e-Surgeon: Diagnosing Energy Leaks of Application Servers

Adel Noureddine, Aurélien Bourdon, Romain Rouvoy, Lionel Seinturier

e-Surgeon: Diagnosing Energy Leaks of Application Servers

Adel Noureddine[1,2], Aurélien Bourdon[1], Romain Rouvoy[1,2], 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

RESEARCH REPORT
N° 7846
January 2012
Project-Teams ADAM

ISSN 0249-6399 ISRN INRIA/RR--7846--FR+ENG

e-Surgeon: Diagnosing Energy Leaks of Application Servers

Adel Noureddine[1,2], Aurélien Bourdon[1], Romain Rouvoy[1,2], 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

Project-Teams ADAM

Research Report n° 7846 — January 2012 — 27 pages

Abstract: GreenIT has emerged as a discipline concerned with the optimization of software solutions with regards to 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 real-time energy monitoring framework we developed to diagnose energy leaks 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, memory). 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 leaks. In particular, our preliminary validation demonstrates that we can diagnose energy hotspots of Jetty application servers and monitor their variations when stressing web applications.

Key-words: Energy, Performance, Measurement, Power Model, Monitoring, Profiling, Bytecode Instrumentation

e-Surgeon: Diagnostic des fuites énergétiques des serveurs d'applications

Résumé : L'informatique verte a émergé comme une discipline qui s'intéresse à l'optimisation des solutions logicielles en ce qui concerne la consommation d'énergie. Dans ce domaine, la plupart des solutions de l'état de l'art se concentre sur des approches à gros grains pour contrôler la consommation énergétique d'un matériel ou un processus. Toutefois, aucune des solutions existantes gère la surveillance au niveau processus afin de fournir une analyse en profondeur de la consommation énergétique d'un processus. Dans ce papier, nous proposons un canevas logiciel à grain fin pour surveiller en temps réel la consommation énergétique des applications, et pour diagnostiquer les fuites d'énergie avec une meilleure précision que l'état de l'art.

En particulier, notre approche adopte une architecture à 2 couches, une au niveau du système d'exploitation et le suivi de l'énergie au niveau des processus. La couche de surveillance de l'énergie au niveau de l'OS estime la consommation énergétique au niveau du processus selon différents périphériques matériels (processeur, réseau, mémoire). La couche de surveillance de l'énergie au niveau des processus se concentre sur les applications Java et s'appuie sur la couche OS pour fournir une estimation de la consommation d'énergie à la granularité des classes et méthodes. Nous soutenons que cette analyse au niveau des méthodes de la consommation énergétique fournit un meilleur aperçu de l'application afin d'identifier les fuites énergétiques potentielles. En particulier, nos expériences démontrent que nous pouvons diagnostiquer les hotspots énergétique des serveurs d'application Jetty et de surveiller leurs variations lorsque nous mettons sous pression les applications web.

Mots-clés : Énergie, Performance, Mesure, Modèle énergétique, Surveillance, Profilage, Instrumentation du bytecode

1 Introduction

Energy-aware software solutions and approaches are becoming broadly available as energy concerns is becoming mainstream. The increasing usage of computers and other electronic devices (*e.g.*, smartphones, sensors, or digital equipment) is continuously impacting our overall energy consumption. Although ICT accounted for 2% of global carbon emissions in 2007 [14], ICT solutions can help in reducing the energy footprint of other sectors (*e.g.*, building, transportation, industry). In [30], the Climate Group estimates that ICT solutions could reduce carbon emissions by 15% in 2020. However, in 2007, ICT footprint was 830 $MtCO_2e$ and is expected to grow to 1,430 $MtCO_2e$ in 2020 [30]. 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, or only offer coarse-grained energy feedback and optimization.

In this paper, we therefore propose to gather fine-grained energy feedback information at runtime and in real-time. Our approach monitors the hardware and the software of a given system, and is able to report the energy consumption at a finer grained than the state-of-the-art. In particular, the solution we propose an approach that profile applications more deeply with providing thread-level and method-level energy information.

Our approach consists of a 2-layer architecture including a system monitoring library (at the OS level), and an application energy profiler (at the process level). The system level library estimates the energy consumption of running processes, in real-time, based on raw information collected from hardware devices (*e.g.*, CPU, network card) and the operating system. The application profiler gathers resources information of software (*e.g.*, CPU time, bytes transmitted using the network card) and builds a usage model of system resources for each thread and method. We both use state-of-the-art energy models and propose new models for calculating the energy consumption of software at a fine-grained level.

