Energy-Aware Compilation and Execution in Java-Enabled Mobile Devices *

G. Chen, B. Kang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin
Microsystems Design Lab
The Pennsylvania State University
University Park, PA, 16802, USA
{gchen,bkang,kandemir,vijay,mji}@cse.psu.edu
R. Chandramouli
ECE Department
Stevens Institute of Technology
Hoboken, NJ 07030, USA
rchandr1@stevens-tech.edu

Abstract

Java-enabled wireless devices are preferred for various reasons such as enhanced user experience and the support for dynamically downloading applications on demand. The dynamic download capability supports extensibility of the mobile client features and centralizes application maintenance at the server. Also, it enables service providers to customize features for the clients. In this work, we extend this client-server collaboration further by offloading some of the computations (i.e., method execution and dynamic compilation) normally performed by the mobile client to the resource-rich server in order to conserve energy consumed by the client in a wireless Java environment.

In the proposed framework, the object serialization feature of Java is used to allow offloading of both method execution and bytecode-to-native code compilation to the server when executing a Java application. Our framework takes into account communication, computation and compilation energies to dynamically decide where to compile and execute a method (locally or remotely) and how to execute it (using interpretation or just-in-time compilation with different levels of optimizations).

1 Introduction

More than 483 million wireless devices will be sold to end users in 2003 and one-third of the world’s population is expected to own a wireless device by 2008 [6]. Java is expected to be a key player in this proliferation. As an example, the wireless phone maker Nokia expects the industry to ship 50 million Java-based phones by the end of 2002. When Java technology is adopted in the wireless environment, it brings unique benefits that translate into an enhanced user experience. Instead of plain text applications and the latency associated with a browser-based interface, the user is presented with rich animated graphics, fast interaction, the capability to use an application off-line, and maybe most interestingly, the capability to dynamically download new applications to the device. Further, Java is network-agnostic in the sense that Java applications can exchange data with a back-end server over any network protocol, whether it is TCP/IP, WAP, or i-mode, and different bearers, such as GSM, CDMA, TDMA, PHS, CDPD and Mobitex [2].

The dynamic linking and loading feature of Java is particularly relevant in the wireless market in that it supports loading and linking classes, methods, and fields on demand. Dynamic loading can be either from the local storage on the wireless device or from the remote server. This feature enables a dynamic download capability that allows customers to download new applications on demand as opposed to buying a device with applications pre-installed by the device manufacturer. Further, it helps a centralized service provider in upgrading software used by multiple clients and enables wireless providers to differentiate their services by offering individual customers personalized applications. A potential differentiation can be in the form of Java classes provided to the mobile clients. For example, the server can provide pre-compiled binaries for select target clients in addition to the bytecode form of the application.

Battery lifetime is a major constraint in mobile environments [11]. Although battery technology has improved over the years, it has not kept up with the computational demands of mobile systems. Thus, the designer has to optimize energy consumption to avoid either heavier battery packs or short durations between battery re-charges. For example, to limit the re-charge interval to 10 hours, it requires a 5-pound battery to operate a system that consumes 10W. 

In a mobile wireless device, it is important to optimize both computation and communication energy. There are different opportunities in such a system that allow tradeoffs between computation and communication energy costs. A lot of research has been carried out to exploit such tradeoffs to reduce the overall energy consumption (e.g., [12, 16, 15, 14]). For example, Flinn and Satyanarayanan [12] have proposed an environment where applications dynamically modify their behaviors to conserve energy by lowering data fidelity and adjusting the partition of computation tasks between client and server.

The results of these prior efforts are also relevant to Java-based mobile systems. However, due to its bytecode nature, Java has its unique aspects. To achieve the goal of “write once, run anywhere,” Java applications are presented in platform independent bytecodes. The bytecodes are executed by Java Virtual Machine (JVM). Simple JVMs execute Java

---

*This work was supported in part by NSF Grants 0073419, 0082064, 0103583, CAREER AWARDS 0093082, 0093085 & a grant from GSRC.

