Runtime Monitoring of Software Energy Hotspots

Adel Noureddine, Aurélien Bourdon, Romain Rouvoy, Lionel Seinturier

Runtime Monitoring of Software Energy Hotspots

Adel Noureddine[1,2], Aurelien Bourdon[1,2], Romain Rouvoy[1,2] and Lionel Seinturier[1,2,3]
[1] INRIA Lille – Nord Europe, Project-team ADAM
[2] University Lille 1 - LIFL CNRS UMR 8022, France
[3] Institut Universitaire de France
firstname.lastname@inria.fr

ABSTRACT

GreenIT has emerged as a discipline concerned with the optimization of software solutions with regards to their energy consumption. In this domain, most of the state-of-the-art solutions concentrate on coarse-grained approaches to monitor the energy consumption of a device or a process. However, none of the existing solutions addresses in-process energy monitoring to provide in-depth analysis of a process energy consumption. In this paper, we therefore report on a fine-grained runtime energy monitoring framework we developed to help developers to diagnose energy hotspots with a better accuracy than the state-of-the-art.

Concretely, our approach adopts a 2-layer architecture including OS-level and process-level energy monitoring. OS-level energy monitoring estimates the energy consumption of processes according to different hardware devices (CPU, network card). Process-level energy monitoring focuses on Java-based applications and builds on OS-level energy monitoring to provide an estimation of energy consumption at the granularity of classes and methods. We argue that this per-method analysis of energy consumption provides better insights to the application in order to identify potential energy hotspots. In particular, our preliminary validation demonstrates that we can monitor energy hotspots of JETTY web servers[1] and monitor their variations under stress scenarios.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics

General Terms
Performance, Measurement, Experimentation

Keywords
Power Model, Power Monitoring, Profiling, Bytecode Instrumentation

[1]`http://www.eclipse.org/jetty`

1. INTRODUCTION

Energy-aware software solutions are becoming broadly available, as energy concerns is becoming mainstream. The increasing usage of computers and other electronic devices (*e.g.*, smartphones, sensors) is continuously impacting our overall energy consumption. Information and Communications Techonology (ICT) accounted for 2% of global carbon emissions in 2007 [5] or 830 $MtCO_2e$[2], and is expected to grow to 1,430 $MtCO_2e$ in 2020 [14]. ICT also consumed up to 7% of global power consumption (or 168 Gigawatt, GW) in 2008 [13]. This number is predicted to grow and double to 433 GW in 2020 or more than 14.5% of worldwide power consumption [13]. These values illustrate the opportunities for efficient ICT solutions to reduce carbon emissions and energy consumption. Rising energy costs in computers and mobile devices requires the optimization and the adaptation of computer systems. In this domain, research in GreenIT already proposes various approaches aiming at achieving energy savings in computers and software. However, most of the state-of-the-art approaches either focus only the hardware [8], or only offer coarse-grained energy estimation feedback of software [4, 2].

In this paper, we therefore propose to gather fine-grained applications' power feedback information at runtime and with similar accuracy as hardware monitoring while using only a software approach. Our approach, called E-SURGEON, consists of a system monitoring library (at the operating system level), called POWERAPI, and a software monitoring agent, called JALEN. E-SURGEON estimates the power consumption of applications' source code methods, in real-time, based on raw information collected from hardware devices (*e.g.*, CPU, network card) through the operating system (for POWERAPI), and from raw information collected from software (CPU time, bytes transmitted through network) through bytecode instrumentation (for JALEN). We use both state-of-the-art power models and propose new models for estimating the power consumption of software at a finer grain.

As a first implementation, we target Java-based applications and we validate our approach using standard application servers, such as the JETTY Web Server. Our preliminary results demonstrate that we can diagnose power hotspots of Java-based applications at runtime, offering opportunities to reduce their power consumption.

[2]Metric Tonne Carbon Dioxide Equivalent

The remainder of this paper is organized as follows. In Section 2, we describe our motivations and the main challenges we tackle. Section 3 describes our approach, the design of our proposed architecture and our power models. Section 4 details the implementation of our prototype. In Section 5, we report on the preliminary results we obtained and we validate them using a stress benchmark for the JETTY Web Server in Section 6. Related work is discussed in Section 7, while we conclude in Section 8.

2. MOTIVATION AND CHALLENGES

2.1 Motivation

Nowadays, power management of software and hardware is achieved either through runtime coarse-grained monitoring, or through analyzing dump files of the application's resources utilization [8, 12, 7]. Although these approaches allow power management of software, they do not allow runtime and fine-grained monitoring of the applications. Fine-grained monitoring and visualization have many advantages: *i)* diagnose at a detailed level the power consumption and detect power hotspots at the threads and methods level, *ii)* provide detailed power information to be used for runtime power-aware software adaptation, and *iii)* helps in providing insights to developers for producing power-efficient code. The Green Challenge for USI 2010 [1] has identified that profiling applications to detect CPU hotspots is a winning strategy for limiting the power consumption of applications. Therefore, we argue that a fine-grained approach for proposing power-aware information is a keystone for future power-aware systems and software.

2.2 Challenges

Hardware monitoring is usually achieved through additional hardware measurement equipments, such as multimeters or specialized integrated circuits (cf. Section 7). This approach offers a precise and accurate measurement of the power consumption of hardware components but at a cost of an additional investment. However, it can neither monitor the power consumption of software components, nor go into the details of software classes and methods usages. We rather believe that a scalable approach can be better obtained through a software-centric approach. Monitoring the power consumption of software has to yield many challenges in order to build an accurate software-centric approach. We outline some of the main difficulties that software monitoring has to cope with if accurate monitoring is to be offered:

- **Accuracy.** The biggest problem that software monitoring tools face is providing accurate estimations of power consumption based on various collected information. Unlike hardware measurement, software approaches use power models in order to provide an estimation of the power consumption of software components. However, these estimations tend to have different degrees of accuracy and overhead.

- **Overhead.** As software approaches monitor the executing software and calculate a power estimation of their consumption, an overhead is therefore always observed. The latter depends both on the degree of accuracy needed and on the size of the monitoring tool and the monitored application. This leads to a trade-off between the accuracy requirements and the cost of the software monitoring tool.