As a first implementation, we target Java-based applications and we validate our approach using standard application servers, such as the Jetty Web Server [17]. Our preliminary results demonstrate that we can diagnose energy hotspots of Java-based applications at runtime, offering opportunities to reduce their energy consumption.

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 energy 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. Related work is discussed in Section 6, while we conclude in Section 7.

2 Motivation and Challenges

2.1 Motivation

Nowadays, energy management of software and hardware is achieved either through real-time coarse-grained monitoring, or through analyzing dump files of the application's resources utilization. Although these approaches allow energy management of software, they do not allow real-time and fine-grained monitoring of the applications. Fine-grained monitoring and visualization have many advantages: *i)* diagnose at a detailed level the energy consumption and detect energy hot spots at the threads and methods level, *ii)* provide detailed energy information to be used for runtime energy-aware software adaptation, and *iii)* helps in providing insights to developers for producing energy-efficient code. The Green Challenge for USI 2010 [5] has identified that profiling applications to detect CPU hotspots is a winning strategy for limiting the energy consumption of applications. Therefore, we argue that a fine-grained approach for proposing energy-aware information is a keystone for future energy-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 6). This approach offers a precise and accurate measurement of the energy consumption of hardware components but at a cost of an additional investment. However, it can neither monitor the energy 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 energy 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 energy consumption based on various collected information. Unlike hardware measurement, software approaches uses energy models in order to provide an estimation of the energy 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 an energy 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 difficult tradeoff between the accuracy requirements and the weight of the software monitoring tool.

- **Fine-grained.** Many of the current approaches (cf. Section 6) stop their energy consumption calculation 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 energy consumption of software components is not as intuitive as mixing raw values and energy models. The question arises to know which raw values is needed. Where can we collect them? Which energy models can we use and in which context?

- **Energy Models.** Models to estimate the energy consumption have already been proposed (cf. Section 6). However, most of these models are coarse-grained and hardware related, such as providing general formulas for energy consumption of the hardware components (*e.g.*, CPU, Memory, Hard disk). Models therefore need to be optimized for our context of fine-grained energy 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 e-Surgeon general architecture and we describe the approach we use for defining our energy models. The architecture is based on a modular approach, mixing power monitoring and profiling tools with energy models in order to provide energy information per-method in software.

3.1 Architecture

e-Surgeon architecture is split in two parts: a low level hardware resources monitoring environment; and a high level application profiling environment. These two parts work along each other in order to provide accurate real-time energy information at the application level (threads and methods levels). Figure 1 depicts the overall architecture of our approach.

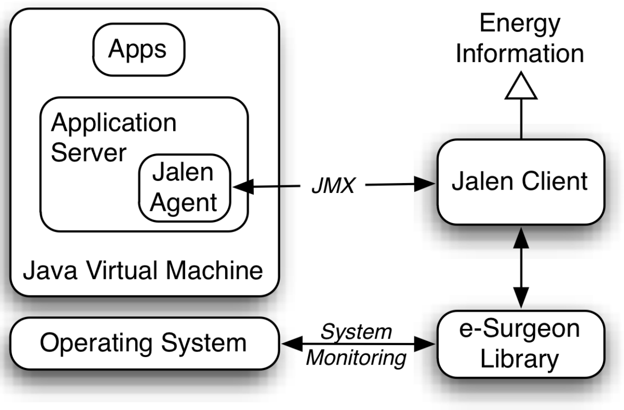

Figure 1: e-Surgeon Reference Architecture.

Low-level monitoring

The low-level monitoring part (named *e-Surgeon Library* in Figure 1) is the base layer of our architecture. It provides system resources (*e.g.*, CPU, network card, . . .) energy consumption by process (PID) in real-time. It is based on a modular design, adapting to both the execution environment and to the needs of the application profiling.

This low-level monitoring is constructed as separate modules that can be started or stopped at runtime upon needs. A set of modules (*e.g.*, CPU, network) collects raw information about the hardware resource utilization directly from the devices or through the operating system. These information are then exposed to another set of modules (*e.g.*, energy CPU, energy network) that uses our energy models (cf. Section 3.2) to compute the energy consumption for each hardware component. These modules also compute the energy consumption of running processes per hardware resource. Finally, all of these modules are managed through a module manager. The manager is responsible for starting, stopping or modifying the modules at runtime, by following commands sent by applications.

Intra-Process Correlation

Using the low-level API, energy information for all or some hardware components are used to calculate the energy consumption at a process level. Process correlation therefore provides per-process energy information classified for each monitored hardware component. This information is then exposed in an API to be used by the high-level application profiling.

