and 100 ms). In the final scenario, MAUI offloads code over a 3G interface. In the case of the video game and chess, code offload over 3G consumes more energy than local execution, and thus MAUI's optimizer refuses to perform the offload. However, we modified MAUI to force the code offload when running over 3G to measure its energy consumption. We refer to this modification as MAUI* in the Figure. Note that we did not use the voice translation application because it cannot run standalone on the smartphone.

As seen in Figure 9, the face recognition application can achieve drastic energy savings when using MAUI. When the server is nearby, the energy savings reach one of order of magnitude; as the latency to the server is increasing, MAUI saves less energy. In fact, the energy consumed when offloading code over 3G is 2.5 times higher than offloading code to a nearby server. The energy savings for both video and chess are less drastic but they remain significant; when offloading to a nearby server, MAUI saves 27% energy for the video game and 45% for chess.

7.2.2 How Much Does MAUI Improve the Performance of Mobile Applications?

We ran the applications in the same six scenarios but instead measured performance rather than energy consumption and we present the results in Figure 10. The performance results are also impressive; offloading the code to a nearby server reduces the latency of performing face recognition by one order of magnitude from 19 seconds down to less than 2 seconds. At these performance levels, face recognition becomes an interactive application.

For the video game, MAUI offloads one method (called "MoveComplex"), whereas for chess, MAUI offloads two methods: "Select Piece" and "Move Piece". The results shown are averaged over 20 executions. For the video game, MAUI also provides substantial performance improvements: the nearby MAUI server reduces latency by a factor of 4.8; this latency reduction effectively double the game's refresh rate from 6 frames per second to 13. For chess, offloading using MAUI when the RTT is higher than 25 ms can hurt performance; for example, the "SelectPiece" method takes longer to execute when the RTT is 50 ms. The worst case for performance is offload over 3G: the 'Select Piece" method for chess incurs an 77% performance overhead and the "Fire Bullet" method for the video game incurs a 54% overhead.

7.2.3 Can MAUI Run Resource-Intensive Applications?

Figure 11 shows the memory consumption and CPU utilization over time of our translator application when running on a PC with a 2 GHz Intel Core2 CPU and 2 GB of RAM. The peak memory

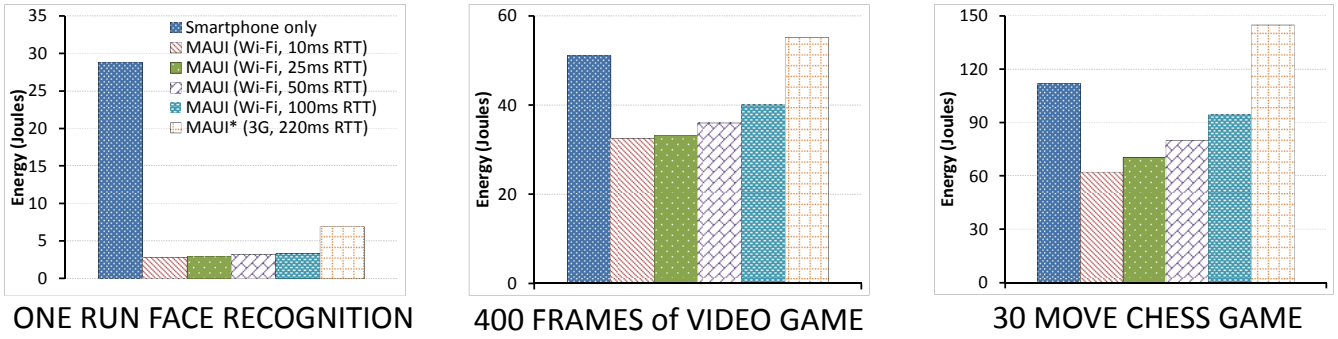

Figure 9: A comparison of MAUI's energy consumption. ***We compare the energy consumption of running three applications standalone on the smartphone versus using MAUI for remote execution to servers that are successively further away (the RTT is listed for each case). The graph on the left shows one run of the face recognition application; the graph in the middle shows running the video game for 400 frames; the graph on the right shows running the chess game for 30 moves. MAUI* is a slight modification to MAUI to bypass the optimizer and to always offload code. Without this modification, MAUI would have not performed code offload in the case of the video game and chess because offload ends up hurting energy performance.***

