An Energy Consumption Framework for Distributed Java-Based Systems

Chiyoung Seo Sam Malek Nenad Medvidovic

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

Abstract

In this paper we present a framework for estimating the energy consumption of Java-based software systems. Our primary objective 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 extended. Our framework explicitly takes a component-based perspective. It allows the engineer to estimate a system's energy consumption prior to deployment and refine it at runtime. In a large number of distributed application scenarios, the framework has given results that are within 5% of the actually measured power losses incurred by executing the software. Our work to date has also highlighted a number of future enhancements.

1. Introduction

Modern software systems are predominantly distributed, dynamic, and mobile. 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 may comprise 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.

Consider the scenario depicted in Figure 1, in which seven software components are deployed on four battery-powered hardware hosts, and are communicating over the network. Without concerning ourselves with any other details of this application, we can ask a number of questions about its energy consumption. For example, does the location of a given component (e.g., c_4) impact its energy consumption rate? Would redeploying a component (e.g., c_4) from one host (e.g., H_4) to another (e.g., H_2) change the system's, or a given system service's, life span? Can we compare the likely energy consumption profiles of two or more candidate deployments? What is the best deployment for the system with respect to energy consumption?

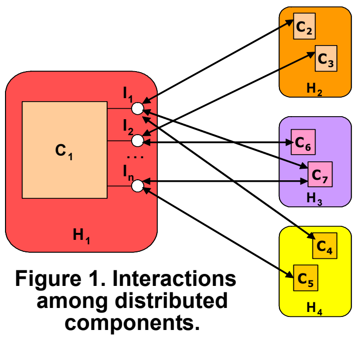


Figure 1. Interactions among distributed components.

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

To this end, in this paper we define and empirically evaluate a framework that estimates the power consumption of distributed software systems implemented in Java. 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 simplifying assumptions possibly not afforded by other mainstream languages. One novel aspect of our framework is its component-based development perspective, where the power consumption of a component (e.g., a Java class or cluster of classes) is modeled and calculated as a function of its public interface. In turn, this perspective also simplifies the model of (inter-component) communication costs. Another contribution of this work is its ability to efficiently adjust energy consumption estimates at runtime, based on monitoring the changes in a small number of easily tracked system parameters (e.g., network bandwidth, size of data exchanged over the network, each interface's invocation frequency, etc.).

We have evaluated our framework for precision on a large number of distributed Java applications, by comparing its estimates against actual electrical current measurements. Our results suggest that the framework is always able to estimate the power consumed by a distributed Java system to within 5% of the actual consumption.

A very narrow view of the work described in this paper is that it can help an engineer determine when her Java-based system will run out of battery power. A broader, and in our view more appropriate, view is that this work is the necessary first step in the direction of

1. improving our understanding of the existing strategies for minimizing the power consumption of a distributed software system (e.g., component off-loading, redeployment, degraded mode operation), and the trade-offs among them,
2. improving those existing strategies, and
3. devising and quantifying new such strategies.

The remainder of the paper is organized as follows: Section 2 presents the related research in the energy estimation and measurement areas. We then introduce our energy estimation framework in Section 3 and detail how it is applied to component-based Java systems in Section 4. We present our evaluation strategy in Section 5 and evaluation results in Section 6. Finally, Section 7 concludes the paper.

2. Related Work

Previous research has suggested energy consumption models at the level of CPU instructions [22,23]. However, an engineer would not be able to use these models directly to estimate the energy consumption of a distributed component-based software system: they do not take into account system elements other than CPU (e.g., main memory) or the communication energy cost incurred by remote interactions among software components.

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. A JVM generally has five stages during its life cycle [15]: *start*, *initialize*, *load main class, interpreter loop*, and *exit*. Farkas et al. have discussed different JVMs' design trade-offs in each stage and measured their energy consumption. The energy consumed at the *interpreter loop* stage corresponds to the actual energy required to execute a Java application, while the energy consumed by the other stages is constant [15]. Lafond et al. [15] have measured the energy consumption of each stage, and showed that the energy required for memory accesses usually accounts for 70% of the total energy consumed. Vijay et al. [24] have discussed the characteristics of the energy consumption by cache and main memory when executing the SPEC JVM98 benchmarks [19] in the just-in-time (JIT) and interpreter modes on the Sun Labs Virtual Machine for Research, EVM [20]. However, none of these studies suggest a model that can be used for estimating the energy consumption of a distributed Java-based system.