1This assumes the use of current Ni-Cd battery technology that offers a capacity of 20W-hour/pound.
applications by interpreting their bytecodes. More sophisticated JVMs may selectively compile parts of the applications’ bytecodes into native code at runtime using a just-in-time (JIT) compiler. Based on the “hotness” of each code fragment, some JIT compilers may even apply different levels of optimizations on different Java methods when compiling bytecodes (adaptive dynamic optimization). In wireless environments, energy-consuming JIT compilation can also be offloaded to the server, which brings the tradeoffs between local and remote compilation. Local/remote JIT compilation and adaptive dynamic optimization complicate energy (as well as performance) tradeoffs for Java-based systems, but they also create new opportunities for optimization. In this paper, we combine local/remote execution with local/remote compilation to optimize the energy behavior of Java applications running on hand-held systems.

The rest of this paper is organized as follows. In Section 2, we present our target platform and introduce the benchmark codes used in this study. Section 3 is the core part of the paper and presents an analysis of different execution and compilation strategies for a wireless Java environment. In Section 4, we compare our study to related work in the area. Finally, Section 5 concludes the paper by summarizing our major contributions.

2 Target Platforms and Benchmarks

The target remote server platform is a SPARC workstation clocked at 750MHz. The target client platform is a mobile PDA that has the ability to communicate with the server. The major components of the system include the processor core, off-chip main memory, and the communication chip set. We do not model the energy consumption of other system components, such as input/output devices, as operations using these devices need to be performed locally and hence, their energy consumption is largely unaffected by our approach.

The processor core of the target client platform is based on the microSPARC-IIep embedded processor. This core is a 100MHz, 32-bit five-stage pipelined RISC architecture that implements the SPARC v8 specification. It is primarily intended for low-cost embedded applications. It has an on-chip 8KB direct-mapped data cache and a 16KB instruction cache. The off-chip main memory is assumed to be a 32MB DRAM module.

To obtain detailed energy profiles of the processor and memory system, we have customized an energy simulator and analyzer using the Shade [9] (SPARC instruction set simulator) tool-set and simulated LaTTe [24] JVM executing a Java code. Our simulator tracks the energy consumptions in the processor core (datapath), on-chip caches, off-chip DRAM module and the wireless communication components. The energy consumed in the processor core is estimated by counting (dynamically) the number of instructions of each type and multiplying the count by the base energy consumption of the corresponding instruction. The energy consumptions of the different instruction types are obtained using a customized version of the SimplePower energy simulator [23] and are shown in Fig 1. The simulator is configured to model a five-stage pipeline similar to that of the microSPARC-IIep architecture. The DRAM energy cost is obtained from data sheets [5].

The communication components of our system support an effective data rate of 2.3Mbps and can operate with four different power control settings for transmitting data. The power consumption numbers of the transmitter power amplifier vary from a Class 1 setting for poor channel condition (power = 5.88W) to a Class 4 setting for the best (optimal) channel condition (power = 0.37W). This adaptive power setting is useful because mobile wireless channels exhibit variations that change with time and the spatial location of a mobile node. This in turn means that a fairly accurate and fast channel condition estimation mechanism is also necessary. One such mechanism that is employed by wireless standards such as the IS-95 CDMA system is the usage of a pilot channel [13]. Here, pilot CDMA signals are periodically transmitted by a base station to provide a reference for all mobile nodes. A mobile station processes the pilot signal and chooses the strongest signal among the multiple copies of the transmitted signal to arrive at an accurate estimation of its time delay, phase, and magnitude. These parameters are tracked over time to help the mobile client decide on the power-setting for its transmitter. In our simulation infrastructure, we model such tracking by varying the channel state using user supplied distributions.

The energy consumption due to communication is evaluated by modeling the individual components of the W-CDMA chip set. The power consumptions of the individual components obtained from data sheets [4, 1] are shown in Fig 2. The energy cost of communication is evaluated by using the number of bits transmitted/received, the power values of the corresponding components used, and the data rate.

We further optimize the energy consumption during re-
remote execution by placing the processor, memory and the receiver into a power-down state (mode). In the power-down state the processor still consumes leakage energy which is assumed to be 10% of the normal power consumption in our target design. The estimate of the time for executing a method remotely at the server is used by the client to determine the duration of its power-down state. When the server is done with its computation, it checks the “mobile status table” (that also contains the estimated execution times) to see if the mobile client is awake. This can be accomplished as follows.

1. The server computes the difference between the time the request was made by the client and the time when the object for that client is ready. If this difference is less than the estimated power-down duration (for the client), the server knows that the client will still be in power-down mode, and queues the data for that client until it wakes up. In case the server-side computation is delayed, we incur the penalty of early re-activation of the client from the power-down state.