- **Fine-grained.** Many of the current approaches (cf. Section 7) stop their power consumption estimation at the process level. Some of these approaches provide limited fine-grained but still raw values (such as execution time of methods or active time of threads). However, providing fine-grained estimation of the power consumption of software components is not as intuitive as mixing raw values and power models. The question arises to know which raw values is needed. How can we collect them? Which power models can we use and in which context?

- **Power Models.** Models to estimate the power consumption have already been proposed (cf. Section 7). However, most of these models are coarse-grained and hardware related, such as providing general formulae for power consumption of the hardware components (*e.g.*, CPU, Network card). Models therefore need to be optimized for our context of fine-grained power consumption computation.

Laying these challenges, we propose in the next section an approach named E-SURGEON, for monitoring and profiling applications at runtime.

3. E-SURGEON DESIGN AND APPROACH

In this section, we present our power monitoring approach called E-SURGEON. We first present E-SURGEON general architecture, then we describe the power models we integrated.

3.1 Architecture

E-SURGEON architecture is composed from two distinct but complementary parts: a system-level power monitoring environment called POWERAPI; and a software-level application profiling environment called JALEN. These two parts work along each other in order to provide accurate runtime energy information at the application level (threads and methods levels). Figure 1 depicts the overall architecture of our approach.

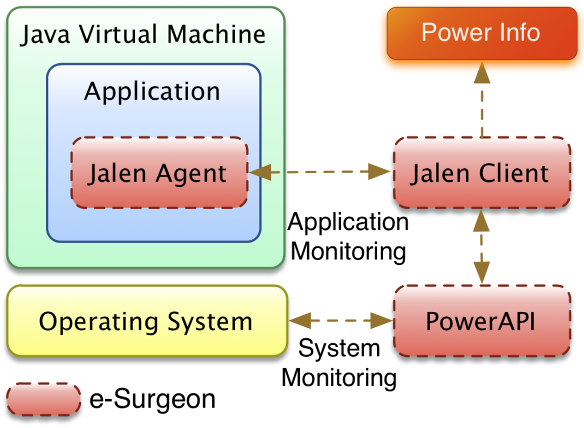

Figure 1: E-SURGEON **Reference Architecture.**

3.1.1 PowerAPI

PowerAPI architecture is a modular architecture built for agile software programming. The core of the architecture are *power modules*. PowerAPI is developed as separate modules that can be started or stopped at runtime upon needs. A set of OS-dependent *sensor* modules (*e.g.*, S_{CPU}, $S_{Network}$) collect raw information about hardware resource utilization, either directly from the devices or through the operating system. These information are then exposed to another set of OS-independent *formula* modules that uses our power models (cf. Section 3.2) to compute the power consumption of each hardware component (*e.g.*, F_{CPU}, $F_{Network}$). These modules also compute the power consumption of running processes and applications per hardware resource. A local database (DB in Figure 2) is also used to store static information about the hardware resources and used to automatically calibrate PowerAPI depending on the environment. Finally, all these modules are managed through a *life cycle* module. The latter allows to start, stop, add, remove or modify modules depending on monitoring needs and commands sent by applications.

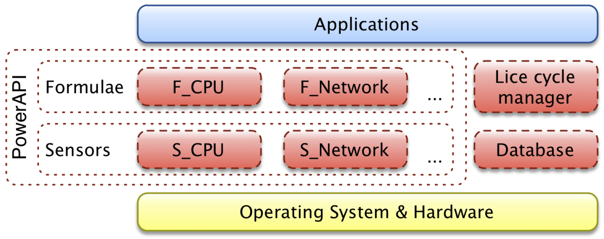

Figure 2: PowerAPI **Reference Architecture.**

3.1.2 Jalen

Jalen is a software-level profiling architecture. It is responsible for profiling running applications and estimating their energy consumption at a finer grain, *i.e.*, at threads or methods level. Several profiling techniques can be used, such as bytecode instrumentation or sampling the application. Each of these methods have benefits and drawbacks. The former does not modify the code of the application, and provides an overview of the application's energy consumption. However, it is less accurate than code instrumentation. The latter injects profiling code into the application's code (or bytecode), therefore allowing the profiler to capture all the necessary events related to energy consumption. On the other hand, instrumentation adds an overhead for running the additional code. Our approach, however, does not specify a single method of profiling. We prefer to keep this as a technical choice during implementation and to use dedicated APIs to communicate with the profiler and the low-level monitoring environment are to be respected.

The profiling part introspects the application at runtime, collecting statistics about its resources utilization. Information, such as methods durations, CPU time, or the number of bytes transferred through the network card, are collected and classified at a finer grain, *e.g.*, for each method of the application. Next, a correlation phase takes place to correlate the application-specific statistics with the process-level power information. Details on our power model for the correlation are presented in Section 3.2. Finally, the per-method or per-thread power consumption information is displayed to the user and can be exposed as a service (to be used, for example, in an application's autonomous adaptation cycle).

3.2 Power Models

We propose a comprehensive power model using our proposed formulae as well as formulae taken from the state-of-the-art. In [12], the energy cost of a software is computed based on the following formula:

$$E_{software} = E_{comp} + E_{com} + E_{infra} \tag{1}$$

Where E_{comp} is the computational cost (*i.e.*, CPU processing, memory access, I/O operations), E_{com} is the cost of exchanging data over the network, and E_{infra} is the additional cost incurred by the OS and runtime platform (*e.g.*, Java VM).

We base our model on a similar principle, taking into account the modular aspect of the power calculation (*e.g.*, the sum of the power consumption of different hardware components). Infrastructure power, E_{infra}, is included in the computational cost of our power models and in our prototype. Power (in watt) is computed as the energy consumption (in joule) per unit of time (one second). From this, we can abstract our global power formula to the following:

$$P_{software} = P_{comp} + P_{com} \tag{2}$$

At this stage, we defined two models, one for CPU computational costs and one for network communication costs. P_{comp} is therefore equal to the CPU power consumed by software, and P_{com} is equal to the power consumed by the network card for transmitting software's data. Next, we detail the CPU and network power models we use in PowerAPI, and the CPU and network power models we use in Jalen.