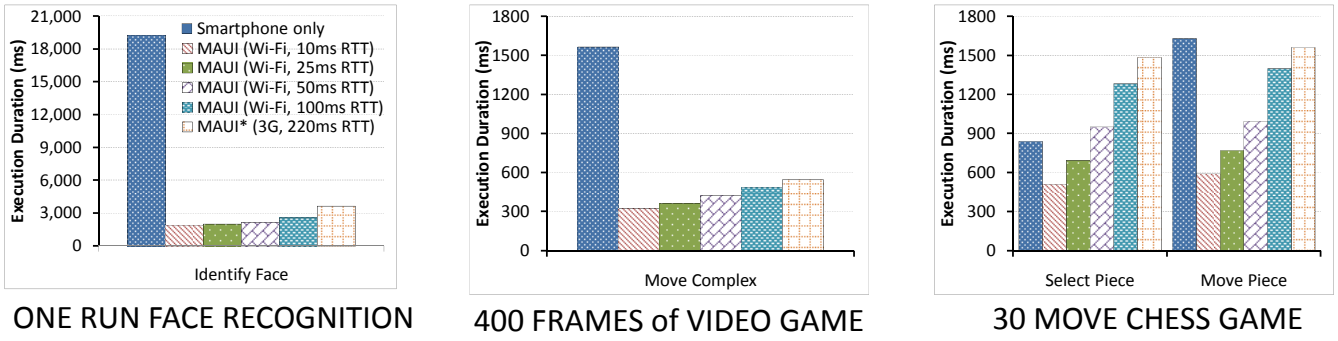

Figure 10: A comparison of MAUI's performance benefits. ***We compare the performance of running three application standalone on the smartphone versus using MAUI for remote execution to servers that are successively further away (the RTT is listed for each case). The graph on the left shows one run of the face recognition application; the graph in the middle shows running the video game for 400 frames; the graph on the right shows running the chess game for 30 moves.***

consumption of the speech recognizer is approximately 110 MB, which is higher than the maximum memory limit for a single process on a typical smartphone (32 MB for our HTC Fuze or 48 MB for an iPhone). MAUI enabled us to bypass the smartphone's limits on an application's memory footprint.

7.3 Microbenchmarks

We now look at four microbenchmarks that characterize the behavior of both the MAUI solver and the MAUI proxy.

1. MAUI uses an optimization framework to decide what code to offload to a remote server. Upon invoking a function whose code can be offloaded, MAUI solves a 0-1 integer linear problem to decide where the function should execute. *What is the performance overhead of MAUI's optimizer?*

2. Deciding where to execute each method of a program is challenging because it requires a global view of the program's behavior. As Section 6 describes (and Figure 8 shows), the decision of whether to offload a method depends on more than just its own computational cost and its state. *Is the MAUI solver effective at identifying offload opportunities based on a global view of the program?*

3. The MAUI proxy keeps track of what state it has already sent to the remote server, and it ships incremental deltas to the remote server rather than the entire state for those cases where the deltas are smaller. *How effective are incremental deltas at reducing MAUI's data transfer overhead?*

4. The MAUI solver receives new information from the profiler on a periodic basis as the program executes. *Does the solver adapt to changing network conditions and to changes in the computational costs of individual methods?*

7.3.1 What is the Overhead of MAUI's Solver?

To measure the overhead of MAUI's optimizer, we instrumented the optimizer to record how long it takes to solve each instance of the integer linear programming (ILP) problem. We found that the solver takes 18 ms on average to solve the call graph of the chess application and 46 ms for the video game. Our results show that the ILP problems arising from MAUI's program partitioning needs can be solved quickly.