High-level profiling

The high-level profiling part is responsible for profiling running applications in real-time and estimating their energy consumption at a finer grain, *e.g.* at a thread and/or methods level. Several profiling techniques can be used, such as bytecode instrumentation or sampling the application. Each of these methods has its advantages and drawbacks. Our approach, however, does not specify a single method of profiling. We prefer to keep this as a technical choice during implementation. Nevertheless, whatever implementation method is chosen, APIs to communicate in the profiler and with the low-level monitoring environment are to be respected.

The profiling part introspects the application at runtime, building 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 energy information. Details on our energy model for the correlation are presented in Section 3.2. Finally, the per-method or per-thread energy consumption information is visualized 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 energy model using our own proposed formulas and formulas taken from the state-of-the-art. In [27], the energy cost of a software component c is computed based on the following formula:

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

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).

In [18], the following model is proposed:

$$E_{App} = E_{Active} + E_{Wait} + E_{Idle} \tag{2}$$

where E_{Active} is the energy cost of running the application and the underlying system software, E_{Wait} is the energy spent in wait states (when a subsystem is powered up while the application is using another), and E_{Idle} is the energy spent while the system is in idle state.

We base our model on a similar principle, taking into account the modular aspect of the energy calculation (*e.g.*, the sum of the energy consumption of different hardware components). We also consider the wait and idle time, but we exclude it from our calculation. This is because, at a finer level (*e.g.*, threads and methods level), current applications are multi-threaded. If a method is waiting for a network packet, another thread is likely to be running and consuming CPU and network energy too. Infrastructure energy is included in the computational cost of our energy models and in our prototype. From this, we can abstract our global energy formulas to the following:

$$E_c = E_{comp} + E_{com} \tag{3}$$

Next, we will detail the energy model we use in e-Surgeon, both at the system intra-process level and at the application profiling method level.

CPU Energy

In order to calculate the energy consumed by the processor, we use the *Thermal Design Power* (TDP). TDP represents the maximum amount of power the cooling system of a computer is required to dissipate the heat produced by the CPU. Manufacturers generally provide the maximum value of the TDP, or TDP_{max}. Then, the following formula can be used to estimate the TDP of the CPU at different frequencies and voltages [9]:

$$TDP = C \times TDP_{max} \times (\frac{freq}{freq_{max}}) \times (\frac{voltage}{voltage_{max}})^2 \tag{4}$$

where *freq* and *voltage* are the current CPU frequency and voltage, and $freq_{max}$ and $voltage_{max}$ are the maximum frequency and voltage that the CPU can support for the provided TDP_{max}. Because the TDP is usually much higher than the real CPU energy consumption, we multiply the equation with a constant factor C equal to 0.7, which is the appropriate value reported by [26]. The resulting consumption is therefore always lower than TDP_{max} [31].

In our experimentations (cf. Section 5), our TDP formula is accurate (compared to values reported by a wattmeter) on a Intel Core 2 Quad processor (4 physical cores), while we observe that the results are not as satisfactory on an Intel Core i7 processor (4 cores including 2 logical cores). As such, we can conclude that the TDP formula, as it is currently reported in the literature, strongly depends on the internals of the processor architecture and is only valid for a specific family of processor. Our goal, in a near future, is to investigate this issue and to define an agnostic formula to compute the processor energy consumption of a wider range of processor architectures.

Using this calculated value, we estimate the energy consumption per-process by basing our model on the CPU time of each process within each CPU frequency. We propose the following formula for calculating the process CPU energy consumption:

$$Energy^{CPU}_{process} = \frac{\sum_{i \in Frequencies} t_i \times TDP_i}{t_{total}} \tag{5}$$

where t_i is the CPU time spend at frequency i, TDP_i is the TDP calculated at frequency i using formula 4, and t_{total} is the total CPU time spend at all supported frequencies.

Network Energy

The network energy of a process is calculated using a formula similar to the CPU energy. We base our model on available information, whether it is collected at runtime or from devices' documentations. As we are targeting application servers, our network model is focusing on ethernet network card. A similar model using a linear equation can be applied for wireless network cards [12], but we did not investigate these options yet.

From the documentations provided by constructors, we obtained the energy consumed (in watt) for transmitting bytes for one second according to a given throughput mode of the network card (*e.g.*, 1 MB, 10 MB...). Our network energy model is therefore defined as:

$$Energy_{process}^{network} = \frac{\sum_{i \in states} t_i \times E_i \times d}{t_{total}} \tag{6}$$

where E_{state} is the energy consumed by the network card in the state i (provided by constructors), d is the duration of the monitoring cycle, and t_{total} is the total time spent in transmitting data with the network card.

CPU Computation

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 calculate the CPU energy consumed per method.

