Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems

Chiyoung Seo[1], Sam Malek[2], and Nenad Medvidovic[1]

[1]Computer Science Department, University of Southern California, Los Angeles, CA 90089-0781, U.S.A. {cseo,neno}@usc.edu

[2]Department of Computer Science, George Mason University, Fairfax, VA 22030-4444, U.S.A. smalek@gmu.edu

Abstract. Efficiency with respect to energy consumption has increasingly been recognized as an important quality attribute for distributed software systems in embedded and pervasive environments. In this paper we present a framework for estimating the energy consumption of distributed software systems implemented in Java. Our primary objective in devising the framework is to enable an engineer to make informed decisions when adapting a system's architecture, such that the energy consumption on hardware devices with a finite battery life is reduced, and the lifetime of the system's key software services increases. Our framework explicitly takes a component-based perspective, which renders it well suited for a large class of today's distributed, embedded, and pervasive applications. The framework allows the engineer to estimate the distributed system's energy consumption at system construction-time and refine it at runtime. In a large number of distributed application scenarios, the framework showed very good precision on the whole, giving results that were within 5% (and often less) of the actual energy consumption incurred by executing the software. Our work to date has also highlighted the framework's practical applications and a number of possible enhancements.

Keywords. Distributed systems, energy consumption, Java, component-based software

1 Introduction

Modern software systems are predominantly distributed, embedded, and pervasive. They increasingly execute on heterogeneous platforms, many of which are characterized by limited resources. One of the key resources, especially in long-lived systems, is battery power. Unlike the traditional desktop platforms, which have uninterrupted, reliable power sources, a newly emerging class of computing platforms have finite battery lives. For example, a space exploration system comprises satellites, probes, rovers, gateways, sensors, and so on. Many of these are "single use" devices that are not rechargeable. In such a setting, minimizing the system's power consumption, and thus increasing its lifetime, becomes an important quality-of-service concern.

The simple observation guiding our research is that if we could estimate the energy cost of a given software system in terms of its constituent components ahead of its actual deployment, or at least early on during its runtime, we would be able to take appropriate, possibly automated, actions to prolong the system's life span: unloading unnecessary or expendable components, redeploying highly energy-intensive components to more capacious hosts, collocating frequently communicating components, and so on.

To this end, we have developed a framework that estimates the energy consumption of a distributed Java-based software system at the level of its components. We chose Java because of its intended use in network-based applications, its popularity, and very importantly, its reliance on a virtual machine, which justifies some simplify-

ing assumptions possibly not afforded by other mainstream languages. We have evaluated our framework for precision on a number of distributed Java applications, by comparing its estimates against actual electrical current measurements. In all of our experiments the framework has been able to estimate the power consumed by a distributed Java system to within 5% of the actual consumption.

One novel contribution of our estimation framework is its component-based perspective. To facilitate component-level energy cost estimates, we suggest a computational energy cost model for a software component. We integrate this model with the component's communication cost model, which is based on the experimental results from previous studies. This integrated model results in highly accurate estimates of a component's overall energy cost. Furthermore, unlike most previous power estimation tools for embedded applications, we explicitly consider and model the energy overhead of a host's OS and an application's runtime platform (e.g., JVM) incurred in facilitating and managing the execution of software components. This further enhances the accuracy of our framework in estimating a distributed software system's energy consumption. Another contribution of our work is its ability to adjust energy consumption estimates at runtime efficiently and automatically, based on monitoring the changes in a small number of easily tracked system parameters (e.g., size of data exchanged over the network, inputs to a component's interfaces, invocation frequency of each interface, etc.).

In the remainder of this paper we first present the related research in the energy estimation and measurement areas (Section 2). We then introduce our energy estimation framework (Section 3) and detail how it is applied to component-based Java systems (Section 4). This is followed by our evaluation strategy (Section 5) and results (Section 6). The paper concludes with a discussion of planned applications of this research (Section 7).

2 Related Work

Several studies have profiled the energy consumption of Java Virtual Machine (JVM) implementations. Farkas et al. [3] have measured the energy consumption of the Itsy Pocket Computer and the JVM running on it. They have discussed different JVMs' design trade-offs and measured their impact on the JVM's energy consumption. Lafond et al. [11] have showed that the energy required for memory accesses usually accounts for 70% of the total energy consumed by the JVM. However, none of these studies suggest a model that can be used for estimating the energy consumption of a distributed Java-based system.

There have been several tools that estimate the energy consumption of embedded operating systems (OSs) or applications. Tan et al. [19] have investigated the energy behaviors of two widely used embedded OSs, μC/OS [10] and Linux, and suggested their quantitative macro-models, which can be used as OS energy estimators. Sinha et al. [16] have suggested a web-based tool, *JouleTrack*, for estimating the energy cost of an embedded software running on StrongARM SA-1100 and Hitachi SH-4 microprocessors. While they certainly informed our work, we were unable to use these tools directly in our targeted Java domain because none of them provide generic energy consumption models, but instead have focused on individual applications running on

specific OSs and platforms.

Recently, researchers have attempted to characterize the energy consumption of the Transmission Control Protocol (TCP) [15]. Singh et al. [15] measured the energy consumption of variants of TCP (i.e., Reno, Newreno, SACK, and ECN-ELFN) in ad-hoc networks, and showed that ECN-EFLN has a lower energy cost than the others. These studies also show that, since TCP employs a complicated mechanism for congestion control and error recovery, modeling its exact energy consumption remains an open problem. While we plan to incorporate into our framework the future advancements in this area, as detailed in the next section we currently rely on the User Datagram Protocol (UDP), which does not provide any support for congestion control, retransmission, error recovery, and so on.