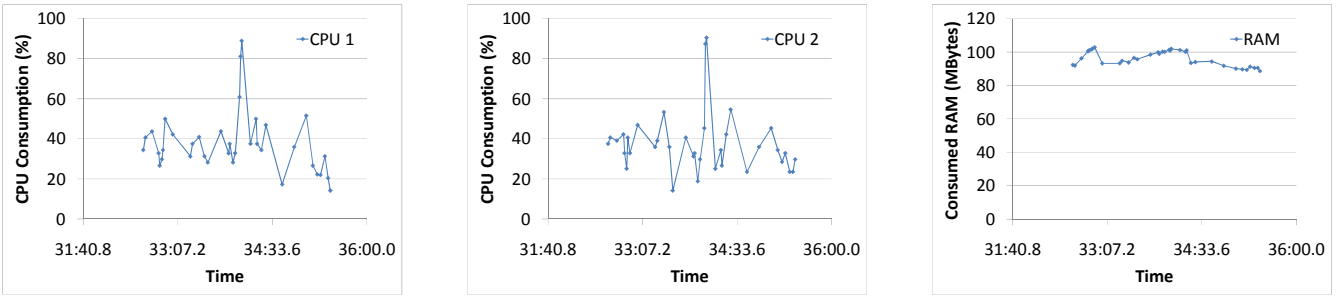

Figure 11: Profiling the Speech Recognition Engine on the MAUI Server. ***The MAUI server is a 2 GHz Intel Core2 CPU with 2 GB of RAM. The load of one CPU is shown in the graph on the left, whereas the other CPU is shown in the middle. The graph on the right shows the RAM consumed by the speech engine.***

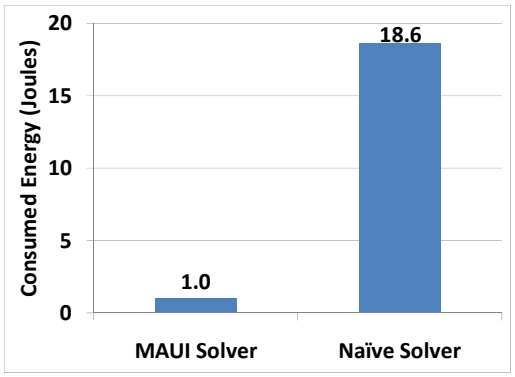

Figure 12: The energy consumed by the face recognition application for different code offload solvers. ***We ran the application using both the MAUI solver and a naïve solver takes a local view of the program.***

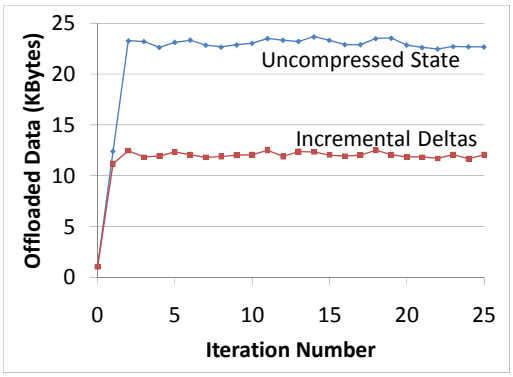

Figure 13: The Performance of the Incremental Deltas to and from the Remote Server. ***MAUI uses incremental deltas when offloading code to a remote server. In the case of the video game, this optimization reduces the amount of state transferred from 23 KB to 12 KB, on average.***

7.3.2 Does MAUI Require a Global View of the Program to Identify Offload Opportunities?

To answer this question, we constructed a much simpler solver as an alternative to MAUI. This naïve solver considers each method separately; a method is offloaded only if its remote execution (i.e., the cost of transferring its state) consumes less energy than its local execution. We used both the MAUI solver and the naïve solver to solve the call graph shown in Figure 8, which is a simplified version of the call-graph for the face recognition application. Figure 12 shows that the MAUI solver consumed two orders of magnitude less energy than the naïve solver. By taking a local view of each method, the naïve solver decided to execute all methods locally, whereas MAUI chose to offload parts of the application.

7.3.3 How Effective are Incremental Deltas at Reducing MAUI's Data Transfer Overhead?

We instrumented MAUI to measure the amount of data exchanged with a remote server when offloading code both with and without using the incremental deltas optimization. Figure 13 shows the benefits of transferring deltas rather than the entire uncompressed state, for the video game for each subsequent execution of the offloaded method. After the first two iterations, when the method is still being initialized, MAUI benefits from using incremental deltas: this optimization reduces the amount of state transferred by a factor of two from 23 KB to 12 KB.