We first calculate the energy consumed per thread. For that, we apply the following formula:

$$Energy_{thread}^{CPU} = \frac{Time_{thread}^{CPU} \times Energy_{process}^{CPU}}{Duration_{cycle}} \tag{7}$$

where $Time_{thread}^{CPU}$ is the CPU time of the thread in the last cycle, $Energy_{process}^{CPU}$ is the energy consumed by the application process in the last cycle, 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 cycle (whether they are still running or not). For each thread, we get its methods from the list. We then estimate with a good accuracy the CPU time for each method using the following formula:

$$Time_{method}^{CPU} = \frac{Duration_{method} \times Time_{thread}^{CPU}}{\sum_{m \in Methods} Duration_m} \tag{8}$$

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

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

$$Energy_{method}^{CPU} = \frac{Time_{method}^{CPU} \times Energy_{thread}^{CPU}}{Duration_{method}} \tag{9}$$

In our code, we also take account of the method call tree in order to limit the computation to the leafs of the call tree. This is done to avoid situations where, for example, a main method is always running, thus consuming energy, while in fact, its invoked methods are the one actually consuming CPU cycles and energy.

Network Computation

The client collects the number of bytes read and written per method from the agent. We, first, calculate the number of bytes read/written in the last cycle (a simple subtraction between the current and the last raw values). Then, we collect the network energy consumed by the application process from our system library. The network energy consumed per method is therefore:

$$Energy_{method}^{Network} = \frac{Bytes_{method} \times Energy_{process}^{Network}}{Bytes_{process}} \tag{10}$$

where $Bytes_{method}$ is the number of bytes read and written by the method, $Energy_{process}^{Network}$ is the energy consumed by the application, and $Byte_{process}$ is the number of bytes read and written by all methods of the application.

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

$$Energy_{thread}^{Network} = \sum Energy_{methods}^{Network} \tag{11}$$

In the next section, we will describe the implementation of our power models and the architecture of e-Surgeon.

4 Implementation

e-Surgeon is implemented as both a system level modular library, and a Java agent instrumenting bytecode at runtime. CPU and network modules as well as profiling functionalities are provided.

4.1 Library

Our system-level library aims to provide energy information per PID [1] for each system component (CPU, NIC[2]...).

The library is therefore based on a modular approach where each system component is represented as an *energy module*. *Energy modules* operate independently of each others 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 [3] container. In particular, we use *Service Oriented Framework* (SOF) [3] to implement the various modules of e-Surgeon in C++.

The sensor sub-module is responsible for gathering hardware and 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 frequencies when it supports *Dynamic Voltage Frequency Scaling* (DVFS). This sub-module is OS-dependent. We implemented sensor sub-modules for the CPU and NIC on a GNU/Linux operating system. In particular, our implementation exploits system informations available in the *procfs* and *sysfs* file systems.

The formula sub-module, on the other hand, is platform independent. The sub-module is responsible for calculating the energy consumed for each process by using information gathered by the sensor sub-module.

Additionally, our library supports the lifecycle management of its energy 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.

From the application point of view, the library communications are achieved by using the *formula*'s interfaces. These interfaces follow the following pattern:

$$TYPE - Energy : get - TYPE - Energy(PID, d) \tag{12}$$

Where $TYPE$ is the system component type to monitor (*e.g.*, CPU, NIC), PID is the process ID and d is the monitoring cycle duration. For example, for the CPU, the pattern indicates that a method with the following signature offers the energy consumed by the PID for a duration d: `CPUEnergy: getCPUEnergy(PID, d)`. CPU monitoring for a given PID and duration can be described as the sequence diagram depicted in Figure 2.

When a `getCPUEnergy(PID, duration)` request is sent, `CPUFormulaService` asks a `CPUFormulaComputer` (associated to a specified `PID`) to start the computation. In collaboration with the `CPUSensorService`, the `CPUFormulaComputer` produces a `CPUEnergy` report when computation is completed (every `duration` time).

[1]Process IDentifier
[2]Network Integrated Card
[3]formerly Open Service Gateway Initiative