2. Fig 3 lists the applications used in this study. Median-Filter, High-Pass-Filter, and Edge-Detector are codes that are used frequently in embedded image and video processing. Function-Evaluator and Path-Finder are applications available in many hand-held devices and digital assistants. Sorting is a frequently-used utility package in many application domains. Our last two benchmarks (ess and db) are from SpecJVM98. As these two are benchmarks for high-performance JVMs, to make them behave more like typical applications running on embedded systems, their smallest input size (s1 dataset) was used. To make offloading possible, some necessary modifications have also been made to their codes. Their core logic, however, is carefully retained. In the rest of this paper, we use these eight benchmark codes to evaluate different execution and compilation strategies.

3 Analysis of Execution and Compilation Strategies

In our approach, we attempt to partition the activities during Java code execution across the mobile client and the resource-rich server. The goal of this partitioning is to reduce the energy consumed by the mobile client in executing the application. In particular, we focus on two important aspects of Java code execution, namely, dynamic compilation and method execution.

1. At each invocation of a potential method, our JVM dynamically decides whether to execute it locally or remotely.

In our prototype implementation, the decision logic is encoded as a “helper method” for each potential method. Helper methods can be either defined by the programmer at development time or automatically created using profile data when the application is deployed on the server for the client to download. Helper methods are incorporated into the same class file that contains potential methods. Our JVM implicitly invokes the corresponding helper method for each invocation of the methods annotated as potential method. The helper method evaluates and compares the computation and communication costs of the potential method. If the potential method is determined to be executed locally, the helper method will generate a compilation plan that contains (i) the names of the potential method and the methods that will be called by the potential method and (ii) the desirable optimization level for these methods. Those methods whose current optimization levels are lower than their desirable level will then be compiled or recompiled to the right level before their first invocation. If the potential method is executed remotely, until it returns, all the methods called by this method are also executed remotely on the server. Our prototype has been validated using two SPARC workstations, one acting as a server and the other as the mobile client and has also been implemented on top of the simulation framework for energy characterization.

2. We first evaluate tradeoffs between remote (server-based) and local (client-based) method execution. Six different execution strategies were considered. In the first strategy, denoted as Remote (R), all potential methods are executed remotely at the server. In the second execution strategy, called Interpreter (I), all methods are executed locally (i.e., on the client) in the bytecode form. Note that this strategy incurs no compilation or communication energy. In the next three execution strategies (called Local1 (L1), Local2 (L2), and Local3 (L3)), the methods are compiled with different degrees of optimizations and executed on the client (in the form of native code). Local1 performs no special optimizations in compiling the code and just translates the bytecode to native
form before the first execution. Local2, performs some well-
known optimizations during the compilation; these include common sub-expression elimination, loop invariant code motion, strength reduction, and redundancy elimination. Local3, performs virtual method inlining [8, 24] in addition to the optimizations performed by Local2. These five strategies are all static as they fix the execution strategy for each method in a given application. Besides these static strategies, we evaluate two adaptive strategies: Adaptive Execution/Local Compilation (AL) and Adaptive Execution/Adaptive Compilation (AA). The AL strategy determines, for each potential method, the best execution strategy (Remote, Local1, Local2, Local3, or Interpreter) dynamically just before the execution. In addition to local/remote execution modes, AA tries to further optimize the client's energy consumption by exploiting the tradeoffs between local/remote compilation. All adaptive strategy results include the overhead for the dynamic decision making. Fig 5 gives a summary of the static and dynamic (adaptive) strategies evaluated in this work.

3.1 Analysis of Static Strategies

In this subsection, we investigate the tradeoffs between the different static strategies. For the experiments in this subsection, for the native code strategies (Local1, Local2, and Local3), we perform compilation on the mobile client. The energy numbers presented in this subsection include the energy cost of loading and initialing the compiler classes. Fig 6 on page 5 shows the energy consumption of the static strategies (R, I, L1, L2, and L3) for three of our benchmarks. All energy values are normalized with respect to that of L1. For the bar denoting remote execution (R), the additional energies required when channel condition is poor is shown using stacked bars over the Class 4 operation (the best channel condition). For each benchmark, we selected two different values for the size parameters (See Fig 3). It can be observed that the optimal static strategy varies depending on the input parameter size, and current channel condition. As an example, for a small image size (64x64), remote execution (R) is the preferable strategy for high frequency when the channel condition is Class 4, 3, or 2. But, when the channel condition degrades to Class 1, the interpreter strategy (I) becomes the best choice. On the other hand, when the image size is increased to 512x512, the best strategy becomes L2. Similar differences in optimal strategies can be observed for the other benchmarks as well. These results motivate the need for dynamically determining the execution strategy. In particular, these experimental results show that, depending on the input parameters and channel conditions, different execution strategies might be preferable for the same Java method.