7.3.4 Does the MAUI Solver Adapt to Changing Network Conditions and CPU Costs?

To answer this question, we examine the decisions that the MAUI solver makes for the modified version of the video game[2]. As we mentioned in Section 4.3, we restructured the video game to improve its offload behavior. At the same time, we extended the game's functionality to include physics modeling, using an off-the-shelf physics engine [33]. In particular, we modified the missiles fired by the enemies to act as homing missiles, which means that we increase the chance that the missiles will hit the player. Because the physics calculations are CPU intensive, when the number of active missiles becomes larger, the CPU cost of *HandleMissles()* grows significantly.

The modified game's structure is as follows: there is a top-level method called *DoLevel()* which performs the vast majority of the video game's computation that can be offloaded, because it does not draw directly to the screen. *DoLevel()* in turn invokes *HandleEnemies(), HandleMissiles(),* and *HandleBonuses()*. The size of the

[2]All the results we presented earlier were for the unmodified version of the video game.

game state that needs to be transferred in order to offload *DoLevel()* is just over 11 KB. When MAUI decides to offload *DoLevel()*, it will also offload the *HandleEnemies(), HandleMissiles(), HandleBonuses()* methods because they do not require any additional state to be transferred. Of the three *Handle* methods, only *HandleMissiles()* performs a significant amount of computation because of the physics modelling, and only when there are a moderate number of active missiles. The size of the missile state that needs to transferred in order to offload only *HandleMissiles()* is small: each additional active missile adds approximately 60 bytes of state transfer overhead.

To demonstrate how the solver adapts to changing network conditions and CPU costs, we examine the solver's behavior in two specific scenarios: 1) shortly after the game begins, when there are no active missiles; and 2) after the game has been running for some time and the enemies can fire up to five active missiles. When no missiles are active, MAUI will offload *DoLevel()* only when the round-trip latency to the MAUI server is less than 10 ms. When the latency is greater than 10 ms, no methods are offloaded. After the game progresses and the enemies have five active missiles, MAUI will offload *DoLevel()* when the latency is less than 30 ms, and when the latency to the MAUI server is between 30 ms and 60 ms, it will only offload *HandleMissiles()* but not *DoLevel()*. This is because the state transfer required to offload *HandleMissiles()* is much smaller than that required to offload *DoLevel()*. Finally, when the latency exceeds 60 ms, the solver decides not to offload anything. This example demonstrates that the MAUI solver can make dynamic decisions to save energy that incorporate changes in the network conditions as well as changes in the CPU consumption of individual methods over time.

8. RELATED WORK

Over the last two decades, there has been much work on supporting remote execution for mobile applications. Most of this previous work used remote execution to increase the performance and improve the availability of of resources, such as faster CPU's and more RAM, for mobile applications. Some previous efforts sought to balance the thirst for performance with energy conservation, because these goals can sometimes be contradictory [10]. In contrast, MAUI's primary goal is using remote execution to save energy. With MAUI, code is offloaded to a remote server only if MAUI predicts that remote execution ends up saving energy. Despite the difference in goals, MAUI borrows many techniques and ideas from these previous efforts.

8.1 Program Partitioning

One common approach for remote execution is to rely on programmers to modify the program to handle partitioning, state migration, and adaptation to changes in network conditions. In Spectra [9, 10], programmers provide execution plans on how to partition an application given different *fidelities*, which is an application-specific measure of quality of service. At runtime, Spectra monitors the connectivity to a remote server and chooses the execution plan that maximizes a user-provided utility function. Chroma [1, 3] builds on ideas from Spectra while making an effort to reduce the burden on the programmer. Chroma allows programmers to specify "tactics" (strategies for how a program can utilize infrastructure resources) using a declarative language. While Spectra requires users to provide the utility function to the system, Chroma relies on an external system called Prism [2] to automatically construct these utility functions by tracking users. Both Spectra and Chroma borrow ideas from Odyssey [30], an initial system that investigated operating system support for applications adapting their fidelity to changes in network bandwidth, CPU load, and battery conditions. Another common previous approach, used by Protium [41], manually partitions applications into local viewers that execute on a mobile device, and remote servers that execute the application logic. Although MAUI's architecture borrows ideas from all these systems, MAUI reduces the burden on programmers by automating many of the steps needed for program partitioning.