Recently, researchers have attempted to characterize the energy consumption of the Transmission Control Protocol (TCP) [16]. Singh et al. [16] 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 supports for congestion control, retransmission, error recovery, and so forth.

Several studies [4,5] 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 and inversely proportional to the available bandwidth. We use these experimental results as a basis for defining a component's communication energy cost.

3. Overview of the Energy Cost Framework

We model a distributed software system's energy consumption at the level of its components. 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 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 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.[1] 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 highlights component c_1 on host H_1, c_1's provided interfaces, and the invocation of those interfaces by remote components. Given the energy consumption *iCompEC* resulting from invoking an interface

1. Note that we use the them "interface" in a broader sense than the programming language-level construct supported by Java. Our use of the term is consistent with component-based software engineering literature.

I_i, and the total number b_i of invocations for interface I_i, we can calculate the overall energy consumption of component c_1 with n interfaces (expressed in *Joule or J*) 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 multiple threads via an internal implementation of a *monitor*.

Each Java statement maps to a specific sequence of bytecodes, native methods, and/or monitor operations; consequently, each Java method, class, and application maps to a (much longer) such sequence. Therefore, based on the 256 bytecodes, *m* native methods, and *monitor* operations that are available on a given JVM, we can estimate the energy consumption *iCompEC*(I_i,*j*) of invoking an interface on that JVM 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 configurations, JVMs, and OSs, their respective energy values for bEC_k, fEC_l, and *mEC* will likely be different. We will explain how these values can be obtained for an actual host in Section 5.

3.2. Communication 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 CPU, 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 amounts of energy. Given the communication energy consumption *iCommEC* due to invoking an interface I_i, and the total number b_i of invocations for that interface I_i, we can calculate the overall communication energy consumption of a component c_1 with *n* interfaces (expressed in *Joule*) as follows:

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

In this equation, *iCommEC*(I_i,*j*), the communication energy cost incurred by the j_{th} invocation of I_i, depends on the amount of data transmitted or received during the invocation and might 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. As discussed in Section 2, UDP is a light-weight protocol (e.g., it provides no congestion control, retransmission, and error recovery mechanisms), and is more prevalent than TCP in embedded and resource-constrained computing domains. Previous research [4,5] has shown that the actual energy consumption of wireless communication is directly proportional to the amount of transmitted and received data, and inversely proportional to the bandwidth between two hosts. Based on this, we quantify the communication energy consumption of 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:

$$iCommEC(I_i, j) = \left(\frac{tEvtSize_{c_1,c_2}}{tBw_{H_1,H_2}} \times tEC_{H_1} + tS_{H_1} \right) + \left(\frac{rEvtSize_{c_1,c_2}}{rBw_{H_1,H_2}} \times rEC_{H_1} + rS_{H_1} \right) \qquad \textbf{\textit{Eq. 4}}$$

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. *tBw* and *rBw* are the actual UDP bandwidths (*KB*/*sec*) used by H_1 for transmission and receipt. tEC_{H1} and rEC_{H1} are the host's energy consumption rates (*Joule*/*sec*) while it transmits and receives data. Finally, tS_{H1} and rS_{H1} represent constant energy overheads associated with device state changes and channel acquisition.[2]

In Equation 4, the energy values of *tEC*, *rEC*, *tS*, *rS* are constant and platform-specific. The system parameters that

2. When there is no network traffic for a certain period of time, a wireless interface card changes its state from *idle* to *sleep*. When a new packet is ready for transmission or reception, the wireless interface card changes its state from *sleep* to *active* and acquires a channel for communication. These state changes and channel acquisition activities consume energy.

need to be monitored on each host are only the sizes of messages exchanged (*tEvtSize* and *rEvtSize*, which include the overhead of UDP protocol headers) and the transmission and receipt bandwidths (*tBW* and *rBW*). 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 may be determined as follows:

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

However, there are additional energy costs for executing a Java component incurred by JVM's garbage collection (GC) and implicit OS routines.