### 3.2 Analysis of the Adaptive Execution Strategy

In this subsection, we present an adaptive approach that chooses the most appropriate execution strategy for each method each time it is invoked. Specifically, when the client is about to execute a potential method, it invokes the helper method to make a decision as to whether to execute the method locally or remotely. If the method is to be executed locally, the client also needs to select a desirable optimization. Since compilation, if necessary, is always performed locally in this strategy, we call it Adaptive Execution/Local Compilation (AL).

In order to make the remote/local execution decision, the client needs to estimate the remote execution energy, local compilation energy, and local execution energy. Since, given a specific platform, a method and an optimization level, the compilation cost is constant, in our prototype implementation, the local compilation energy values (for each potential method and each optimization level) are obtained by profiling; these values are then incorporated into the applications' class files as static final variables. To make this strategy platform independent, we specify a scaling factor for each platform. These variables are then referred to by the helper methods.

We employ a curve fitting based technique to estimate the energy cost of executing a method locally. It should be noted that this cost includes the energy consumed not only by the potential methods themselves but also by the methods called by the potential methods. To verify the accuracy of these curves, the points from these curves were compared with 20 other data points (for each application) from actual executions. We found that our curve fitting based energy estimation is within 2% of the actual energy value. This input parameter based energy estimation is observed to work well in all the methods that consume a significant part of execution time and
energy in our benchmark suite. Our approach, however, has a limitation in that it may not work well for methods whose parameter sizes are not representative of their execution costs. To estimate the remote execution energy, the client uses the channel condition, the sizes of the input objects (which are known at the time of method invocation), and the estimated sizes of output objects. The formulation obtained from curve fitting is then encoded in the helper methods. Since the cost estimation is performed at runtime and introduces overhead, the calculation performed for the estimation should not be too complex. For many methods, the energy cost can be predicted based on their parameters pretty accurately with simple calculations.

Once the local compilation and execution (Interpreter, Local1, Local2, and Local3), and remote execution energies are estimated, AL uses the following strategy to decide how and where to execute the method. Let us first define the following notation to explain our strategy:

\[
E_o(m, s) : \text{Estimated energy for local execution}
\]
\[
E_c(m) : \text{Estimated energy for local compilation}
\]
\[
E''(m, s, p) : \text{Estimated energy for remote execution}
\]

The subscript \(o\) in this notation refers to the optimization level considered; \(m\) denotes the method in question; \(p\) is the communication power required by current channel condition; and \(s\) is the parameter(s) of the method that determines the computation complexity of local execution and the sizes of sent and received data for remote execution (in the rest of this paper, we refer to this parameter as the “size parameter”). In the following discussion, without loss of generality, we assume three optimization levels: \(o_1\) (most primitive, corresponds to Local1), \(o_2\), and \(o_3\) (most aggressive, corresponds to Local3). Suppose that a method has been executed \(k\) times using the current level of optimization. AL optimistically assumes that this method will be executed \(k\) more times in the remaining portion of the optimization. We predict the future parameter size and communication power based on the weighted average of current and past values. Specifically, at
the $k^h$ invocation, we use the following formulations to estimate the future parameter size and communication power ($s_k$ and $p_k$ are the current parameter size and communication power, respectively):

$$\bar{s}_k = u_1 \bar{s}_{k-1} + (1 - u_1) s_k;$$
$$\bar{p}_k = u_2 \bar{p}_{k-1} + (1 - u_2) p_k;$$
$$0 \leq u_1, u_2 \leq 1.$$