PowerAPI Power Models

3.2.1 CPU Model

The CPU power consumed by a specific application (in our case we use process $PIDs$) can be represented by the following formula:

$$P_{CPU}^{PID}(d) = P_{CPU}(d) \times U_{CPU}^{PID}(d) \tag{3}$$

Where $P_{CPU}^{PID}(d)$ is the CPU power consumed by the specific process *PID* during a given duration d, $P_{CPU}(d)$ is the overall CPU power during d and $U_{CPU}^{PID}(d)$ represents the process CPU usage during d. Thus, our approach is to estimate the power required by the CPU to execute the process *PID*. This is achieved by computing the CPU percentage usage of the *PID* by the overall CPU power during a given duration d. Next, we detail our model in order to compute $P_{CPU}(d)$, the global CPU power, and $U_{CPU}^{PID}(d)$, the process CPU usage.

Overall CPU power

The overall power consumption for the majority of modern processors (CMOS[3]) follows the standard equation [10]:

$$P_{CPU}^{f,v} = c \times f \times V^2 \tag{4}$$

[3]Complementary Metal Oxide Semiconductor

Where f is the frequency, V the voltage and c a constant value depending on the hardware materials (such as the capacitance and the activity factor). Thanks to this relation, we note that power consumption is not always linearly dependent to the percentage of CPU utilization. This is due to DVFS (*Dynamic Voltage and Frequency Scaling*) and also to the fact that power depends on the voltage (and subsequently the frequency) of the processor. For example, a process at 100% CPU utilization will not necessarily consume more power than a process running at 50% CPU utilization but with a higher voltage. Therefore, a simple CPU utilization profiler is not enough in order to monitor power consumption. Our power model takes into consideration these aspects of the CPU and allows accurate power consumption monitoring.

According to formula (4), computing the overall CPU power for a given time is equal to computing a static part (the constant c) and a dynamic part (the frequency f and its associated voltage V). For the static part, the c constant is a set of data describing the physical CPU (*e.g.*, capacitance, activity factor). Manufacturers may provide this constant, but in most cases this value is missing. To alleviate this problem, we use the existing relation between the overall power of a processor and its *Thermal Design Power* (TDP) value. TDP represents the power the cooling system of a computer is required to dissipate the heat produced by the processor. Therefore, the overall CPU power can be associated with the TDP as described in the following formula [11]:

$$P_{CPU}^{f_{TDP},V_{TDP}} \simeq 0.7 \times TDP \tag{5}$$

Where f_{TDP} and V_{TDP} represent respectively the frequency and the voltage of the processor within the *TDP state*. The benefit of using this formula is that TDP is a value provided by most manufacturers. In our architecture, TDP is stored in PowerAPI's local database.

For the dynamic part, the frequency f is associated to a specific voltage V. For a given voltage, one or more frequencies are associated. Thus, lowering the voltage results in changing frequency. The other way around is also valid, although in some cases a single voltage can support more than one frenquency. Frequencies used by a processor are provided by the operating system APIs, while voltages are given by manufacturers.

Process CPU usage

In order to compute the CPU usage for a given process (identified by its *PID*), we propose to calculate the ratio between the CPU time for this *PID* and the global CPU time (the time the processor is active for all processes) during a duration d:

$$U_{CPU}^{PID}(d) = \frac{t_{CPU}^{PID}}{t_{CPU}}(d) \tag{6}$$

Our approach is inspired by well-known tools such as the Top linux program[4]. Thus, the CPU power consumption in a duration d and for a given frequency f, P_{comp}^{f} is equal to :

$$P_{comp}^{f} = \frac{0.7 \times TDP}{f_{TDP} \times V_{TDP}^2} \times f \times V^2 \times \frac{t_{CPU}^{PID}}{t_{CPU}}(d) \tag{7}$$

[4]`http://linux.die.net/man/1/top`

Finally, the CPU power consumption for a process P_{comp} of formula (2) is equal to the average of the CPU power of each frequency balanced by the CPU time of all frequencies:

$$P_{comp} = \frac{\sum_{f \in frequencies} P_{comp}^{f} \times t_{CPU}^{f}}{\sum_{f \in frequencies} t_{CPU}^{f}} \tag{8}$$

3.2.2 Network Model

The network power of a process is calculated using a formula similar to the CPU power formula. We base our model on available information whether they are collected at runtime or provided by manufacturers' documentations. As a first step, we focus on Ethernet network cards. A similar model using a linear equation can be applied for wireless network cards [3], but we did not investigate wireless cards yet. We obtain, from manufacturers' documentations the power consumed (in watt) for transmitting bytes for a certain duration (typically one second) according to a given throughput mode of the network card (*e.g.*, 1 MB, 10 MB). Our network power model is therefore defined as:

$$Power_{process}^{network} = \frac{\sum_{i \in states} t_i \times P_i \times d}{t_{total}} \tag{9}$$

Where P_{state} is the power consumed by the network card in the state i (provided by manufacturers), d is the duration of the monitoring cycle, and t_{total} is the total time spent in transmitting data using the network card.

In the next section, we detail the CPU and network power models in Jalen.

Jalen Power Models

3.2.3 CPU Model

Using the information collected from profiling applications and the monitored system, we are able to calculate a reasonable estimation of the CPU time per method. And we use this information to compute the CPU power consumed per method and thread. As application code is generally executed inside threads (*i.e.*, Java), we first calculate the power consumed per thread. For that, we apply the following formula:

$$Power_{thread}^{CPU} = \frac{Time_{thread}^{CPU} \times Power_{process}^{CPU}}{Duration_{cycle}} \tag{10}$$

Where $Time_{thread}^{CPU}$ is the CPU time of the thread in the last monitoring cycle (obtained from the environment, such as the OS or the JVM), $Power_{process}^{CPU}$ is the power consumed by the application process in the last monitoring cycle (obtained from PowerAPI), and $Duration_{cycle}$ is the duration of the monitoring cycle. We then filter the methods to get the list of methods running in the last monitoring cycle (whether they are still running or not). For each thread, we get the methods that it invoked from the list (a thread usually has its own execution stack which is made of frames. A frame represents a method invocation). Furthermore, we estimate with a good accuracy the CPU time for each method using the following formula:

$$Time_{method}^{CPU} = \frac{Duration_{method} \times Time_{thread}^{CPU}}{\sum_{m \in Methods} Duration_{m}} \tag{11}$$

Where $Duration_{method}$ is the execution time of the method in the last monitoring cycle, and $\sum Duration_{methods}$ is the sum of the execution time of all methods in the last monitoring cycle.