During garbage collection, all threads within the JVM process are suspended and the GC thread takes over the execution control. We estimate the energy consumption resulting from GC 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 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 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 [21]. 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 [21].

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 consumption 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 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, determining an appropriate initial system deployment) rely on the ability to obtain accurate energy measurements "off line".

Second, we assume that interpreter-based JVMs, such as Sun Microsystems' KVM [14] and JamVM [8], 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.

We assume that the target network environment is a (W)LAN that consists of dedicated routers (e.g., wireless access points) and either 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 environment, there could be a hand-off overhead as a result of mobile hosts moving and changing 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 (recall Section 3.3). We plan to investigate this issue as part of our future work.

Finally, the energy estimates of a small subset of native methods depend on a given platform's hardware configuration, such as the LCD brightness and speaker volume. An underlying assumption in this paper is that the configuration of target hardware platforms for which native methods are profiled at construction-time does not change significantly at runtime. However, we could trivially account for this by profiling the native methods that depend on hardware configurations for each variation point. For example, we could profile the energy consumption of a native method that is impacted by LCD's refresh rate (e.g., 60Hz vs. 90Hz).

4. Energy Consumption Estimation

In this section, we discuss how our framework can be used for estimating a distributed software system's energy consumption at the level of its components both during system construction-time and during runtime.

4.1. Construction-Time Estimation

In order to estimate a distributed system's energy consumption during construction-time, we first need to characterize the computational energy consumption 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 interfaces of type II, we first generate a set of random inputs, profile the number of bytecodes, native methods, and monitor operations for each input, and then calculate its energy consumption 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]. Multiple regression is 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 (in our case, each generated input for a type II interface) and the corresponding output value (energy consumption calculated from Equation 2), it is possible to construct an equation that estimates the relationship between the input variables and the output value. In Figure 4 (further discussed in Section 6), we show an example of applying multiple regression to the *Shortest Path* component that finds the shortest path tree with the source location as a root.

Interfaces of type III present a challenge because there is no direct relationship between an interface's input parameters and the amount of computation required, yet a lot of interface implementations fall in this category (e.g., many Java methods containing loops and branches). To characterize the energy behavior of type III interfaces with a set of *finite* execution paths, we use symbolic execution [13], a well known program analysis technique that allows using symbolic values for input parameters to explore program execution paths. We leverage previous research [12], 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 consumption from Equation 2, and finally calculate the interface's average energy consumption by dividing the total energy consumption 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 invoke the interface with a large set of random inputs, calculate the energy consumption 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: if the random inputs result in execution paths that are shorter (or longer) than the actual paths executed at runtime, the interface's energy consumption will be underestimated (or overestimated). Closer approximations can be obtained if an interface's expected runtime context is known (e.g., expected inputs, possible system states, values of certain variables, and so on). As we will detail in Sections 4.2 and 5.4, we can also refine our construction-time energy estimates for type III interfaces by monitoring the actual amount of computation required at runtime.

To estimate the communication energy consumption of each interface, based on domain knowledge, types of input parameters and return values, and the target hardware environment, we estimate the average size of messages exchanged and the maximum available bandwidth for communication. Using this data we can approximate the communication energy cost of interface invocation via Equation 4. Finally, based on these analyses for computational and communication energy costs of each interface, we can estimate the overall energy consumption of a component on its candidate host(s) using Equations 1, 3, and 5.

Before estimating the entire distributed system's energy consumption, 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 can 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 consumption of each individual component. Finally, we determine the distributed system's overall energy consumption 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 for such systems should account for variations in the energy consumption due to changes in the runtime environment, or due to the system's adaptations. In this section, we discuss our approach to refining our construction-time energy estimates of a system after its initial deployment.

The initial deployment of a software system onto the target hosts may be based on energy cost estimates that are made in the manner discussed above. However, many aspects of the system, such as the frequency with which interfaces of a component are actually invoked, may have not been accurately estimated. The construction-time estimates are based on a system engineer's guesses or domain knowledge. For a large class of systems, such as the space exploration system that was mentioned in Section 1, most of the runtime properties can be predicted fairly accurately at construction-time: available bandwidth between a rover and a satellite can be estimated based on the orbital position of the satellite, frequency of each interface's invocation can be estimated for a duration of time based on the mission or task, and so on. On the other hand, in many systems, construction-time estimates may differ from the system's actual use and thus its actual energy consumption. In such situations, more accurate estimates are only possible at runtime as discussed below.