$\bar{s}_k$ is the expected future parameter size and $\bar{p}_k$ is the expected future communication power after the $k^h$ method invocation. $u_1$ and $u_2$ are used to appropriately weight the current and history values. According to our experiments, setting both $u_1$ and $u_2$ to 0.7 yields satisfactory results. Before each method invocation, a decision is made as to whether it is beneficial from the energy perspective to employ a more aggressive optimization. For example, after $k$ times of bytecode executions, our JVM checks which of the following energy values is minimum:

$$E_I = kE(m, \bar{s}_k)$$
$$E_R = kE_m^{\text{init}}(m, \bar{s}_k, \bar{p}_k)$$
$$E_{L1} = E_{o_I}(m) + kE_{o_s}(m, \bar{s}_k)$$
$$E_{L2} = E_{o_I}(m) + kE_{o_s}(m, \bar{s}_k)$$
$$E_{L3} = E_{o_I}(m) + kE_{o_s}(m, \bar{s}_k)$$

$E_I$, $E_R$, $E_{L1}$, $E_{L2}$ and $E_{L3}$ are the expected energies that will be consumed if all the remaining invocations of this method (i.e., an estimated total of $k$ invocations) are executed using Interpreter, Remote, Local1, Local2 and Local3 modes, respectively. The alternative that gives the minimum energy is chosen as the preferred mode of execution. If either the bytecode or remote execution is preferred, no compilation is performed; otherwise, the compilation is performed (locally) before execution. If a particular compiled form is already available from previous compilation, the corresponding $E_{o_s}(m)$ term is omitted when evaluating the alternatives. As discussed earlier, if remote execution is favored, the input object is serialized and transmitted to the server. The client is then powered down for a specified interval based on the estimated execution time on the server. It then awakens to receive the result of the computation from the server. The server also serializes and sends the output object to the client. However, in a mobile environment, there could be loss of connection to the server for prolonged time durations. When the result is not obtained within a predefined time threshold, connectivity to server is considered lost and execution begins locally.

In order to evaluate the adaptive execution strategy, we compare its energy behavior with the five static strategies. Each benchmark is executed by choosing three different situations having different channel condition and input distribution. The distributions have been carefully selected to mimic these three situations: (i) the channel condition is predominantly good and one input size dominates; (ii) the channel condition is predominantly poor and one input size dominates; and (iii) both channel condition and size parameters are uniformly distributed. Executing each of the eight applications under these three situations contributes to 24 scenar-

![Fig 7. Average of normalized energy consumptions of eight benchmarks. Left eight bars: channel condition is predominantly good and one input size dominates. Middle eight bars: channel condition is predominantly poor and one input size dominates. Right eight bars: both channel condition and size parameters are uniformly distributed. All values are normalized with respect to L1.](image)

ios. For each scenario, an application is executed 300 times with inputs and channel conditions selected to meet the required distribution.

Fig 7 shows the energy consumption of different execution strategies, normalized with respect to L1. Note that these values are averaged over all eight benchmarks. We observe that under all the three situations (i, ii, iii), the adaptive strategy AL consumes less energy than the static strategies. Compared to static strategies, AL introduces overheads, but, since the calculation performed to evaluate the computation costs and make decisions is simple, these overheads are too small to highlight in the graph. We observe from Fig 7 that AL outperforms all static strategies in all three situations. Specifically, it consumes 25%, 10%, and 22% less overall energy than the best static strategy (L2) in situations i, ii and iii, respectively. These results emphasize the importance of dynamic adaptive execution.

Our adaptive approach also has an influence on the performance of the code. This performance impact is dependent on the relative performance of the mobile client and the server as well as the bandwidth limitations of the wireless channel. When using a 750MHz SPARC server and a 2.3Mbps wireless channel, we find that performance improvements (over local client execution) vary between 2.5 times speedup and 10 times speedup based on input sizes whenever remote execution is preferred. However, it must be observed that remote execution could be detrimental to performance if the communication time dominates the computation time for the potential method offloaded. Such cases are also found to be detrimental for energy consumption.

### 3.3 Analysis of Adaptive Compilation Strategy

When a class is loaded, Java Virtual Machine verifies the class file to guarantee that the class file is well formed and that the program does not violate any security policies [17]. This verification mechanism does not work for native code. However, if the server is trusted and the communication channel is safe, the security rules of JVM can be relaxed.
to allow JVM to download, link and execute pre-compiled native code of some methods from the server.