Finally, we calculate the power consumed per method using this formula:

$$Power^{CPU}_{method} = \frac{Time^{CPU}_{method} \times Power^{CPU}_{thread}}{Duration_{cycle}} \qquad (12)$$

Network Model

We calculate the network power using the number of bytes transmitted by the application. We first calculate the number of bytes read/written in the last monitoring cycle. Then, we collect the network power consumed by the application process from our system library PowerAPI. The network power consumed per method is therefore:

$$Power^{Network}_{method} = \frac{Bytes_{method} \times Power^{Network}_{process}}{Bytes_{process}} \qquad (13)$$

Where $Bytes_{method}$ is the number of bytes read and written by the method, $Power^{Network}_{process}$ is the power consumed by the application, and $Byte_{process}$ is the number of bytes read and written by all methods of the application.

The network power consumption per thread is therefore the sum of the network power of all methods running in the thread as shown in the following formula:

$$Power^{Network}_{thread} = \sum Power^{Network}_{methods} \qquad (14)$$

In the next section, we describe the implementation e-Surgeon using our power models.

4. IMPLEMENTATION

The implementation of e-Surgeon includes a system level modular library, PowerAPI, and a Java tool instrumenting bytecode at runtime, Jalen. CPU and network modules as well as profiling functionalities are provided. The source code of our e-Surgeon prototype is available at github[5].

4.1 PowerAPI

PowerAPI is implemented as a system level modular library. We implemented so far the CPU and network modules and power models. Our system-level library aims to provide power information per PID for each system component (CPU, network card, etc.). The library is therefore based on a modular approach where each system component is represented as a *power module*. *Power modules* operate independently from each other and are composed by two sub-modules: *formula* and *sensor*. These sub-modules communicate using the *Service-Oriented Architecture* (SOA) paradigm, and are contained in an OSGi[6] container. In particular, we use Service Oriented Framework (SOF)[7] to implement the various modules of PowerAPI in C++. The sensor sub-module is responsible for gathering operating system related information for the module. For example, it gathers the number of bytes transmitted by the network card, and the time spent by the CPU at each of the processor frequencies (when DVFS is supported). These data are basically given by the operating system, giving to the sensor sub-module of being OS-dependent. In particular, our implementation, based on a GNU/Linux distribution, exploits system information available in the *procfs* and *sysfs* file systems [9]. The formula sub-module, on the other hand, is hardware independent. Indeed, this sub-module is responsible for estimating the power consumed for each process by using both information gathered by the sensor sub-module and information based on hardware characteristics. For instance, in case of CPU energy consumption, our model has to take into account the frequency and voltage ranges, or Thermal Dissipation Power. We implemented sensor and formula sub-modules for the CPU and NIC on a GNU/Linux operating system. Additionally, our library supports the lifecycle management of its power modules. The latter can be started, stopped and their parameters changed at runtime, using a *modules manager*. The benefit of this modular approach is to offer flexibility while monitoring the system and to control the overhead of the monitoring library.

We provide a modular library where only part of its components are platform-dependent. Its modularity allows easy porting of the library while retaining most of its power modeling code. Although PowerAPI works as a standalone library, it is used in addition to our application power monitoring component: Jalen.

4.2 Jalen

We also developed a runtime application power monitoring component, called Jalen, as part of the e-Surgeon architecture. Jalen uses the per-process power information provided by PowerAPI, and correlates it with information collected from the application monitoring in order to provide per-method power information. Jalen is designed as a two-parts architecture: *i)* a Java programming language agent that instruments the bytecode of the application in order to inject our monitoring code, and *ii)* a JMX client computing and correlating the collected data. This client extracts the per-method power consumption values. Figure 3 overviews the architecture of Jalen.

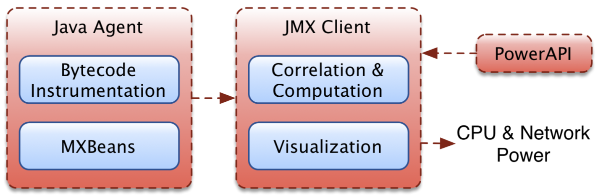

Figure 3: The Jalen Architecture.

We use bytecode instrumentation technics[8] to inject our monitoring code into the methods of legacy applications. In order to reduce the overhead introduced by the instrumentation, we offload all the computations to a remote JMX client. Therefore, our Java agent only collects raw values, such as for the CPU: the start time, end time, running status, executing thread identifier and callee method (using a custom execution stack trace). For the network, we use a delegator

[5]https://github.com/abourdon/e-Surgeon
[6]formerly Open Service Gateway Initiative
[7]`http://sof.tiddlyspot.com`
[8]`http://asm.ow2.org`

class to route calls from the class `SocketImpl`[9] to a custom implementation. We override the methods `getInputStream()` and `getOutputStream()` to monitor the number of bytes read and written to sockets. This information is then correlated with the method names invoking the methods `getInputStream()` or `getOutputStream()`, in order to get the number of bytes read/written by method.

All these collected information are then exposed in a `MXBean`[10] interface. In particular, the agent performs calculations in order to determine the actual duration of the execution of the method during the last monitoring cycle. This calculation takes into account all the calls to the methods executed in a same thread. It also separates the calculations of these methods by thread. For example, method *A()* is called twice from thread *X*, and three times from thread *Y*. The calculations generate two results: one where the duration is the sum of the two calls in thread *X*, and the second where the duration is for the sum of the three calls in thread *Y*. We do not merge these numbers because we need to construct the call tree in order not to take into account the delta duration of the callee method when its children are being executed. Our prototype can handle this on a per-thread basis thanks to this separation.

