Estimating the Energy Consumption in Pervasive Java-Based Systems

Chiyoung Seo[1] Sam Malek[2] 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

We define and evaluate a framework for estimating the energy consumption of pervasive Java-based software systems. The framework's 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 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 provides a novel approach that facilitates the accurate estimation of a system's energy consumption both during system construction-time and during runtime. In a large number of distributed application scenarios, the framework showed very good precision on the whole, giving results that were within 5% of the actually measured power losses incurred by executing the 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 power sources, many new computing platforms have finite battery lives. Consider for illustration the distributed sensor application shown in Figure 1, which consists of eight software components deployed and running on three battery-powered hosts. Each line between two components represents an interaction path between them, while each dotted line between two hosts is a physical network link. This distributed application provides various user-level services. For instance, a fire-alarm sensor interacts with its environment and periodically transmits an alarm signal to the *FAR* (*Fire Alarm Receiver*) component on Host 1, which then forwards the alarm to the *FAA* (*Fire Alarm Analyzer*) component on Host 2. The *FAA* component then analyzes the alarm message and determines whether there is actually a fire.

The main motivation for our research is that if we could estimate the energy costs of a given software system (e.g., sensor application in Figure 1) in terms of its constituent software components ahead of its actual deployment, or during runtime, we would be able to take appropriate component-level actions to extend the system's life span: unloading unnecessary or expendable software components, redeploying highly energy-intensive components to more capacious hosts, collocating frequently communicating components, and so on.

In this paper we present a framework that estimates the energy consumption of a pervasive Java-based software system at the level of its software components. Our estimation framework provides a novel approach that facilitates the estimation of a system's energy consumption during construction-time and the refinement of construction-time estimates during 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 a recent paper [11], we outlined the overall architecture of our estimation framework. This paper provides a more detailed explanation of how the framework is used both prior to and during runtime, and presents the evaluation results of the framework.

We have chosen to focus on Java because it is increasingly used in constructing pervasive applications [5]. Clearly, certain types of software applications (e.g., highly computationally intensive programs such as those recently considered by Mantovani et al. [10]) will still call for solutions in languages such as C and C++ because of their better performance and lower energy consumption. However, Java is becoming attractive for pervasive systems because it sup-

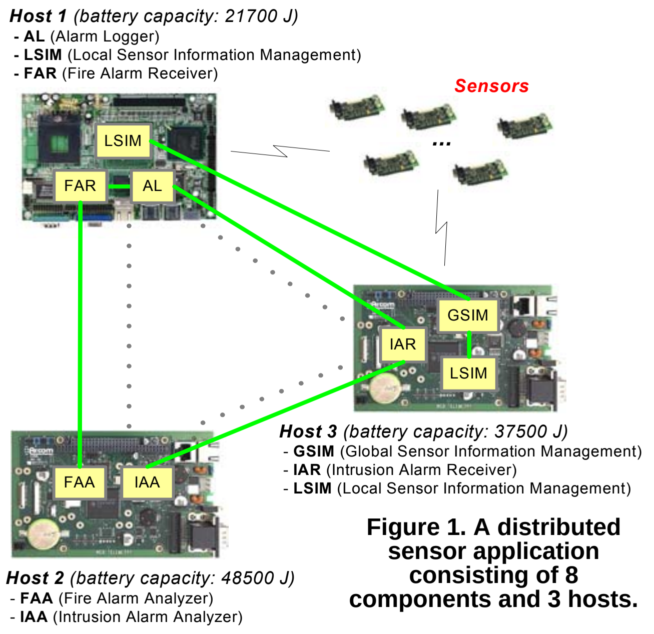
Host 1 (battery capacity: 21700 J)
- **AL** (Alarm Logger)
- **LSIM** (Local Sensor Information Management)
- **FAR** (Fire Alarm Receiver)
Sensors
...
LSIM
FAR
AL
GSIM
IAR
LSIM
FAA
IAA
Host 3 (battery capacity: 37500 J)
- **GSIM** (Global Sensor Information Management)
- **IAR** (Intrusion Alarm Receiver)
- **LSIM** (Local Sensor Information Management)
Host 2 (battery capacity: 48500 J)
- **FAA** (Fire Alarm Analyzer)
- **IAA** (Intrusion Alarm Analyzer)

Figure 1. A distributed sensor application consisting of 8 components and 3 hosts.

ports platform-independence, language-level dynamism, and relative ease of system deployment and maintenance, all the while continuously improving in performance and application footprint size.