Several studies [4,21] have measured the energy consumption of wireless network interfaces on handheld devices that use UDP for communication. They have shown that the energy usage by a device due to exchanging data over the network is directly linear to the size of data. We use these experimental results as a basis for defining a component's communication energy cost.

Finally, this research builds on our previous works [12,13], where we have outlined the architecture of the framework [12], and the overall energy estimation process [13]. In this paper, we provide a comprehensive and detailed description of the framework, runtime refinement of its estimates, its practical applications, and an extensive evaluation of its accuracy in the context of several applications.

3 Energy Consumption Framework

We model a distributed software system's energy consumption at the level of its components. A *component* is a unit of computation and state. In a Java-based application, a component may comprise a single class or a cluster of related classes. The energy cost of a software component consists of its *computational* and *communication* energy costs. The computational cost is mainly due to CPU processing, memory access, I/O operations, and so forth, while the communication cost is mainly due to the data exchanged over the network. In addition to these two, there is an additional energy cost incurred by an OS and an application's runtime platform (e.g., JVM) in the process of managing the execution of user-level applications. We refer to this cost as *infrastructure energy overhead*. In this section, we present our approach to modeling each of these three energy cost factors. We conclude the section by summarizing the assumptions that underlie our work.

3.1 Computational Energy Cost

In order to preserve a software component's abstraction boundaries, we determine its computational cost at the level of its public interfaces. A component's interface corresponds to a service it provides to other components. While there are many ways of implementing an interface and binding it to its caller (e.g., RMI, event exchange), in the most prevalent case an interface corresponds to a method. In Section 3.2 we discuss other forms of interface implementation and binding (e.g., data serialization over sockets).

As an example, Figure 1 shows a component c_1 on host H_1, c_1's provided interfaces, and the invocation of them by remote components. Given the energy cost *iCompEC* resulting from invoking an interface I_i, and the total number b_i of invocations for the interface I_i, we can calculate the overall energy cost of a component c_1 with n interfaces (in *Joule*) as follows:

$$cCompEC(c_1) = \sum_{i=1}^{n} \sum_{j=1}^{b_i} iCompEC(I_i, j) \qquad \textbf{\textit{Eq. 1}}$$

In this equation, $iCompEC(I_i,j)$, the computational energy cost due to the j_{th} invocation of I_i, may depend on the input parameter values of I_i and differ for each invocation.

In Java, the effect of invoking an interface can be expressed in terms of the execution of JVM's 256 Java bytecode types, and its native methods. Bytecodes are platform-independent codes interpreted by JVM's interpreter, while native methods are library functions (e.g., `java.io.FileInputStream`'s `read()` method) provided by JVM. Native methods are usually implemented in C and compiled into dynamic link libraries, which are automatically installed with JVM. JVM also provides a mechanism for synchronizing threads via an internal implementation of a *monitor*.

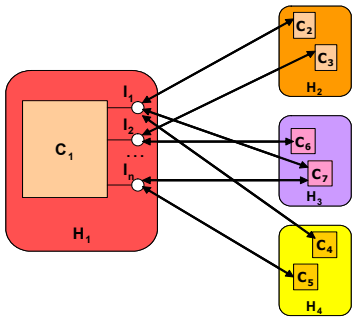


Figure 1. Interactions among distributed components.

Each Java statement maps to a specific sequence of bytecodes, native methods, and/or monitor operations. Based on the 256 bytecodes, m native methods, and *monitor* operations that are available on a given JVM, we can estimate the energy cost $iCompEC(I_i,j)$ of invoking an interface as follows:

$$iCompEC(I_i, j) = \left(\sum_{k=1}^{256} bNum_{k,j} \times bEC_k\right) + \left(\sum_{l=1}^{m} fNum_{l,j} \times fEC_l\right) + mNum_j \times mEC \qquad \textbf{\textit{Eq. 2}}$$

where $bNum_{k,j}$ and $fNum_{l,j}$ are the numbers of each type of bytecode and native method, and $mNum_j$ is the number of monitor operations executed during the j_{th} invocation of I_i. bEC_k, fEC_l, and mEC represent the energy consumption of executing a given type of bytecode, a given type of native method, and a single monitor operation, respectively. These values must be measured before Equation 2 can be used. Unless two platforms have the same hardware setup, JVMs, and OSs, their respective values for bEC_k, fEC_l, and mEC will likely be different. We will explain how these values can be obtained in Section 5.

3.2 Communication Energy Cost

Two components may reside in the same address space and thus communicate locally, or in different address spaces and communicate remotely. When components are part of the same JVM process but running in independent threads, the communication among the threads is generally achieved via native method calls (e.g., `java.lang.Object`'s `notify()` method). A component's reliance on native methods has already been accounted for in calculating its computational cost from

Equation 2. When components run as separate JVM processes on the same host, Java sockets are usually used for their communication. Given that JVMs generally use native methods (e.g., `java.net.SocketInputStream`'s `read()`) for socket communication, this is also captured by a component's computational cost.

In remote communication, the transmission of messages via network interfaces consumes significant energy. Given the communication energy cost *iCommEC* due to invoking an interface I_i, and the total number b_i of invocations for that interface, we can calculate the overall communication energy consumption of a component c_1 with n interfaces (expressed in *Joule*) as follows:

$$cCommEC(c_1) = \sum_{i=1}^{n} \sum_{j=1}^{b_i} iCommEC(I_i, j) \quad \textbf{Eq. 3}$$

In this equation, $iCommEC(I_i,j)$, the energy cost incurred by the j_{th} invocation of I_i, depends on the amount of data transmitted or received during the invocation and may be different for each invocation. Below we explain how we have modeled $iCommEC(I_i,j)$.