There are many earlier efforts on lightweight approaches to code migration, including systems for mobile objects, such as Emerald [18], Network Objects [4], Obliq [5], Rover [17], and Agent Tcl [12]. The focus of these systems is primarily on enabling code and data to easily move between nodes in a distributed system. To the best of our knowledge, none of these systems attempted to hide distribution from the programmer, nor did they focus on automating migration to optimize for energy consumption.

Previous work also investigated the use of automatic program partitioning [16, 23, 28, 29]. Coign [16] provides coarse-grain automatic partitioning of DCOM applications into client and server components without source-code modification. Kremer et al. [23] propose using static analysis to select tasks for remote execution to save energy. In [28] the authors propose statically partitioning a C-like program into a collection of nodelevel nesC programs that run on sensor motes. Hydra [40] provides support for offloading computation to specialized processors such as GPUs, NICs, and disk controllers, and it uses an ILP to decide what code to offload. Wishbone [29] uses a profile-based approach to partition applications, specified as a data-flow graph of operators, between sensor nodes and servers. MAUI retrofits many of these ideas to today's mobile landscape, where mobile devices and remote servers use different instruction set architectures (unlike [16]) and where mobile applications differ from the data-collection applications that typically run on sensor networks.

Another approach is to build replication into mobile applications and to use distributed protocols to synchronize the application's replicas. In such systems, the mobile user can run the application using any of the available replicas, whether they are local or remote. Data staging [11] and fluid replication [20] propose opportunistic use of "surrogates" (nearby untrusted and unmanaged public machines) as staging servers for the applications' replicas. Slingshot [39] extends this earlier work by adding the capability of dynamically instantiating replicas of "stateful" applications. All these systems also rely on the programmer to partition the applications and to build in support for replication.

Finally, of all the extensive previous work on program partitioning and remote execution, the system that is closest to MAUI is the OLIE system [13]. OLIE performs dynamic partitioning of Java applications at runtime, with little burden on the programmer. The OLIE runtime monitors network conditions and profiles the program behavior. However, one of the principal differences is the focus of OLIE's dynamic offloading engine on overcoming the memory resource constraints of mobile devices, in constrast with MAUI's focus on reducing energy consumption.

8.2 Process and VM Migration

Another approach to remote execution is providing operating system support for process migration, as in systems such as Sprite [8] and Amoeba [26]. More recently, Zap [31] enabled process migration using OS support for checkpoint and restart. Recent work on live migration of virtual machines [7] enables moving an entire OS and all its running applications, and the CloneCloud system [6] and Cloudlets [37] suggest applying this technique to mobile device environments. All these approaches drastically reduce the burden on the programmer, which is also one of MAUI's goals.

However, MAUI's focus on energy savings made us choose a design that is more aggressive and exploits more opportunities to offload code. This includes the ability to offload portions of a single application; we did not want to restrict MAUI's code migration to the granularity of a whole process or an entire OS.

9. CONCLUSIONS

In this paper we proposed MAUI, a system that enables fine-grained energy-aware offload of mobile code to the infrastructure. MAUI uses the benefits of managed code to reduce the burden on programmers to deal with program partitioning while maximizing the energy benefits of offloading code. This paper presented how MAUI partitions programs, how it profiled them, and how it formulated and solved program partitioning as a 0-1 integer linear programming problem. Throughout our presentation of MAUI architecture, we also showed several examples of low-level challenges discovered during our implementation. Our results showed that MAUI's energy savings and performance are impressive (up to one order of magnitude for one of our applications).

10. REFERENCES

[1] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The Case for Cyber Foraging. In *The 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[2] R. K. Balan. *Simplifying Cyber Foraging*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2006.

[3] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-Based Remote Execution for Mobile Computing. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, 2003.

[4] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. In *Proc. of the Fourteenth ACM Symposium on Operatings Systems Principles (SOSP)*, 1993.

[5] L. Cardelli. A Language with Distributed Scope. In *Proc. of the 22nd Symposium on Principles of Programming Languages (POPL)*, 1995.

[6] B.-G. Chun and P. Maniatis. Augmented Smartphone Applications Through Clone Cloud Execution. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, Monte Verita, Switzerland, May 2009.

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[8] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–785, August 1991.

[9] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, Schloss Elmau, Germany, May 2001.

[10] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. In *Proc. of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[11] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanaryanan. Data Staging on Untrusted Surrogates. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, March – April 2003.