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 able to estimate the power consumed by a distributed Java system to within 5% of the actual consumption.

In the remainder of this paper we first present the related research in the energy estimation 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 results (Section 5). We then conclude the paper (Section 6).

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. [9] 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.

A number of measurement- and simulation-based tools have been presented for estimating the energy consumption of embedded operating systems (OSs) or applications. One of the representative tools is a web-based tool, *JouleTrack*, for estimating the energy cost of an embedded software running on StrongARM SA-1100 and Hitachi SH-4 microprocessors [12]. While they certainly informed our work, we were unable to use these tools directly in our targeted distributed Java domain because none of them provide generic energy consumption models, but instead have focused on individual applications running on specific platforms, which indicates that they cannot be used for Java-based software systems running on platforms different from theirs.

Previous research has also proposed solutions for the energy accounting problem by computing the overall energy consumption of a system in terms of individual processes within an operating system [16]. Although these approaches have some similarities with ours, we cannot adapt their results directly because they model the system's energy consumption at the level of processes while in Java multiple software components usually run in a single JVM process.

Several studies [4,15] 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 leverage these experimental results in defining a component's communication energy cost.

3. ENERGY COST FRAMEWORK

We model a pervasive Java-based 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 briefly discuss our approach to modeling each of these three energy cost factors. Refer to [11] for a more detailed explanation of our energy consumption models

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 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 facility for synchronizing threads via a *monitor* mechanism. Consequently, we can model the computational energy cost of invoking an interface on a given JVM in terms of the energy costs of bytecodes, native methods, and *monitor* operations executed during the invocation. The computational energy cost of a component can be then calculated by aggregating the energy costs of the component's constituent interfaces.

Unless two platforms have the same hardware configurations, JVMs, and OSs, the energy costs of each bytecode type, each native method, and a monitor operation will likely be different.

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. 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.SocketInput-`

1. We use the them "interface" in a broader sense than the language-level construct supported by Java. Our usage is consistent with component-based software engineering literature.

`Stream`'s `read()`) for socket communication, this is also captured by a component's computational cost.

For remote communication between components, we focus on modeling the energy consumption due to UDP-based interactions. Since UDP is a much more light-weight protocol than TCP (e.g., UDP provides no congestion control or error recovery), it is becoming increasingly prevalent in resource-constrained pervasive domains [2,14]. For example, UPnP [14] supports web service invocations on top of HTTP-UDP. Previous research [4,15] has shown that the energy consumption of wireless communication is directly proportional to the size of transmitted and received data. Based on these results, we model the communication energy cost due to invoking a component's interface in terms of the size of transmitted and received data and the platform-specific energy consumption of transmitting/receiving a unit of data. The communication energy cost of a component can be then modeled by aggregating the communication energy costs of the component's interfaces.

3.3. Infrastructure Energy Overhead

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 of the GC thread (*Joule/second*) and monitoring the total time the thread is active (*second*).

As a JVM runs as a separate process in an OS, it is necessary to consider the energy overhead caused by implicit OS routine calls for facilitating and managing the execution of JVM processes. Previous research has shown that process scheduling, context switching, and paging are the main consumers of energy due to implicit OS routine calls [13]. Therefore, we can estimate the overall infrastructure energy overhead of each JVM process in terms of the energy costs of the GC thread, process scheduling, context switching, and paging. Unless two platforms have the same hardware configurations, JVMs, and OSs, the GC thread's energy consumption rate and the energy costs of process scheduling, context switching, and paging on one platform may not be the same as those on the other platform.