We focus on modeling the energy consumption due to the remote communication based on UDP. Since UDP is a light-weight protocol (e.g., it provides no congestion control, retransmission, and error recovery mechanisms), it is becoming increasingly prevalent in resource-constrained pervasive domains [2,20]. Previous research [4,21] has shown that the energy consumption of wireless communication is directly proportional to the size of transmitted and received data. Based on this, we quantify the communication energy consumption due to the j_{th} invocation of component c_1's interface I_i on host H_1 by component c_2 on host H_2 as follows:

$$iCommEC(I_i, j) = \left(tEvtSize_{c_1,c_2} \times tEC_{H_1} + tS_{H_1}\right) + \left(rEvtSize_{c_1,c_2} \times rEC_{H_1} + rS_{H_1}\right) \quad \textbf{Eq. 4}$$

Parameters *tEvtSize* and *rEvtSize* are the sizes (e.g., *KB*) of transmitted and received messages on host H_1 during the j_{th} invocation of I_i. The remaining parameters are host-specific. tEC_{H1} and rEC_{H1} are the energy costs (*Joule*/*byte*) on host H_1 while it transmits and receives a unit of data, respectively. tS_{H1} and rS_{H1} represent constant energy overheads associated with device state changes and channel acquisition [4].

In Equation 4, the energy values of *tEC*, *rEC*, *tS*, *rS* are constant and platform-specific.[1] The system parameters that need to be monitored on each host are only the sizes of messages exchanged (*tEvtSize* and *rEvtSize*, which include the overhead of network protocol headers). Note that transmission or receipt failures between the sender and receiver hosts do not affect our estimates: UDP does not do any processing to recover from such failures, while our framework uses the actual amount of data transmitted and received in calculating the communication energy estimates.

3.3 Infrastructure Energy Consumption

Once the computational and communication costs of a component have been calculated based on its interfaces, its overall energy consumption is determined as follows:

$$overallEC(c) = cCompEC(c) + cCommEC(c) \quad \textbf{Eq. 5}$$

[1] We will elaborate on how these parameters are determined for an actual host in Section 6.2

However, in addition to the computational and communication energy costs, there are additional energy costs for executing a Java component incurred by JVM's garbage collection and implicit OS routines. During garbage collection, all threads except the Garbage Collection (GC) thread within the JVM process are suspended temporarily, and the GC thread takes over the execution control. We estimate the energy consumption resulting from garbage collection by determining the average energy consumption rate *gEC* of the GC thread (*Joule/second*) and monitoring the total time *tGC* the thread is active (*second*). In Section 5 we describe how to measure the GC thread's execution time and its average energy consumption rate.

Since a JVM runs as a separate user-level process in an OS, it is necessary to consider the energy overhead of OS routine calls for facilitating and managing the execution of JVM processes. There are two types of OS routines:

1. explicit OS routines (i.e., system calls), which are initiated by user-level applications (e.g., accessing files, or displaying text and images on the screen); and
2. implicit OS routines, which are initiated by the OS (e.g., context switching, paging, and process scheduling).

Java applications initiate explicit OS routine calls via JVM's native methods. Therefore, Equation 2 already accounts for the energy cost due to the invocation of explicit OS routines. However, we have not accounted for the energy overhead of executing implicit OS routines. Previous research has shown that process scheduling, context switching, and paging are the main consumers of energy due to implicit OS routine calls [19]. By considering these additional energy costs, we can estimate the overall infrastructure energy overhead of a JVM process *p* as follows:

$$ifEC(p) = \left(tGC_p \times gEC\right) + \left(csNum_p \times csEC\right) + \left(pfNum_p \times pfEC\right) + \left(prNum_p \times prEC\right) \qquad \textbf{\textit{Eq. 6}}$$

Recall that *gEC* is the average energy consumption rate of the GC thread, while tGC_p is the time that the GC thread is active during the execution of process *p*. $csNum_p$, $pfNum_p$, and $prNum_p$ are, respectively, the numbers of context switches, page faults, and page reclaims that have occurred during the execution of process *p*. *csEC*, *pfEC*, and *prEC* are, respectively, the energy consumption of processing a context switch, a page fault, and a page reclaim. We should note that *csEC* includes the energy consumption of process scheduling as well as a context switch. This is due to the fact that in most embedded OSs a context switch is always preceded by process scheduling [19].

Since there is a singleton GC thread per JVM process, and implicit OS routines operate at the granularity of processes, we estimate the infrastructure energy overhead of a distributed software system in terms of its JVM processes. In turn, this helps us to estimate the system's energy consumption with higher accuracy. Unless two platforms have the same hardware configurations, JVMs, and OSs, the energy values of *gEC*, *csEC*, *pfEC*, and *prEC* on one platform may not be the same as those on the other platform. We will describe how these values can be obtained for an actual host in Section 5.

Once we have estimated the energy consumption of all the components, as well as the infrastructure energy overhead, we can estimate the system's overall energy con-

sumption as follows: $$systemEC = \sum_{i=1}^{cNum} overallEC(c_i) + \sum_{j=1}^{pNum} ifEC(p_j) \qquad \textbf{\textit{Eq. 7}}$$

where *cNum* and *pNum* are, respectively, the numbers of components and JVM processes in the distributed software system.