Then, we build a JMX client that, not only displays the collected information, but also does the computation and correlation in order to determine the CPU and network power consumed per method in each monitoring cycle. The JMX client collects information exposed by the agent's `MXBean`. Using power models and metrics correlations (cf. Section 3.2), we estimate the CPU power and the network power consumed per method.

5. EMPIRICAL VALIDATION

We validate the accuracy and precision of our E-SURGEON prototype on a Dell Precision T3400 workstation with an Intel Core 2 Quad processor (Q6600), running Ubuntu Linux 11.04 and Java 1.6 We first evaluate our POWERAPI library (cf. Section 5.1), and then evaluate our JALEN Java agent (cf. Section 5.2). Based on these results, we conduct an analysis of a stress benchmark on a JETTY Web Server (version 8.0.4.v20111024) in Section 6.

5.1 PowerAPI Validation

5.1.1 CPU Power

We first assess the accuracy of the results provided by our system library. We compared the power values provided by POWERAPI with the actual power consumption of the computer using a powermeter. In our tests we use POWERSPY[11], a bluetooth powermeter. We compare the values of our library and the powermeter in a stress test on JETTY Web Server using APACHE JMETER[12] (cf. Figure 5), and using the Linux STRESS command[13] (cf. Figure 4). Note that due to synchronization time lag between POWERSPY and our library, values are shifted for a few seconds in the beginning of the monitoring. These values are normalized in order to observe trends in the CPU power consumption. We normalize these values by subtracting for each measured value of the powermeter, the average of the differences between the values measured by the powermeter and provided by our library. The results show minor variations between the values estimated by our library, and the actual power consumption. The margin of error is estimated to 0.5% of the normalized and averaged values in the core stressing scenario. The error grows to nearly 3% in the JETTY stress test. We use the Linux TOP program to compute the CPU utilization of monitoring one process by POWERAPI and estimate it at around 0.1%. Therefore, we can reasonably argue that using a software-centric approach provides values that are accurate enough to be used by power management software.

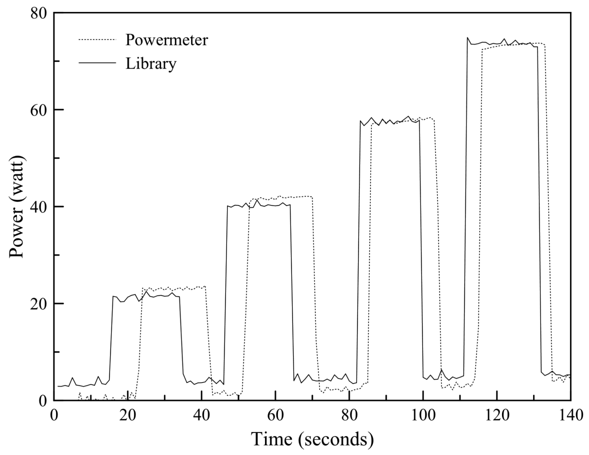

Figure 4: Stressing the processor cores with the STRESS command.

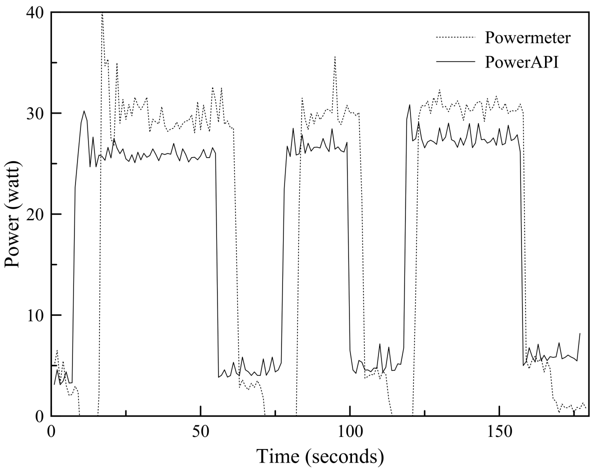

Figure 5: Running stress tests on JETTY using JMETER.

[9]http://docs.oracle.com/javase/6/docs/api/java/net/SocketImpl.html
[10]http://docs.oracle.com/javase/6/docs/api/javax/management/MXBean.html
[11]http://www.alciom.com/en/products/powerspy2.html
[12]http://jmeter.apache.org
[13]http://linux.die.net/man/1/stress

5.1.2 *Network Power*

We run a network stress test using IPERF[14] and measured the power consumption of IPERF's CPU server on our host configuration. We send two sets of TCP packets of 100MB each from a distributed client to our host server. We used the default settings of IPERF, where also its CPU server executes following a periodically cycle (every second). Our results show network consumption around 0.017 watt compared to CPU consumption of IPERF process around 0.9 watt. These numbers show that, although CPU power is quite low (average around 0.9 watt) and the network card uses all its capacity, the consumed network power is largely negligible compared to the consumed CPU power on our test server. This observation is in correlation with the literature [11]. Therefore, we mostly outline the results of our CPU experimentation.

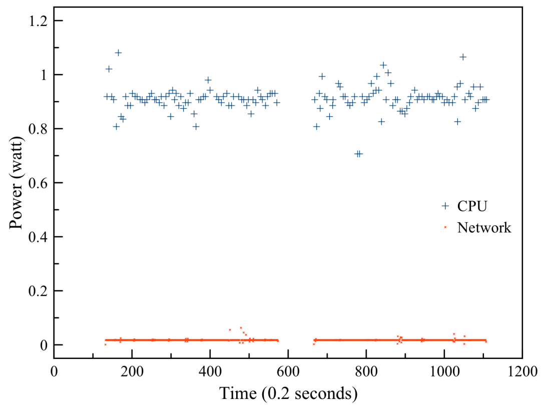

Figure 6: CPU and network power consumption in IPERF stress test.