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 both the invocation frequency and the most frequently executed paths. Therefore, to refine a type III interfaces' construction-time estimates, we need to monitor the actual amount of computation at runtime (i.e., number of bytecodes, native methods, and monitor operations). In Section 5.4 we present an efficient way of monitoring these parameters.

The communication cost of each component can be refined in an analogous manner, based on the appropriate system parameters (recall Equation 4). For example, by monitoring the variations in the available bandwidth and the sizes of messages exchanged over the network links, we can determine their effects on each interface's communication cost, and thus update a component's overall energy cost.

Finally, the fact that the frequency at which interfaces are involved may vary significantly 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, we also need to monitor the GC thread execution time and the number of implicit OS routines invoked at runtime. 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 our construction-time energy estimates of the distributed system at runtime.

5. Evaluation Strategy

This section describes our evaluation environment and the tools on which we relied. We also describe the strategy we employed in selecting Java components for the evaluation. We round out the section by detailing the energy measurement and monitoring approaches we have used.

5.1. Experimental Setup

In order to evaluate the accuracy of our framework's 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. To measure the actual energy consumption of running software on a device, we first monitor the constant amount of current drawn by the device when it is idle. When a software component executes, the current increases. The cumulative difference in these two current levels over the execution time represents the actual energy consumption of executing the software.

Figure 2 shows our experimental environment setup that included a Compaq iPAQ 3800 handheld device running Linux and Kaffe 1.1.5 JVM [11], 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 digital multimeter and read the current samples from it. This basic setup is slightly varied depending on the focus of our measurements, as discussed below.

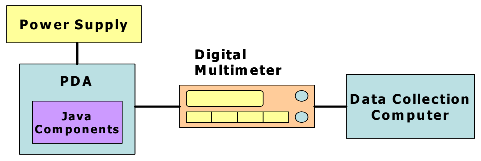


Figure 2. Experimental setup.

5.2. Selecting Java Components

We have selected a large number of Java components with various characteristics for evaluating our framework. They can be classified into the following three categories:

- *Computation-intensive components* that require a large number of CPU operations. We have used components that perform data encryption/decryption, image processing, data compression, sorting components, and so on.
- *Memory-intensive components* that require large segments of memory at runtime. Database and various data structure components have been selected for this type.
- *Communication-intensive components* that interact frequently with other components over a network. Database and FTP components have been chosen for this type.

Table 1:
Java components used in evaluation.

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

For illustration, Table 1 shows a cross-section of the Java components used in our evaluation. These components vary in size and complexity (`HSQLDB` [6] is the largest, with more than 50,000 SLOC, while `Jess` [10] is somewhat smaller, with approximately 40,000 SLOC). The source code of `Jess`, `HSQLDB`, and `IDEA` components can be found at Jess [10], Source Forge [18], and Java Grande Forum [9] respectively, while the source code of the other components shown in Table 1 was obtained from Source Bank [17].

5.3. Measurement

Prior to system deployment, we first need to measure the energy consumption 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 (GC). 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*.

To measure the energy consumption of implicit OS routines, we employ the approach suggested by Tan et al. [21], which captures the energy consumption behavior of embedded operating systems. Using this approach we are able 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 interested readers to [21] for more information. 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" Java objects, and measure the average energy consumption rate during the garbage collection phase.

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 facilities. 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 in Section 4.2, this type of runtime monitoring is only used for type III interfaces. We also added a global 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. Finally, for measuring the available bandwidth on each host we used *Iperf* [7], a tool for efficient measurements of network bandwidth variations. 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 consumption without any monitoring was 3.8%. We should note that this overhead is transient: engineers can choose to monitor systems during specific time periods only (e.g., whenever any changes occur in the system or in its usage).