Finally, we can estimate the system's overall energy consumption by aggregating the energy costs of all the components and the infrastructure energy overhead of all JVMs.

4. ENERGY CONSUMPTION ESTIMATION

We have implemented our framework within Kaffe 1.1.5 JVM [6] by instrumenting its source code for obtaining automatically (1) the numbers of bytecodes, native methods, and monitor operations executed due to an interface's invocation; (2) the size of data exchanged over the network; and (3) the GC thread execution time and the numbers of implicit OS routines executed on the JVM. Below we highlight the steps that a system engineer must take in using our framework:

1. For each unique platform, profile platform-specific energy cost parameter (e.g., energy cost of each type of bytecode and native method, energy cost of sending a unit of data) required by our framework. The engineer can use our approach described in [11] to do this automatically for a given platform. This is a one-time effort for each type of platform in a distributed system.
2. Use the framework for estimating the energy costs of a component's interfaces and the infrastructure energy overhead by generating a set of inputs for each interface during system construction-time.
3. The framework can then refine the above construction-time estimates automatically during runtime by monitoring various system's properties.

In the remainder of this section, we detail how our framework can be used both during system construction-time and during runtime.

4.1. Construction-Time Estimation

In order to estimate a distributed system's energy cost at construction-time, 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 our computational model.

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 our computational model. 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 (in our case, each generated input for a type II interface) and the corresponding output value (energy consumption calculated from our computational model), 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 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., 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 [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 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 our computational model, 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 automatically invoke the interface with a large set of random inputs, calculate the energy consumption of the interface for each input via our computational model, 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).

The above classification of a component's interfaces is based on their normal execution paths. If an interface's implementation has any exception handling routines, they must be treated separately in calculating the interface's energy cost. For estimating the energy cost due to processing an exception, we target the exception by generating inputs that raise it. We then profile the number of bytecodes, native methods, and monitor operations executed as a result of those inputs, and again estimate the energy consumption from our computational model. Our estimates of the frequency with which the exception code will be executed can be adjusted at runtime as detailed in Section 4.2.

To estimate the communication energy consumption of each interface, based on domain knowledge and types of input parameters and return values, we predict the average size of messages exchanged due to an interface's invocation. Using this data we can approximate the communication energy cost of interface invocation via our communication model. 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).

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 our energy cost model we can estimate the infrastructure's energy cost. We do so by executing all components on their target hosts *simultaneously*, with the same sets of inputs that were used in characterizing the energy consumption of each individual component.

4.2. Runtime Estimation

Many systems for which energy consumption is a significant concern are long-lived, dynamically adaptable, and mobile. An energy cost framework for such systems should account for variations in 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 after a system's 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 computation (i.e., number of bytecodes, native methods, and monitor operations) is monitored by our framework during runtime.

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 overall energy cost can be updated automatically.

Finally, the fact that the frequency at which interfaces are invoked 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, the GC thread execution time and the number of implicit OS routines invoked at runtime must also be monitored. Based on the refined estimates of each interface's computational and communication costs, and of the infrastructure's energy overhead, we will be able to improve (possibly automatically) our construction-time energy estimates of distributed systems at runtime.

5. EVALUATION

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

5.1. Evaluation 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

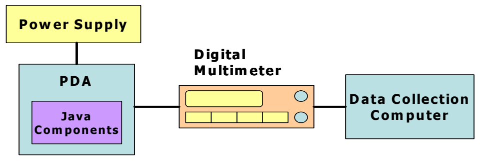

Figure 2. Experimental setup.

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.

5.2. Evaluation Results

We have evaluated our framework over a large number of distributed Java-based applications. Figure 3 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 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 and returns the encrypted results to `DB Client`; finally, `DB Client` invokes the `decrypt` interface of its collocated `IDEA` component to get the results.