5.2 Jalen Validation

We calculate the CPU overhead of our JALEN Java agent using the start time and the time per request of APACHE TOMCAT 7.0.22 application server[15] as a comparison metric. This metric involves many of TOMCAT's classes and methods and is provided by default by TOMCAT. The average start time of TOMCAT on our host configuration (a MacBook Pro, mid 2009, and Java 1.6) is around 821.8 ms. With JALEN Java agent, TOMCAT, while having its methods instrumented, took 1178 ms in average. Therefore, our agent introduces an overhead of 43.34%. We also calculate the time per request using ApacheBenh 2.3. On 10,000 requests, the mean time per request is at 20.206 ms with Jalen, in comparison to 12.861 ms without. The time overhead per request is therefore at 57.11%. Even though these numbers may appear to be high, they should compared to the overhead of similar profilers. We therefore compared our results of Tomcat's start time with the JAVA INTERACTIVE PROFILER (JIP)[16]. JIP also instruments bytecode but does not offer power related information. With JIP, TOMCAT took 1469.6 ms to start, or an overhead of 78.82%. Thus, JALEN have an overhead 45% lower to similar method-level bytecode instrumentation profilers such as JIP. Figure 7 summarizes these numbers.

[14]http://www.manpagez.com/man/1/iperf
[15]http://tomcat.apache.org
[16]http://jiprof.sourceforge.net

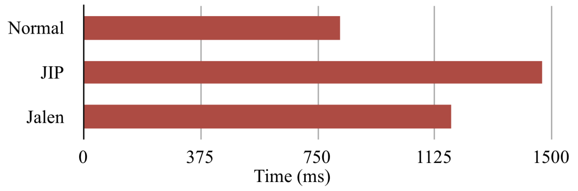

Figure 7: Time overhead of JALEN with TOMCAT 7 start time.

In the next section, we present and discuss the results of our approach on monitoring and detecting energy hotspots of Jetty Web Server.

6. ENERGY HOTSPOTS OF JETTY

The goal of our approach is to detect energy hotspots in applications. JETTY Web Server is an example of such complex applications, counting 88,513 source lines of code (SLOC) in the version we used for our study (version 8.0.4.v20111024). We run our E-SURGEON prototype (system library and Java agent plus a JMX client) on an instance of JETTY Web Server. We use JMETER to stress JETTY server using a benchmark scenario (stressing the examples provided by default in JETTY). We run the experimentation for an average time of one minute, with 20 threads (users in JMETER) and a loop count of 500. The results we gathered are presented in Figure 8. The graph portrays the top 10 most power-consuming methods in the X axis (out of 726 instrumented methods). The right Y axis (thus the bars) represents the power consumed during our execution in percentage of the total power consumed. The left Y axis (thus the line) represents the number of invocations of the method. We run this experiment several times, and although we had difference in the watt power values, we notice that the global and proportional percentage is stable. Note also that the provided values are an aggregation of the execution of the methods on all threads.

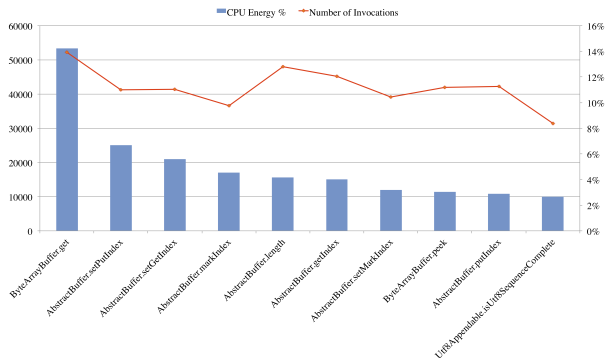

Figure 8: Cumulated power consumption of JETTY methods under JMETER stresses (top 10 most power-consuming methods).

The first observation is that the top 10 methods consumes nearly half of the total power consumed by JETTY during the stress benchmark (50.99%). More interestingly, we observed that the method `org/eclipse/jetty/io/ByteArrayBuffer.get` consumes 14.22% of the total power by its own, with a similar number of invocations compared to other methods.

We also analyze the power consumption per method invocation of the top 10 methods. The results are presented in Figure 9. We observe that `org/eclipse/jetty/io/AbstractBuffer.putIndex` has a lower power per invocation in the top 10 methods. This method consumed 2.89% of total power during the tests, but was invoked 42293 times. Thus, this method has a power per invocation of 134.057 microwatts. On the other hand, `org/eclipse/jetty/io/ByteArrayBuffer.get` has a power per invocation of 535.237 microwatts with 52159 invocations and 14.22% of global power consumed.

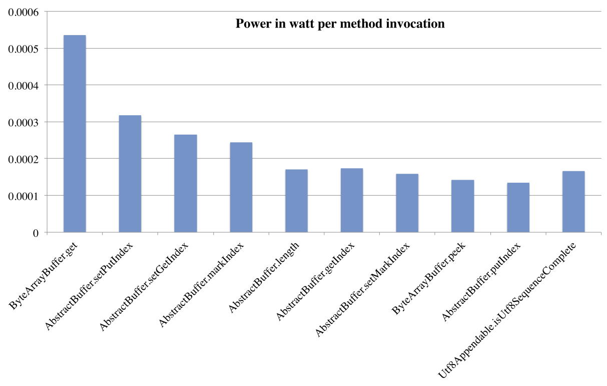

Figure 9: Power in watt per invocation of the top 10 most power-consuming methods (lower is better).

The results for the most consuming classes (out of 146) are reported in Figure 10. We note that the 2 aforementioned classes (`org/eclipse/jetty/io/AbstractBuffer` with 39.22% and `org/eclipse/jetty/io/ByteArrayBuffer` with 23.89%) consume alone more than 60% of the total power (63.11%). `io/AbstractBuffer` is an averaged sized class in Jetty of 544 SLOC where only 10 of its methods has been invoked 255,472 times, while the latter is even smaller with 304 SLOC where 26 of its methods have been invoked 606,710 times. These numbers show a strong relation between the energy consumption of methods and their number of invocation. These results can be explained by our benchmark stress scenario. The latter stresses Jetty Web Server using HTTP requests on servlets with simple executed JSP code. Therefore, the highest power consumption in our tests is concentrated on classes which manage reads and writes of HTTP requests (*e.g.*, `org/eclipse/jetty/io/AbstractBuffer`, `org/eclipse/jetty/io/ByteArrayBuffer`). This strengthen our initial validation of the accuracy of e-Surgeon.