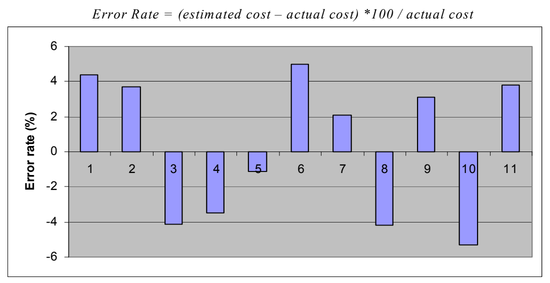
6. Evaluation Results

In this section, we present the results of evaluating our framework. Specifically, we assess its accuracy in estimating the energy cost of distributed Java-based systems.

6.1. Computational Energy Cost

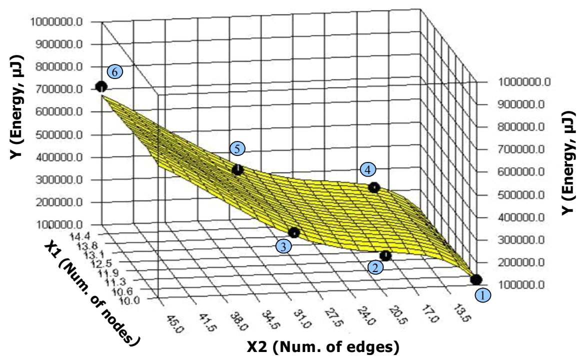
To validate our computational energy model, we compare the values calculated from Equation 2 with the actual energy costs. In this section, all of the actual energy costs have been calculated by subtracting the infrastructure energy overhead (Equation 6) from the energy consumption measured by the digital multimeter. As an illustration, Figure 3 shows the results for components of Table 1. For each component, we have executed each of its interfaces 20 times separately 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 experiments performed on a large number of additional Java components [9,17,18].

As discussed in Section 4, multiple regression can be used for characterizing the energy consumption of invoking type II interfaces. For this we used a tool called DataFit [2]. In measurements we conducted on over 50 different type II interfaces, our estimates of their energy consumption have been within 5% of the actual energy costs. As an illustration, Figure 4 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 2. Several actual energy costs are shown for illustration as the discrete points on the graph.



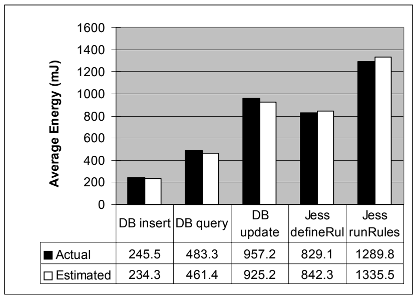
1	2	3	4	5	6	7	8	9	10	11
SHA	MD5	IDEA	Median Filter	LZW	Sort	Jess	DB	Shortest path	AVL	Linked List

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



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



	DB insert	DB query	DB update	Jess defineRul	Jess runRules
■ Actual	245.5	483.3	957.2	829.1	1289.8
□ Estimated	234.3	461.4	925.2	842.3	1335.5

Figure 5. Framework's accuracy for type III interface of *DB* and *Jess* 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 2, and calculated its average energy consumption. Figure 5 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%.

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 set up a fixed environment in which the network's transmission and receipt bandwidths are constant. In particular, we fixed the UDP transmission/receipt bandwidths available to the iPAQ to 625 *KB/sec* by using *Iperf* [7]. We then used the digital multimeter to measure the actual energy consumptions on the iPAQ as a result of transmitting and receiving a sample set of messages of various sizes to/from the laptop. 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):

$$y_t = 2.7572 * x_t + 0.015 \qquad \textbf{\textit{Eq. 8}}$$

$$y_r = 2.5639 * x_r + 0.012 \qquad \textbf{\textit{Eq. 9}}$$

We then 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 0.015 represents the parameter *tS* in Equation 4, which is independent of the size of transmitted data. The variables *tEC* and *tBW* are captured by the constant factor 2.7572, as follows:

Total Data (MB)	0.98	1.95	2.86	3.82	4.76	5.72
Actual (J)	2.647	5.207	7.694	10.232	12.747	15.288
Estimated (J)	2.634	5.231	7.636	10.19	12.691	15.245
Total Data (MB)	6.68	7.64	8.58	9.54	10.5	11.44
Actual (J)	17.809	20.343	22.855	25.441	27.887	30.441
Estimated (J)	17.799	20.353	22.455	25.373	27.962	30.243