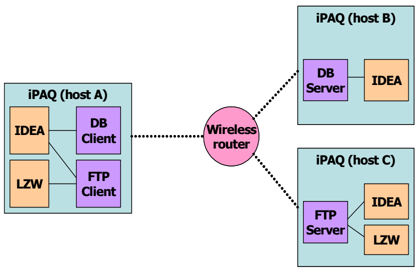

Figure 3. A distributed Java-based system comprising three hosts.

We have executed the above distributed software system for both cases, varying the frequencies and sizes of messages exchanged among the components. We have measured the system’s overall energy consumption and compared it with our framework’s estimates. As shown in Figure 4, our estimates always fall within 5% of the actual energy costs regardless of interaction frequencies and the average size of a single message. These results have been corroborated by a large number of additional distributed applications.

6. CONCLUSION

In this paper we have proposed and evaluated a framework for estimating the energy consumption of pervasive Java-based software systems. Our framework explicitly takes a component-based perspective, which renders it well suited for a large class of today’s distributed pervasive applications. The framework is applicable both during system construction-time and runtime. In our experiments the framework has shown very good precision, giving results that have been within 5% (and often less) of the actual energy consumption 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, which has been identified as important in the areas of distributed, embedded, and pervasive systems. We have recently begun exploring, and successfully applying in an industrial setting, one such avenue for the framework.

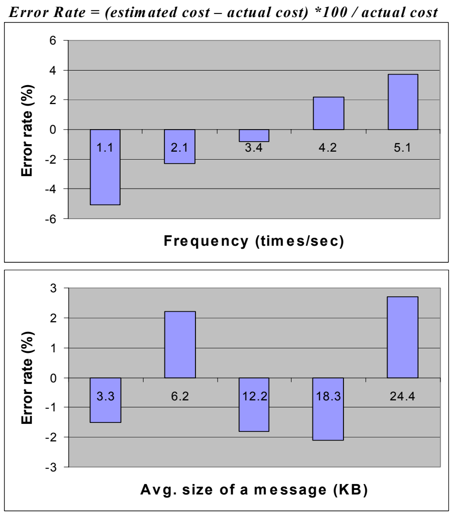

Figure 4. The framework’s error rates with respect to the interaction frequency (top) and the average size of a message (bottom).

7. REFERENCES

[1] P. D. Allison. Multiple regression. Pine Forge, 1999.
[2] W. Drytkiewicz, et al. pREST: a REST-based protocol for pervasive systems. *IEEE International Conference on Mobile Ad-hoc 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*, 2000.
[4] L. M. Feeney, et al. Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. *IEEE INFOCOM*, 2001.
[5] S. Helal. Pervasive Java. *IEEE Pervasive Computing*, Vol. 1, No. 1, Jan-Mar, 2002.
[6] Kaffe 1.1.5. http://www.kaffe.org/, 2005.
[7] S. Khurshid, et al. Generalized Symbolic Execution for Model Checking and Testing. *TACAS*, 2003
[8] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, vo.19, no. 7, 1976.
[9] S. Lafond, et al. An Energy Consumption Model for An Embedded Java Virtual Machine. *ARCS*, 2006.
[10] M. Mantovani, et al. A Lightweight Parallel Java Execution Environment for Embedded Multiprocessor Systems-on-Chip. *GLSVLSI* 2007, Italy, March 2007.
[11] C. Seo, et al. An Energy Consumption Framework for Distributed Java-Based Systems. In *Proc. of IEEE/ACM Int’l Conference on Automated Software Engineering*, November, 2007
[12] A. Sinha, et al. JouleTrack - A Web Based Tool for Software Energy Profiling. In *Proceedings of DAC,* 2001.
[13] T. K. Tan, et al. Energy macromodeling of embedded operating systems. *ACM Trans. on Embedded Comp. Systems*, 2005.
[14] UPnP Device Architecture, http://www.upnp.org/, 2007.
[15] R. Xu, et al. Impact of Data Compression on Energy Consumption of Wireless-Networked Handheld Devices, *ICDCS*, 2003.
[16] H. Zeng, et al. ECOSystem: Managing Energy as a First Class Operating System Resource. ACM *ASPLOS*, 2002.