3.4 Assumptions

In formulating the framework introduced in this section, we have made several assumptions. First, we assume that the configuration of all eventual target hosts is known in advance. This allows system engineers to closely approximate (or use the actual) execution environments in profiling the energy consumption of applications prior to their deployment and execution. As alluded above, and as will be further discussed in Sections 4 and 5, several elements of our approach (e.g., profiling the energy usage of a bytecode, assessing infrastructure energy costs) rely on the ability to obtain accurate energy measurements "off line".

Second, we assume that interpreter-based JVMs, such as Sun Microsystems' KVM [9] and JamVM [5], are used. These JVMs have been developed for resource-constrained platforms, and require much less memory than "just-in-time" (JIT) compilation-based JVMs. If a JIT-based JVM is used, the energy cost for translating a bytecode into native code "on the fly" would need to be added into Equation 2 since the JIT compilation itself happens while a Java application is being executed. We are currently investigating how our framework can be extended to JIT-based JVMs.

Third, we assume that the systems to which our framework is applicable will be implemented in "core" Java. In other words, apart from the JVM, we currently do not take into account the effects on energy consumption of any other middleware platform. While this does not prevent our framework from being applied on a very large number of existing Java applications, clearly in the future we will have to extend this work to include other middleware platforms.

Finally, we assume that the target network environment is a (W)LAN that consists of dedicated routers (e.g., wireless access points) and stationary or mobile hosts. This is representative of a majority of systems that rely on wireless connectivity and battery power today. In the case of mobile hosts, we assume that each host associates itself with an access point within its direct communication range and communicates with other hosts via dedicated access points. In this setting, there could be a hand-off overhead when mobile hosts move and change their associated access points. However, it is not the software system that causes this type of energy overhead, but rather the movement of the host (or user). Therefore, we currently do not consider these types of overhead in our framework. Note that in order to expand this work to a wireless *ad-hoc* network environment, we also need to consider the energy overhead of routing event messages by each host. This type of energy overhead can be accounted for by extending the infrastructure aspect of our framework. We plan to investigate this issue as part of our future work.

4 Energy Consumption Estimation

In this section, we discuss the framework's application for estimating a software system's energy cost at both during system construction-time and runtime.

4.1 Construction-Time Estimation

For construction-time estimation, we first need to characterize the computational energy cost of each component on its candidate hosts. To this end, we have identified three different types of component interfaces:

I. An interface (e.g., a date component's `setCurrentTime`) that requires the same amount of computation regardless of its input parameters.
II. An interface (e.g., a data compression component's `compress`) whose input size is proportional to the amount of computation required.
III. An interface (e.g., DBMS engine's `query`) whose input parameters have no direct relationship to the amount of computation required.

For a type I interface, we need to profile the number of bytecodes, native methods, and monitor operations only once for an arbitrary input. We can then calculate its energy consumption from Equation 2.

For type II interfaces, we generate a set of random inputs, profile the number of bytecodes, native methods, and monitor operations for each input, and then calculate its energy cost from Equation 2. However, the set of generated inputs does not show the complete energy behavior of a type II interface. To characterize the energy behavior of a type II interface for any arbitrary input, we employ multiple regression [1], a method of estimating the expected value of an output variable given the values of a set of related input variables. By running multiple regression on a sample set of input variables' values (i.e., each generated input for a type II interface) and the corresponding output value (the calculated energy cost), it is possible to construct an equation that estimates the relationship between the input variables and the output value.

Interfaces of type III present a challenge as there is no direct relationship between an interface's input and the amount of computation required, yet a lot of interface implementations fall in this category. For type III interfaces with a set of *finite* execution paths, we use symbolic execution [8], a program analysis technique that allows using symbolic values for input parameters to explore program execution paths. We leverage previous research [7], which has suggested a generalized symbolic execution approach for generating test inputs covering all the execution paths, and use these inputs for invoking a type III interface. We then profile the number of bytecodes, native methods, and monitor operations for each input, estimate its energy cost from Equation 2, and finally calculate the interface's average energy cost by dividing the total energy cost by the number of generated inputs.

The above approach works only for interfaces with finite execution paths, and is infeasible for interfaces whose implementations have *infinite* execution paths, such as a DBMS engine. We use an approximation for such interfaces: we automatically invoke the interface with a large set of random inputs, calculate the energy cost of the interface for each input via Equation 2, and finally calculate the average energy consumption of the interface by dividing the total consumption by the number of random

inputs. This approach will clearly not always give a representative estimate of the interface's actual energy consumption. Closer approximations can be obtained if an interface's expected runtime context is known (e.g., expected inputs, their frequencies, possible system states, and so on). As we will detail in Sections 4.2, we can refine our estimates for type III interfaces by monitoring the actual amount of computation required at runtime.

To estimate the communication energy cost of an interface, we rely on domain knowledge (e.g., the known types of input parameters and return values) to predict the average size of messages exchanged due to an interface's invocation. Using this data we approximate the communication energy cost of interface invocation via Equation 4. Finally, based on the computational and communication energy costs of interfaces, we estimate the overall energy cost of a component on its candidate host(s) using Equations 1, 3, and 5.

Before estimating the entire distributed system's energy cost, we also need to determine the infrastructure's energy overhead, which depends on the deployment of the software (e.g., the number of components executing simultaneously on each host). Unless the deployment of the system's components on its hosts is fixed *a priori*, the component-level energy estimates can help us determine an initial deployment that satisfies the system's energy requirements (e.g., to avoid overloading an energy-constrained device). Once an initial deployment is determined, from Equation 6 we estimate the infrastructure's energy cost. We do so by executing all the components on their target hosts *simultaneously*, with the same sets of inputs that were used in characterizing the energy cost of each individual component. Finally, we determine the system's overall energy cost via Equation 7.