Fig 8 provides the (compilation) energy consumed when a client either compiles methods of an application or downloads their remotely pre-compiled native code from the server. We again consider three levels of compiler optimizations. For the remote compilation, the channel condition influences the energy cost as can be seen in the last four columns. We also observe that as the degree of optimization increases, the energy expended in local compilation increases. However, for remote compilation, there are cases where a more aggressive optimization reduces the code size and consequently results in less communication energy than that of a less aggressive optimization. For example, in sort, in going from Level2 to Level3 optimization, the remote compilation energy reduces. Fig 8 also shows that in many cases, remote compilation consumes less energy than local compilation with the same optimization level (e.g., db).

For applications where a significant portion of the energy is consumed in compiling the bytecodes to native code (due to short overall execution time, due to small number of method invocations, or due to frequent recompilations which is required to adapt to changing external conditions and/or values of runtime variables [7]), it may be very beneficial to exploit the energy tradeoffs between local compilation and downloading the pre-compiled code.

We assume that the server supports a limited number of preferred client types for remote compilation and provides the pre-compiled versions of the methods for specific target architectures. Whenever remote compilation is desired, the client passes the fully qualified method name to the server and receives the pre-compiled method from the server. This pre-compiled method also contains necessary information that allows the client JVM to link it with code on the client side. Note that the sizes of pre-compiled versions of methods can differ depending on the optimization level. Optimizations such as method inlining, object inlining, and code duplication [8] can increase the size of the compiled code while providing performance gains at execution time. Such optimizations, in general, provide opportunities for tradeoffs between the size of the compiled code and the performance gains may be critical in a mobile environment.

So far, we have only considered local compilation. We now enhance our adaptive strategy presented earlier by considering the remote compilation option. This enhanced strategy is referred to as AA (Adaptive execution/Adaptive compilation) in the rest of this paper. In this strategy, when a decision is made to compile the method, the client computes the energy needed to receive the pre-compiled code and compares it with the cost of generating this version locally from the bytecodes. The channel condition is used to estimate the energy cost for transmitting the method name and receiving the compiled code.

We observe from Fig 7 that AA saves more energy than AL. The extra savings of AA come from two sources. First, AA reduces compilation cost (energy) by selecting the best compilation alternative from the energy perspective. Second, since the cost of downloading a more optimized version can be less than the cost of local compilation to a less aggressive version, AA can also reduce execution energy.

### Table 1

<table>
<thead>
<tr>
<th>App</th>
<th>Opt Level</th>
<th>Local Compilation</th>
<th>Remote Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>L1</td>
<td>100.0</td>
<td>286.3, 150.8</td>
</tr>
<tr>
<td>m</td>
<td>L2</td>
<td>172.1</td>
<td>286.3, 150.8</td>
</tr>
<tr>
<td>m</td>
<td>L3</td>
<td>363.3</td>
<td>317.5, 182.1</td>
</tr>
<tr>
<td>ed</td>
<td>L1</td>
<td>100.0</td>
<td>80.3, 56.5</td>
</tr>
<tr>
<td>ed</td>
<td>L2</td>
<td>209.2</td>
<td>77.1, 63.3</td>
</tr>
<tr>
<td>ed</td>
<td>L3</td>
<td>230.4</td>
<td>77.1, 63.3</td>
</tr>
<tr>
<td>pf</td>
<td>L1</td>
<td>100.0</td>
<td>124.8, 76.3</td>
</tr>
<tr>
<td>pf</td>
<td>L2</td>
<td>163.0</td>
<td>124.8, 76.3</td>
</tr>
<tr>
<td>pf</td>
<td>L3</td>
<td>174.9</td>
<td>124.8, 76.3</td>
</tr>
<tr>
<td>hpf</td>
<td>L1</td>
<td>100.0</td>
<td>80.1, 75.5</td>
</tr>
<tr>
<td>hpf</td>
<td>L2</td>
<td>309.9</td>
<td>47.5, 42.9</td>
</tr>
<tr>
<td>hpf</td>
<td>L3</td>
<td>329.8</td>
<td>47.5, 42.9</td>
</tr>
<tr>
<td>mf</td>
<td>L1</td>
<td>100.0</td>
<td>92.3, 72.4</td>
</tr>
<tr>
<td>mf</td>
<td>L2</td>
<td>187.5</td>
<td>92.3, 72.4</td>
</tr>
<tr>
<td>mf</td>
<td>L3</td>
<td>238.0</td>
<td>96.9, 77.0</td>
</tr>
<tr>
<td>sort</td>
<td>L1</td>
<td>100.0</td>
<td>244.0, 128.5</td>
</tr>
<tr>
<td>sort</td>
<td>L2</td>
<td>185.6</td>
<td>244.0, 128.5</td>
</tr>
<tr>
<td>sort</td>
<td>L3</td>
<td>205.2</td>
<td>217.4, 101.9</td>
</tr>
<tr>
<td>db</td>
<td>L1</td>
<td>100.0</td>
<td>120.5, 88.2</td>
</tr>
<tr>
<td>db</td>
<td>L2</td>
<td>144.1</td>
<td>113.1, 80.7</td>
</tr>
<tr>
<td>db</td>
<td>L3</td>
<td>213.8</td>
<td>120.5, 88.2</td>
</tr>
</tbody>
</table>