We believe that this information can help the developers to investigate alternative implementations of the class `org/eclipse/jetty/io/ByteArrayBuffer` in order to reduce the power footprint of this method. By keeping track of the power footprint of classes and methods, we think that development tools (*e.g.*, coding completion systems, documentation, debuggers, etc.) could be extended to help developers build *greener* software.

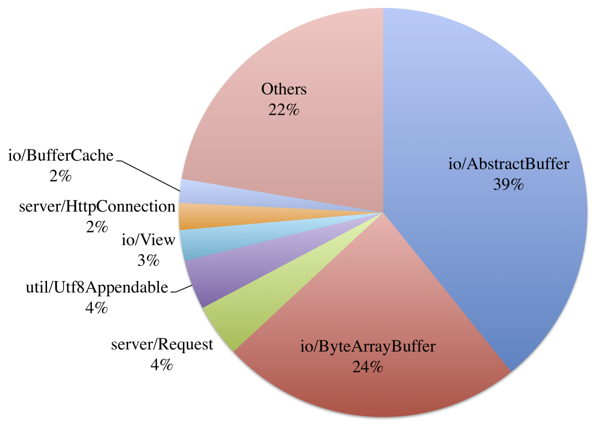

Figure 10: Cumulated power consumption of Jetty classes under JMeter stresses.

6.1 Limitations

Our results show that we can identify energy hotspots in applications, and in Jetty web server in particular. However, and even though we used a benchmark imitating a real-case scenario, we are aware that our results should not be generalized without further consideration.

First, our aim was to observe trends in energy consumption (which is the reason of using percentages when comparing methods and classes). Raw watt values are less relevant as direct measurements vary greatly between hardware (*i.e.*, a given application consumes different amount of energy when running on two different machines, such as a laptop or a server). Therefore, we argue that our approach is useful in profiling applications in order to find the origin of energy leaks. Developers can then provide *hotfixes* for the application in order to reduce its energy footprint.

Second, and in comparison with our first point, the overhead of our prototype is not negligible, thus limiting its usage in production (57.11% overhead for Tomcat's individual requests (cf. Section 5.2)). Although it is comparable to similar bytecode profilers, the overhead is penalizing to be used in production systems (*e.g.*, Tomcat, Jetty). We also developed a lightweight version, providing the energy consumption of running threads (but not methods and classes). This version does not use bytecode instrumentation, therefore its overhead is negligible in Jetty and Tomcat and with a 1.95% increase in time. Note also that the computation of the energy consumption of methods and threads is realized in a separate process, and that PowerAPI runs also in a dedicated process, thus limiting the impact on the execution of the monitored application. We believe that the overhead of our power model and architecture is controlled, and can be considered is acceptable with regards to the benefits of the computed energy indicators.

6.2 Future Directions

Our study aims at providing a representation of the energy consumption of CPU- and network- intensive software at different levels of granularity (*e.g.*, application, source code). In the future, we plan to extend ou approach to measure hardware components that contribute to a high percentage in energy consumption of applications [11], which are, in addition to the CPU, the memory and the disk.

We are also planning to develop additional sensors for different hardware configurations and environments because energy consumption depends on the hardware environment (*e.g.*, modules compatible with Windows, Mac OS, and different hardware models). Developing sensors and modules for virtual machines allows our model to reduce its dependency on hardware parameters (*e.g.*, dealing with the diversity of hardware is therefore left to the virtual machine), and consider energy accounting issues in the context of green computing environments.

7. RELATED WORK

In this section, we outline the relevant related works to energy modeling, energy monitoring and metering, power-related tools at the system level, and application profiling tools, in particular for Java applications.

Energy Metering and Modeling

Monitoring energy consumption of hardware components usually requires an hardware investment, like a multimeter or a specialized integrated circuit. For example in [8], the energy management and preprocessing capabilities is integrated in a dedicated ASIC (*Application Specific Integrated Circuit*). It continuously monitors the energy levels and performs power scheduling for the platform. However, this method has the main drawback of being difficult to upgrade to newer and more precise monitoring and it requires that the hardware component be built with the dedicated ASIC, thus making any evolution impossible without replacing the whole hardware. On the other hand, an external monitoring device provides the same accuracy as ASIC circuits and does not prohibit energy monitoring evolutions. The previous monitoring approaches retrieve energy measures about hardware components only. However, knowing the energy consumption of software services and components requires an estimation of that consumption. This estimation is based on calculation formulae as in [12] and [7].

In [12], the authors propose formulae to compute the energy cost of a software component as the sum of its computational and communication energy costs. For a Java application running in a virtual machine, the authors take into account the cost of the virtual machine and eventually the cost of the called OS routines. Our model is based on a similar principle, although we abstract the cost of the infrastructure in our computational costs. However, the authors calculate the energy cost of components in terms of the cost of its interfaces (*i.e.*, a method in most cases). The latter is calculated as an estimation of the energy cost of executing Java's 256 bytes code types, JVM's native methods, and the cost of threads synchronization. Our computation model is based on runtime power consumption. The CPU power consumed by a method is its percentage share of the power consumed by the application, calculated using the actual CPU time and utilization of the method. On the other hand, our network model is similar to theirs, as both are based on the size of data transmitted (send/receive) during the invocation of the program. Still, we use runtime-monitored values to calculate power consumption, while they use estimations at construction time albeit refined at runtime. In [7], the authors take into account the cost of the *wait* and *idle* states of the application (*e.g.*, an application consumes energy when waiting for a message on the network). We also take these states into account by only using the actual time spend running on a resource (*i.e.*, CPU, network card). In [4], the authors propose a tool, PowerScope, for profiling energy usages of applications. This tool uses a digital multimeter to sample the energy consumption and a separate computer to control the multimeter and to store the collected data. PowerScope can sample the energy usage by process. This sampling is more accurate than energy estimation, although it still needs a hardware investment.