4.2 Runtime Estimation

Many systems for which energy consumption is a significant concern are long-lived, dynamically adaptable, and mobile. An effective energy cost framework should account for changes in the runtime environment, or due to the system's adaptations. Below we discuss our approach to refining the construction-time estimates after the initial deployment.

The amount of computation associated with a type I interface is constant regardless of its input parameters. If the sizes of the inputs to a type II interface significantly differ from construction-time estimates, new estimates can be calculated efficiently and accurately from its energy equation generated by multiple regression. Recall from Section 4.1 that for type III interfaces our construction-time estimates may be inaccurate as we may not be able to predict the frequency of invocation or the frequency of the execution paths taken (e.g., the exception handling code). Therefore, to refine a type III interface's construction-time estimates, the actual amount of runtime computation (i.e., number of bytecodes, native methods, and monitor operations) must be monitored. In Section 5.4 we present an efficient way of monitoring these parameters.

For the communication cost of each component, by monitoring the sizes of messages exchanged over network links, their effects on each interface's communication cost can be determined, and a component's energy cost can be updated automatically.

Finally, the fact that the frequency at which interfaces are invoked may vary signif-

icantly from what was predicted at construction-time, and the fact that the system may be adapted at runtime, may result in inaccurate construction-time infrastructure energy estimates. Therefore, the GC thread execution time and the number of implicit OS routines invoked at runtime must also be monitored. We discuss the overhead of this monitoring in detail in Section 5.4. Based on the refined estimates of each interface's computational and communication costs, and of the infrastructure's energy overhead, we can improve (possibly automatically) our construction-time estimates of distributed systems at runtime.

5 Evaluation Strategy

This section describes our evaluation environment, the tools on which we have relied, and the energy measurement and monitoring approaches we have used.

5.1 Experimental Setup

To evaluate the accuracy of our estimates, we need to know the *actual* energy consumption of a software component or system. To this end, we used a digital multimeter, which measures the factors influencing the energy consumption of a device: voltage and current. Since the input voltage is fixed in our experiments, the energy consumption can be measured based on the current variations going from the energy source to the device.

Figure 2 shows our experimental environment setup that included a Compaq iPAQ 3800 handheld device running Linux and Kaffe 1.1.5 JVM [6], with an external 5V DC power supply, a 206MHz Intel StrongARM processor, 64MB memory, and 11Mbps 802.11b compatible wireless PCMCIA card. We also used an HP 3458-a digital multimeter. For measuring the current drawn by the iPAQ, we connected it to the multimeter, which was configured to take current samples at a high frequency. A data collection computer controlled the multimeter and read the current samples from it.

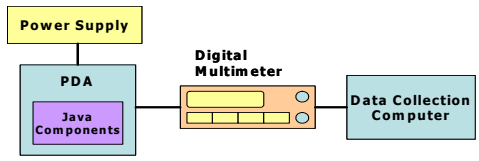


Figure 2. Experimental setup.

5.2 Selecting Java Components

We have selected a large number of Java components with various characteristics for evaluating our framework. They can be categorized as follows: 1) *Computation-intensive components* that require a large number of CPU operations. (e.g., encryption/decryption, data compression); 2) *Memory-intensive components* that require large segments of memory. (e.g., Database components); 3) *Communication-intensive components* that interact frequently with other components over a network (e.g., FTP component).

For illustration, Table 1 shows a cross-section of the Java components used in our evaluation. These components vary in size and complexity (`HSQLDB` is the largest, with more than 50,000 SLOC, while `Jess` is somewhat smaller, with approximately 40,000 SLOC). The source code of `Jess`, `HSQLDB`, and `IDEA` components can be found at Source Forge [18], while the source code of the other components from Table

1 was obtained from Source Bank [17].

5.3 Measurement

Prior to the deployment, we first need to measure the energy cost on a target platform of each bytecode, native method, monitor operation, and implicit OS routine, as well as the average consumption rate during garbage collection. For each bytecode we generate a Java class file that executes that bytecode 1000 times. We also create a skeleton Java class with no functionality, which is used to measure the energy consumption overhead of executing a class file. We use the setup discussed in Section 5.1 for measuring the actual energy cost of executing both class files. We then subtract the energy overhead *E1* of running the skeleton class file from the energy cost *E2* of the class file with the profiled bytecode. By dividing the result by 1000, we get the average energy consumption of executing the bytecode. Similarly, for measuring the energy consumption of each native method, we generate a class file invoking the native method and measure its actual energy consumption *E3*. Note that when JVM executes this class file, several bytecodes are also executed. Therefore, to get the energy cost of a native method, we subtract (*E1* + energy cost of the bytecodes) from *E3*. For a monitor operation, we generate a class file invoking a method that should be synchronized among multiple threads, and measure its energy consumption *E4*. Since several bytecodes are also executed during the invocation, we can get the energy cost of a monitor operation by subtracting (*E1* + energy cost of the bytecodes) from *E4*.

Component	Description
SHA, MD5, IDEA	Components that encrypt or decrypt messages by using SHA, MD5, and IDEA algorithms
Median filter	Component that creates a new image by applying a median filter
LZW	Data compression/decompression component implementing the LZW algorithm
Sort	Quicksort component
Jess	Java Expert Shell System based on NASA's CLIPS expert shell system
DB	HSQLDB, a Java-based database engine
Shortest Path	Component that finds the shortest path tree with the source location as root
AVL, Linked list	Data structure components that implement an AVL tree and a linked list

Table 1: A cross-section of Java components used in evaluation.