[12] R. S. Gray. Agent Tcl: a flexible and secure mobile-agent system. In *TCLTK'96: Proceedings of the 4th USENIX Tcl/Tk Workshop, 1996*, Monterey, CA, 1996.

[13] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic. Adaptive Offloading Inference for Delivering Applications in Pervasive Computing Environments. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2003.

[14] D. Hesse. Sprint's CEO Dan Hesse Chats with Charlie Rose. `http://blueroomsolution.com/showthread.php?t=5689`, 2007.

[15] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing Application Performance Differences on Smartphones. In *Proc. of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, June 2010.

[16] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, February 1999.

[17] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles (SOSP)*, 1995.

[18] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[19] A. Kansal and F. Zhao. Fine-Grained Energy Profiling for Power-Aware Application Design. In *Proceedings of the 1st Workshop on Hot Topics in Measurement & Modeling of Computer Systems (HotMetrics)*, Annapolis, MD, June 2008.

[20] M. Kim, L. Cox, and B. Noble. Safety, Visibility, and Performance in a Wide-Area File System. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[21] J. Kincaid. YouTube Mobile Uploads Up 400% Since iPhone 3GS Launch. `http://www.techcrunch.com/2009/06/25/youtube-mobile-uploads-up-400-since-iphone-3gs-launch/`, 2009.

[22] P. Kougiouris. Use Reflection to Discover and Assess the Most Common Types in the .NET Framework. `http://msdn.microsoft.com/en-us/magazine/cc188926.aspx#S3`, 2002.

[23] U. Kremer, J. Hicks, and J. M. Rehg. Compiler-Directed Remote Task Execution for Power Management. In *Proceedings of The Workshop on Compilers and Operating Systems for Low Power (COLP)*, Philadelphia, PA, October 2000.

[24] J. Lewin. iPhone Users 30 Times More Likely To Watch YouTube Videos. `http://www.podcastingnews.com/2008/03/19/iphone-users-30-times-watch-youtube-videos/`, 2008.

[25] Monsoon Solutions Inc. Monsoon Power Monitor. `http://www.msoon.com/`.

[26] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba - A Distributed Operating System for the 1990s. *IEEE Computer*, 23:44–53, 1990.

[27] National Office of Pollution Prevention (Canada). Canadian Consumer Battery Baseline Study - Final Report. Submitted to Environment Canada, `http://www.ec.gc.ca/nopp/docs/rpt/battery/en/toc.cfm`, 2007.

[28] M. Neubauer and P. Thiemann. From Sequential Programs to Multi-Tier Applications by Program Transformation. In *Proc. of the Symposium on Principles of Programming Languages (POPL)*, Long Beach, CA, January 2005.

[29] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based Partitioning for Sensornet Applications. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation (NSDI)*, Boston, MA, April 2009.

[30] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. of the ACM Symposium on Operating System Principles (SOSP)*, 1997.

[31] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[32] M. R. Palacin. Recent advances in rechargeable battery materials: a chemists perspective. *Chem Soc Review*, 38:2565–2575, 2009.

[33] Physics2D.Net. `http://code.google.com/p/physics2d/`.

[34] R. A. Powers. Batteries for low power electronics. *Proceedings of the IEEE*, 83:687–693, April 1995.

[35] R. S. Prasad, M. Murray, C. Dovrolis, and K. Claffy. Bandwidth Estimation: Metrifcs, Measurement Techniques, and Tools. *IEEE Network*, 17(6):27–35, Nov–Dec 2003.

[36] J. Richter. *CLR via C#*. Microsoft Press; 2nd edition, 2006.

[37] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), 2009.

[38] S. Savage. Sting: a TCP-based Network Measurement Tool. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS)*, Boulder, CO, October 1999.

[39] Y.-Y. Su and J. Flinn. Slingshot: Deploying Stateful Services in Wireless Hotspots. In *Proc. of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Seattle, WA, June 2005.

[40] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff. Tapping into the Fountain of CPUs – On Operating System Support for Programmable Devices. In *Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[41] C. Young and Y. N. Lakshman. Protium, an Infrastructure for Partitioned Applications. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, Schloss Elmau, Germany, May 2001.