System Level Tools

pTop [2] is a process-level power profiling tool. Similar to the Linux Top program, the tool provides the power consumption (in Joules) of the running processes. For each process, it gives the power consumption of the CPU, the network interface, the computer memory and the hard disk. The tool consists in a daemon running in the kernel space and continuously profiling resource utilization of each process. It obtains these information by accessing the */proc* directory. For the CPU, it also uses TDP provided by constructors in the energy consumption calculations. It then calculates the amount of energy consumed by each application in a t interval of time. It also consists of a display utility similar to the Linux Top utility.

Our approach is more flexible and fine-grained than pTop. Not only we offer process-level power information, but we also go deep into the application in order to profile and report thread and method-level power consumptions. Furthermore, the system level part of e-Surgeon offers better flexibility and on-demand scaling of the tool. Monitoring modules can be shutdown or started depending on the context: on limited resources devices, modules, such as the network or hard disk modules, can be shutdown in order to monitor only the CPU. When more resources become available, these modules will be re-started. Other situations are also possible, such as situations where the user is only interested in monitoring the CPU or the network energy consumption. PowerAPI also adapts to its monitored environment thanks to its auto-calibration process, in particular by using calibration data stored in its database. Our flexible and modular approach therefore offers these functionalities, and extends them to not only OS processes, but also inside Java-based applications profiling.

In addition to pTop, several utilities exist on Linux for resource profiling. For example, cpufrequtils[17], in particular cpufreq-info to get kernel information about the CPU (*i.e.*, frequency), and cpufreq-set to modify CPU settings such as the frequency. iostat[18] that is used to get devices'

[17]`http://kernel.org/pub/linux/utils/kernel/cpufreq/cpufrequtils.html`

[18]`http://linux.die.net/man/1/iostat`

and partitions' input/output (I/O) performance information, as well as CPU statistics. Other utilities [6] also exist with similar functionalities, such as SAR, MPSTAT, or the system monitoring applications available in Gnome, KDE or Windows. However, all of these utilities only offer raw data (*e.g.*, CPU frequency, utilized memory) and do not offer power information.

Application Profiling Tools

Several open-source or commercial Java profiling tools already propose some statistics of Java applications. Tools, such as VisualVM[19], Java Interactive Profiler (JIP)[20], or the Oktech Profiler[21], offer coarse-grained information on the application and fine-grained resource utilization statistics. However, they fail in providing power consumption information of the application at the granularity of threads or methods. For example, the profiler of VisualVM only provides self wall time (*e.g.*, time spend between the entry and exit of the method) for its instrumented methods. We rather provide runtime values for the duration of execution of methods in a monitoring cycle, and give a good estimation of the CPU time of these methods. These tools also lack of providing network related information, such as the number of bytes transmitted by methods and thus the power consumed.

8. CONCLUSION AND FUTURE WORK

In this paper, we report on the E-SURGEON runtime energy monitoring solution. It allows to gather and calculate the power consumption at processes and methods level. Its modular architecture allows runtime context-based adaptations of the monitoring environment itself, leveraging performance and accuracy at the wish of the application or the user. We also propose power models to calculate the power consumption. Our models use and extend the state-of-the-art models and formulae, and port them to a fine-grained context. Our initial results show the potential of our approach for diagnosing, at runtime, power hotspots of Java-based applications. As for future work, we plan to: *i)* propose more power models for other hardware resources (in particular, memory and disk); *ii)* as application servers are more and more running on virtual machines, we plan to implement specific sensors to these environments and experiment our model and approach on them; and *iii)* use E-SURGEON and power-aware information to adapt applications at runtime based on power concerns.

Acknowledgments

This work is partially funded by the French Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the Contrat de Projets Etat Region Campus Intelligence Ambiante (CPER-CIA) 2007-2013.

9. REFERENCES

[1] The Green Challenge for USI 2010. `http://sites.google.com/a/octo.com/green-challenge`.

[2] T. Do, S. Rawshdeh, and W. Shi. pTop: A Process-level Power Profiling Tool. In *HotPower'09: Proceedings of the 2nd Workshop on Power Aware Computing and Systems*, Big Sky, MT, USA, october 2009.

[3] L. Feeney and M. Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *INFOCOM'01: Proceesing of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1548–1557, 2001.

[4] J. Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *WMCSA'99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, page 2, Washington, DC, USA, 1999. IEEE Computer Society.

[5] Gartner. Green IT: The New Industry Shockwave. In *Gartner*, Presentation at Symposium/ITXPO Conference, 2007.

[6] V. Gite. How do I Find Out Linux CPU Utilization? `http://www.cyberciti.biz/tips/how-do-i-find-out-linux-cpu-utilization.html`.

[7] A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. *SIGMETRICS Perform. Eval. Rev.*, 36(2):26–31, 2008.

[8] D. McIntire, T. Stathopoulos, and W. Kaiser. ETOP: sensor network application energy profiling on the LEAP2 platform. In *IPSN'07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 576–577, New York, NY, USA, 2007. ACM.

[9] E. Mouw. Linux kernel procfs guide, June 2001.

[10] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *MMSA'00: Proceesings of the 2nd International Symposium on Mobile Multimedia Systems and Applications*, pages 157–164, Delft, The Netherlands, 2000.

[11] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. JouleSort: a balanced energy-efficiency benchmark. In *SIGMOD'07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 365–376, New York, NY, USA, 2007. ACM.

[12] C. Seo, S. Malek, and N. Medvidovic. An energy consumption framework for distributed java-based systems. In *ASE'07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 421–424, New York, NY, USA, 2007. ACM.

[13] W. Vereecken, W. Van Heddeghem, D. Colle, M. Pickavet, and P. Demeester. Overall ict footprint and green communication technologies. In *ISCCSP'10: Proceedings of the 4th International Symposium on Communications, Control and Signal Processing*, pages 1 –6, 2010.

[14] M. Webb. *SMART 2020: enabling the low carbon economy in the information age, a report by The Climate Group on behalf of the Global eSustainability Initiative (GeSI)*. GeSI, 2008.

[19] `http://visualvm.java.net`

[20] `http://jiprof.sourceforge.net`

[21] `http://code.google.com/p/oktech-profiler`