To measure the energy cost of implicit OS routines, we employ the approach suggested by Tan et al. [19], which captures the energy consumption behavior of embedded operating systems. This allows us to determine the energy cost of major implicit OS routine calls, such as context switching, paging, and process scheduling. Due to space constraints we cannot provide the details of this approach; we point the interested readers to [19]. Finally, for getting the average energy consumption rate of the GC thread, we execute over a given period of time a simple Java class file that creates a large number of "dummy" objects, and measure the average energy consumption rate during the garbage collection.

5.4 Monitoring

Since we need to monitor the numbers of bytecodes, native methods, monitor operations, and implicit OS routines, as well as the GC thread execution time, we instrumented the Kaffe 1.1.5 JVM to provide the required monitoring data. Since the monitoring activity itself also consumes energy, we had to ensure that our monitoring mechanism is as light-weight as possible. To this end, we modified Kaffe's source

code by adding: 1) an integer array of size 256 for counting the number of times each bytecode type is executed; 2) integer counters for recording the number of times the different native methods are invoked; and 3) an integer counter for recording the number of monitor operations executed.

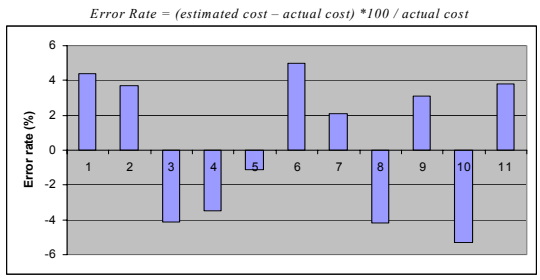
As mentioned earlier, this monitoring is only used for type III interfaces. We also added a timer to Kaffe's GC module to keep track of its total execution time. This timer has a small overhead equivalent to two system calls (for getting the times at the beginning and at the end of the GC thread's execution). For the number of implicit OS routines, we simply used the facilities provided by the OS. Since both Linux and Windows by default store the number of implicit OS routines executed in each process's Process Control Block, we did not introduce any additional overhead. We have measured the energy overhead due to these monitoring activities for the worst case (i.e., type III interfaces). The average energy overhead compared with the energy cost without any monitoring was 3.8%. Note that this overhead is transient: engineers can choose to monitor systems during specific time periods only (e.g., whenever any changes occur or are anticipated in the system or its usage).

6 Evaluation Results

In this section, we present the results of evaluating our framework.

6.1 Computational Energy Cost

To validate our computational energy model, we compare the values calculated from Equation 2 with actual energy costs. All actual energy costs have been calculated by subtracting the infrastructure energy overhead (Equation 6) from the energy consumption measured by the digital multimeter. As an illustration, Figure 3 shows the results of one series of executions for the components of Table 1. In this case, for each component we executed each of its interfaces 20 times with different input parameter values, and averaged the discrepancies between the estimated and actual costs (referred to as "error rate" below). The results show that our estimates fall within 5% of the actual energy costs. These results are also corroborated by additional experiments performed on these as well as a large number of other Java components [17,18].



1	2	3	4	5	6	7	8	9	10	11
SHA	MD5	IDEA	Median Filter	LZW	Sort	Jess	DB	Shortest path	AVL	Linked List

Figure 3. Error rates for the components in Table 1.

In addition to executing components of Table 1 in isolation, we have run these components simultaneously in different sample applications. Recall that, since each component is running in a separate JVM process, the energy overhead due to implicit OS routines is higher when multiple components are running simultaneously than

when each is running in isolation. Figure 4 shows the error rates of our computational energy model as the number of simultaneously running components increases. The experimental results show that, despite the increased infrastructure overhead, our estimates usually fall within 4% of the actual energy costs.

As discussed in Section 4.1, multiple regression can be used for characterizing the energy cost of invoking type II interfaces. For this we used a tool called DataFit. In measurements we conducted on close to 50 different type II interfaces, our estimates of their energy cost have been within 5% of the actual energy costs. As an illustration, Figure 5 shows the graph generated by DataFit for the `find` interface of the `Shortest Path` component, using 20 sets of sample values for `find`'s input parameters (x_1 and x_2), and the resulting energy costs (y) estimated by Equation 3. Several actual energy costs are shown for illustration as the discrete points on the graph.

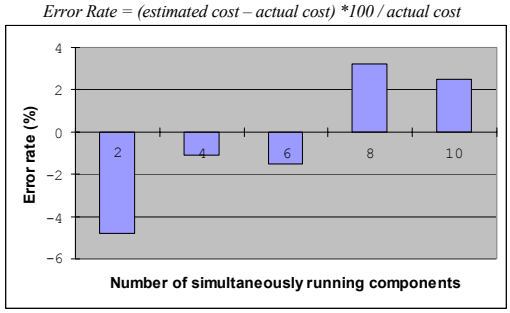
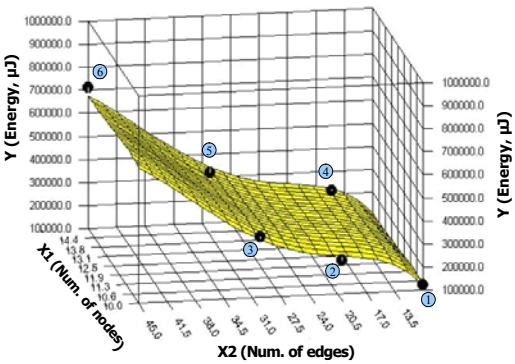


Figure 4. Error rates with respect to the number of simultaneously running components.