Figure 6. Overall communication energy estimation on the iPAQ with fixed bandwidth of 625 KB/sec.

$$\frac{tEC}{tBW} = \frac{tEC\ (J/\sec)}{0.625\,(\mathrm{MB}/\sec)} = 2.7572\ (\frac{\mathrm{J}}{\mathrm{MB}})$$

From the above we can determine that *tEC = 1.7232 J/sec.* The values of *rS* and *rEC* can be calculated in the same manner using Equation 9. We can thus express the overall energy cost of the iPAQ for communicating with the laptop as a specific instance of Equation 4:

$$y_{commEC} = (2.7572 * x_t + 0.015) + (2.5639 * x_r + 0.012) \qquad \textbf{\textit{Eq. 10}}$$

Using this approach, we have been able to estimate the overall communication cost of a device to within 3% of the actual values in our experiments. For illustration, Figure 6 shows energy costs for the iPAQ estimated by Equation 10 versus the energy costs obtained by actual measurements. A comparable error rate (under 3%) has recurred across our experiments regardless of the bandwidth used.

6.3. Overall Energy Cost

Figure 7 shows our setup for the evaluation of a distributed system, where we have connected an iPAQ with two other hosts via a wireless router. Hosts B and C run Windows XP and Fedora Core 4 Linux, respectively. Both host B and C use an IEEE 802.11b compliant wireless network card for communication.

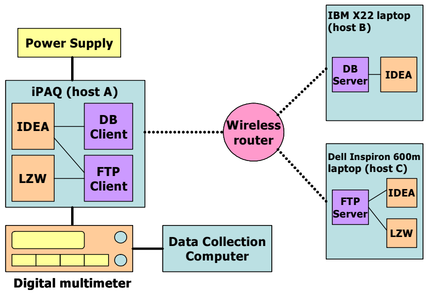


Figure 7. A distributed Java-based system comprising three hosts.

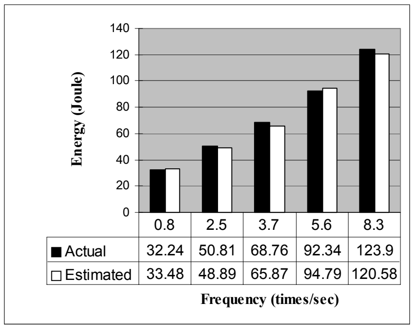
Figure 7 also shows an example software system deployed across the three hosts. Each software component interacts with the other components via a UDP socket. A line between two components (e.g., `IDEA` and `FTP Client` on the iPAQ) represents an interaction path between them. We have used several execution scenarios in this particular system. For example, `DB Client` component may invoke the `query` interface of the remote `DB Server`; in response, `DB Server` calculates the results of the query, and then invokes `IDEA`'s `encrypt` interface and returns the encrypted results to `DB Client`; finally, `DB Client` invokes the `decrypt` interface of its collocated `IDEA` component to get the results.

We have executed the above distributed software system by varying the frequencies of messages exchanged among the components. We have measured the total energy consumption on the iPAQ due to invoking its components' interfaces and compared it with our framework's estimates. In the process, we have measured the average UDP transmission and receipt bandwidths (723 KB/sec and 672 KB/sec, respectively) used by host A for communicating with hosts B and C. We can thus recalculate the communication cost from Equation 10 for this example scenario as follows:

$$y = \left[2.7572*(625/723)*x_t + 0.015\right] + \left[2.5639*(625/672)*x_r + 0.012\right]$$
$$= (2.3834 * x_t + 0.015) + (2.3845 * x_r + 0.012)$$

As shown in Figure 8, our estimates always fall within 5% of the actual energy costs regardless of interaction frequencies. These results have been consistently corroborated by a large number of additional distributed applications.