Fig 8. Local and remote compilation energies. For each application, all values are normalized with respect to the energy consumed when local compilation with optimization Level1 is employed.

### 4 Related Work

There have been prior attempts to exploit the interaction between mobile clients and resource-rich servers for energy savings [16, 14, 20, 18]. Rudenko et al. [20] performed a series of remote process executions to show the effectiveness of remote executions. They measured the impact of the input size on the energy saved using remote execution. In contrast to our approach that makes execution choices at method granularity, their approach executes the entire application remotely. Öthman and Hailes [18] performed simulations to show that battery life can be extended by up to 21% using migration at the process level. Kremer et al. [14] proposed a framework that analyzes the entire program in order to identify candidate code fragments for remote or local execution. Li et al. [16] developed a program partitioning scheme that uses a task mapping algorithm to statically classify tasks as server and client tasks. This scheme uses profiling information on computation time and data sharing at the procedure level to optimize energy consumption. In a more recent work [15], they used a maximum-flow/minimum-cut algorithm to optimize the partition/allocation problem as opposed to the branch-and-bound policy used in [16]. In our approach, we focus on a Java-based environment and dynamically decide whether to execute locally or remotely based on input sizes, computational complexity, and channel conditions. Further, in our approach, an additional challenge is to determine the form of local execution that would provide the most energy-efficient solution.

In order to address the tradeoff between the slow speed of interpreted bytecode execution and the memory/performance overheads of dynamic compilation, Turbo and Quicksilver [10, 21] use pre-compiled binaries. In contrast to our approach of dynamically downloading the binaries from a remote server, Turbo and Quicksilver pre-compile bytecode into native code and place the generated code in the de-
vice’s ROM image. This is a good approach for applications shipped with the device, but is problematic for applications shipped independent of the device [10].

Remote compilation is employed in [3] and [19] to avoid the memory overheads of a JIT compiler. Whenever JCod determines that a method should be compiled, it sends it to a compilation server on the local network. The compilation server replies by sending the native code back to JCod, which installs it within the VM. From that time on, the native code is used, resulting in a gain in speed only for the part of the application for which it is worthwhile. In contrast to JCod, our optimization is performed to conserve energy. Palm et al. [19] shows that offloading JIT compilation to a resource rich server saves energy for all SpecJVM98 benchmarks. In contrast, our technique attempts to dynamically select between local and remote compilation based on the method and channel conditions. Further, we augment energy savings through remote execution modes. Teodorescu and Pandey [22] proposed a Java-based ubiquitous computation environment that makes use of remote JIT compilation. Their work tries to reduce the memory requirement on the client side and does not focus on energy tradeoffs.

5 Conclusions

An important issue when executing a Java application on mobile devices is its energy consumption. Our work emphasizes that the choice of compilation/execution strategy for the machine-neutral Java bytecodes critically impacts the energy consumed by the device.

In particular, the conclusions from this work can be summarized as follows. First, we observe that interpreted Java execution is generally more costly in terms of energy as compared to execution using compiled code. However, the compilation itself involves an energy cost and requires additional memory footprint for storing the compiled code. Hence, one can employ remote compilation which can reduce both the energy and memory overheads. However, remote compilation also incurs the energy cost of receiving the compiled code from the server. Thus, if energy is the constraint of focus, we can dynamically decide between compiling locally and remotely. Mobile systems with larger memories are beginning to emerge that make such tradeoffs for supporting local or dynamic compilation useful. Another technique for energy saving that we present is remote execution of methods. We dynamically evaluate the trade-offs between computational and communication cost based on input parameters and channel conditions in deciding between local and remote executions. The results from the comparison of our dynamic approach with static execution approaches justify the need for adaptive strategies to obtain the best energy behavior.

References