For estimating the energy consumption of type III interfaces, as discussed previously we generated a set of random inputs, estimated the energy cost of invoking each interface with the inputs using Equation 3, and calculated its average energy consumption. Figure 6 compares the average energy consumption of each interface for the DB and Jess components calculated using our framework with the interface's actual average energy consumption. The results show that our estimates are within 5% of the actual average energy costs. Recall that these design-time energy estimates can be refined at runtime by monitoring the numbers of bytecodes, native methods, and monitor operations executed. For example, for a scenario that will be detailed in Section 6.3, we refined the construction-time energy estimate for the DB query interface at runtime, reducing the error rate to under 2.5%.



	Point 1	Point 2	Point 3	Point 4	Point 5	Point 6
Actual (µJ)	122688	245395	364891	215760	316622	710709
Estimated (µJ)	119767	261905	351300	224201	341570	677319

Figure 5. Multiple regression for the *find* interface of the *Shortest Path* component.

6.2 Communication Energy Cost

For evaluating the communication energy cost, we use a wireless router for the iPAQ to communicate with an IBM ThinkPad X22 laptop via a UDP socket implementation over a dedicated wireless network. Recall from Section 3.2 that several parameters (*tEC*, *rEC*, *tS*, and *rS*) from Equation 4 are host-specific. To quantify these parameters for the iPAQ, we created two Java programs that exchange messages via UDP sockets, and executed them on the iPAQ and the laptop. We then used the digital multimeter to measure the actual energy cost *E* on the iPAQ as a result of transmitting and receiving a sample set of messages of various sizes to/from the laptop. Since several bytecodes and native methods (e.g., `java.net.SocketInputStream's read()` method) are executed during the program execution on the iPAQ, we subtract their energy costs from *E* to get the energy consumption of a wireless interface card on the iPAQ. Based on these results, we used multiple regression to find equations that capture the relationship between the input (size of the transmitted or received data *x*) and the output (actual energy consumption *y* of a wireless interface card on the iPAQ):

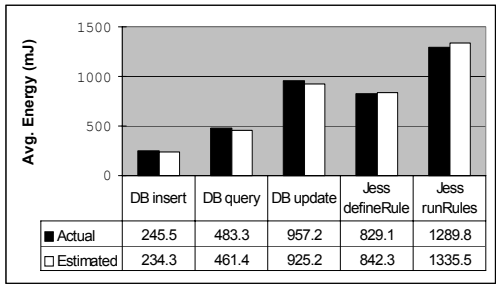


Figure 6. Accuracy of the framework for type III interface of *DB* and *Jess* components.

$$y_t(mJ) = 4.0131 * x_t(KB) + 3.1958 \qquad \textbf{Eq. 8}$$

$$y_r(mJ) = 3.9735 * x_r(KB) + 5.3229 \qquad \textbf{Eq. 9}$$

We used the generated equations to quantify the host-specific parameters in Equation 4. For example, the size of transmitted data x_t in Equation 8 represents *tEvtSize* in Equation 4. The constant energy cost of 3.1958 represents the parameter *tS* in Equation 4, which is independent of the size of transmitted data. The variables *tEC* is captured by the constant factor 4.0131. Figure 7 shows two graphs plotted for Equations 9 and 10, which represent the framework's estimates. As shown, the estimates, which are depicted by the discrete points are within 3% of the actual energy costs.

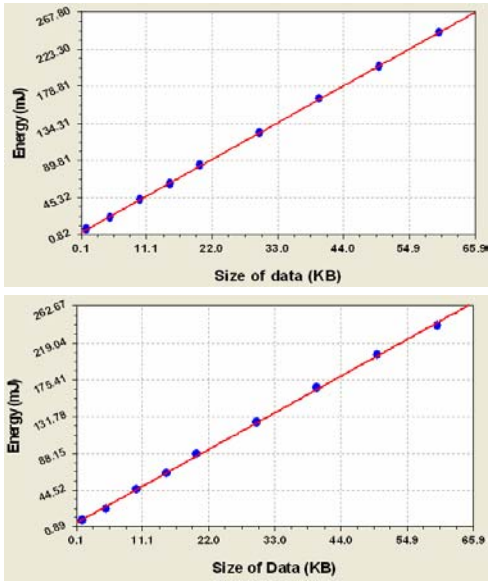


Figure 7. Transmission (top) and receipt (bottom) energy estimation on an iPAQ.

6.3 Overall Energy Cost

We have evaluated our framework

over a large number of applications. Figure 8 shows one example such application deployed across three iPAQ hosts. These iPAQ devices communicate with each other via a wireless router. Each software component interacts with the other components via a UDP socket. A line between two components (e.g., `IDEA` and `FTP Client` on host A) represents an interaction path between them. The `FTP Client` and `Server` components used in our evaluation are UDP-based implementations of a general purpose FTP. We have used several execution scenarios in this particular system. For example, `DB Client` component on host A may invoke the `query` interface (i.e., type III interface) of the remote `DB Server` on host B; in response, `DB Server` calculates the results of the query, and then invokes `IDEA`'s `encrypt` interface (i.e., type II interface) and returns the encrypted results to `DB Client`; finally, `DB Client` invokes the `decrypt` interface (i.e., type II interface) of its collocated `IDEA` component to get the results.

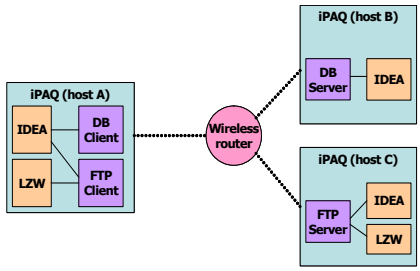


Figure 8. A distributed Java-based system comprising three hosts.