We should note that one restriction we faced in our experiments, such as the one depicted in Figure 7 and described above, was the availability of a single digital multimeter. This meant that, regardless of the numbers of software components in an application and hosts on which they are deployed, we could only measure the energy consumption on a single device at a time. However, the design of the experiments we conducted, the large number of distribution scenarios we considered, the results we obtained on several platforms, and the minimal interference of the digital multimeter with those results, all suggest that performing multiple such measurements simultaneously would not have had any significant effect on our results.



	0.8	2.5	3.7	5.6	8.3
■ Actual	32.24	50.81	68.76	92.34	123.9
□ Estimated	33.48	48.89	65.87	94.79	120.58

Figure 8. Framework's accuracy with respect to the interaction frequency.

7. Conclusion

In this paper we have proposed and evaluated a framework for estimating the energy consumption of Java-based software systems. Our objective is to enable an engineer to make informed decisions when adapting a system's architecture, such that the lifetime of the system's critical services is maximized. Our framework explicitly takes a component-based perspective, which renders it well suited for a large class of today's distributed, dynamic, and mobile applications. The framework is applicable both during system construction-time and during runtime. In a large number of distributed application scenarios the framework has shown very good precision on the whole, giving results that have been within 5% (and often less) of the actually measured power losses incurred by executing the software.

We believe that the framework has significant utility as-is. At the same time, throughout the paper we have identified several potential adjustments that should further improve the framework's accuracy. An important direction of future research will be extending this work to multi-lingual systems, with all of their inherent challenges (e.g., lack of a common execution platform such as the JVM, reliance on third-party middleware, and so on). Another research direction will be investigating and quantifying the impact of existing energy saving techniques: runtime adaptation of component interfaces (e.g., to lower the quality of provided services), off-loading of "unimportant" components while maintaining the most critical functionality, and so on. We consider the development and evaluation of our framework to be the crucial first step in pursuing these avenues of further work.

8. References

[1] P. D. Allison. Multiple regression. Pine Forge Press. 1999.
[2] Data Fit 8.1. http://www.oakdaleengr.com/, 2006.
[3] K. I. Farkas et. al. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. *ACM SIGMETRICS*, 2000.
[4] P. Gauthier et. al. Reducing Power Consumption for the Next Generation of PDAs. In *Proceedings of MoMuC'96*, 1996.
[5] L. M. Feeney et. al. Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. In *Proceedings of IEEE INFOCOM*, 2001.
[6] HSQLDB 1.8.0. http://www.hsqldb.org/, 2005.
[7] Iperf. http://dast.nlanr.net/Projects/Iperf/, 2006.
[8] JamVM 1.3.2. http://jamvm.sourceforge.net/, 2006.
[9] Java Grande. http://www.epcc.ed.ac.uk/javagrande/
[10] Jess. http://www.jessrules.com/, 2005.
[11] Kaffe 1.1.5. http://www.kaffe.org/, 2005.
[12] S. Khurshid et. al. Generalized Symbolic Execution for Model Checking and Testing. *TACAS*, 2003
[13] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, vo.19, no. 7, 1976.
[14] KVM. http://java.sun.com/products/cldc/wp/, 2005.
[15] S. Lafond, et. al. An Energy Consumption Model for An Embedded Java Virtual Machine. *ARCS*, 2006.
[16] H. Singh, et. al. Energy Consumption of TCP in Ad Hoc Networks. *Wireless Networks,* vol. 10, no. 5, 2004.
[17] SourceBank. http://archive.devx.com/sourcebank/
[18] sourceForge.net. http://sourceforge.net/
[19] JVM98 Benchmarks. http://www.spec.org/jvm98/, 2001.
[20] EVM. http://www.sun.com/research/java-topics, 2001.
[21] T. K. Tan et. al. Energy macromodeling of embedded operating systems. *ACM Trans. on Embedded Comp. Systems*, 2005.
[22] V. Tiwari, and T. C. Lee. Power Analysis of a 32-bit Embedded Microcontroller. *VLSI Design Journal*, vol. 7, no. 3, 1998.
[23] V. Tiwari et. al. Power Analysis of Embedded Software: a First Step Towards Software Power Minimization. *IEEE Trans. on VLSI Systems*, vol. 2, no. 4, 1994.
[24] N. Vijaykrishnan et. al. Energy Behavior of Java Applications from the Memory Perspective. *Java VM*, 2001.