We executed the above software system by varying the frequencies and sizes of exchanged messages, measured the system's overall energy cost, and compared it with our framework's runtime estimates. As shown in Figure 9, our estimates fall within 5% of the actual costs regardless of interaction frequencies and the average size of messages. In addition, we have evaluated our framework for a large number of additional distributed applications, increasing the numbers of components and hosts [14], and had similar results as shown in Figure 10.

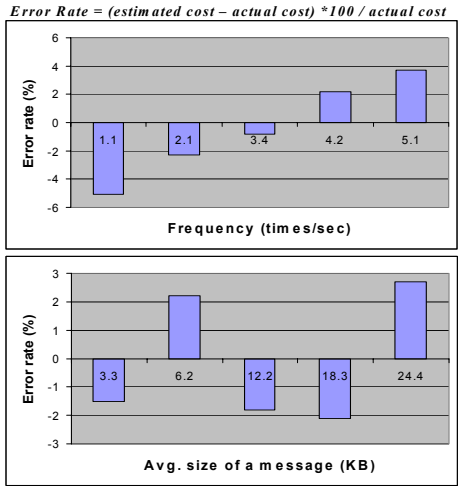


Figure 9. The framework's error rates with respect to the interaction frequency (top) and the average size of a message (bottom).

7 Conclusion

We have presented a framework for estimating the energy cost of Java-based software systems. Our primary objective in devising the framework has been to enable an engineer to make informed decisions, such that the system's energy consumption is reduced and the lifetime of the system's critical services increases. In a large number of distributed application scenarios the framework has shown very good precision, giving results that have been within 5% (and often less) of the actually measured power losses incurred by executing the software. We consider the development and evaluation of the framework to be a critical first step in pursuing several avenues of further work. As part of our future work, we plan to investigate the applications of

the framework to various types of architectural decisions that could improve a system's energy usage, such as off-loading of software components, adapting components, modifying communication protocols, and so on.

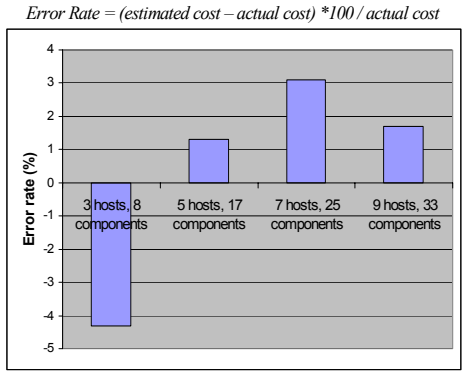


Figure 10. Error rates of the framework with respect to the numbers of hosts and comps.

8 Acknowledgements

This material is based upon work supported partially by the National Science Foundation under Grant Numbers 0312780 and 0820222. Effort also partially supported by the Bosch Research and Technology Center.

9 References

1. P. D. Allison. Multiple regression. Pine Forge Press. 1999.
2. W. Drytkiewicz, et al. pREST: a REST-based protocol for pervasive systems. *IEEE International Conference on Mobile Adhoc and Sensor Systems*, 2004.
3. K. I. Farkas et. al. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. *ACM SIGMETRICS*, New York, NY, 2000.
4. L. M. Feeney et al. Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. *IEEE INFOCOM*, Anchorage, AL, 2001.
5. JamVM 1.3.2. http://jamvm.sourceforge.net/, 2006.
6. Kaffe 1.1.5. http://www.kaffe.org/, 2005.
7. S. Khurshid et al. Generalized Symbolic Execution for Model Checking and Testing. *Int'l Conf. on Tools and Algorithms for Construction and Analysis of Systems*, Warsaw, Poland, April 2003.
8. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, vo.19, no. 7, 1976.
9. KVM. http://java.sun.com/products/cldc/wp/, 2005.
10. J. J. Labrosse. *MicroC/OS-II: The Real-Time Kernel.* CMP Books, 2002.
11. S. Lafond, et al. An Energy Consumption Model for An Embedded Java Virtual Machine. *Virtual Machine Research and Technology Symposium*, 2006.
12. C. Seo, et al. An Energy Consumption Framework for Distributed Java-Based Systems. *Int'l Conf. on Automated Software Engineering*, Atlanta, Georgia, Nov. 2007.
13. C. Seo, et al. Estimating the Energy Consumption in Pervasive Java-Based Systems. Int'l Conf. on *Pervasive Computing and Communication*, Hong Kong, March 2008.
14. C. Seo. Prediction of Energy Consumption Behavior in Component-Based Distributed Systems. Ph.D. Dissertation, University of Southern California, April 2008.
15. H. Singh, et al. Energy Consumption of TCP in Ad Hoc Networks. *Wireless Networks,* vol. 10, no. 5, 2004.
16. A. Sinha, et al. JouleTrack - A Web Based Tool for Software Energy Profiling. *Design Automation Conference,* 2001.
17. SourceBank. http://archive.devx.com/sourcebank/
18. sourceForge.net. http://sourceforge.net/
19. T. K. Tan et al. Energy macromodeling of embedded operating systems. *ACM Trans. on Embedded Comp. Systems*, 2005.
20. UPnP Device Architecture, http://www.upnp.org/, 2007.
21. R. Xu et al. Impact of Data Compression on Energy Consumption of Wireless-Networked Handheld Devices, *Int'l Conf. on Distributed Computing Systems*, Rhode Island, 2003.