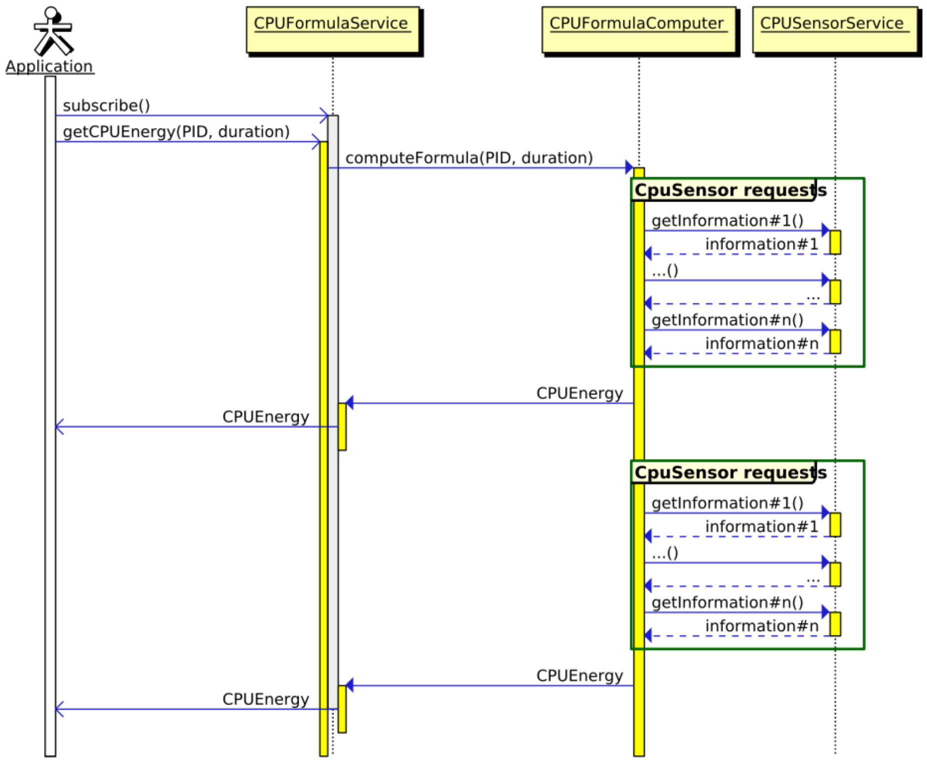

Figure 2: Library sequence diagram.

4.2 Application Energy Monitoring

We also develop a real-time application energy monitoring component, called Jalen, as part of the e-Surgeon architecture. Jalen uses the per-process energy information provided by our system library, and correlates it with information collected from the application monitoring in order to provide per-method energy information. Jalen is designed as a two-parts architecture: *i)* a Java agent using bytecode instrumentation for monitoring the application, and *ii)* a JMX client computing and correlating the collected data. This client extracts the per-method energy consumption values. Figure 3 overviews the architecture of Jalen.

We use bytecode instrumentation technics [8] to inject our monitoring code into the methods of legacy applications. We choose instrumentation rather than sampling for two main reasons: *i) accuracy* as we require fresh energy values; and *ii) repeatability* as bytecode instrumentation produces similar results with the same environment and parameters. Although bytecode instrumentation has a non-negligible overhead for very large Java applications, we argue that supporting precise and accurate per-method energy profiling is better-suited for diagnosing energy leaks in applications. As our solution is expected to be used in a tested environment, we expect that, once optimized, the Java applications would not need to be instrumented when deployed in production.

Nonetheless, in order to reduce this overhead, we migrate all the computation to a remote JMX client. Therefore, our Java agent only collects raw values

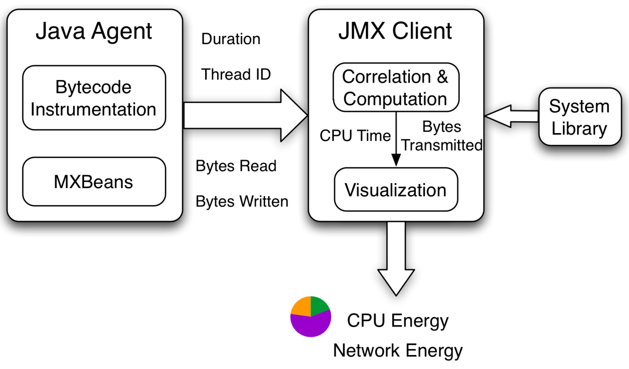

Figure 3: The Jalen Architecture.

(*e.g.*, method start time, thread ID). We instrumente the Java bytecode for collecting CPU and network usages information. For the CPU, we collect the following metrics per method: 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 class to route calls from the class `SocketImpl` 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 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, thus the need for separation.

Then, we build a JMX client that, not only display the collected information, but also do the computation and correlation in order to determine the CPU and network energy consumed per method in each monitoring cycle. The JMX client collects information exposed by the agent's MXBean. Using energy models and metrics correlations (cf. Section 3.2), we estimate the CPU energy and the network energy consumed per method. Finally, a graphical interface shows the energy consumption per methods as a real-time updated pie chart and table. Figure 4 portrays the graphical interface of the Jalen client. The figure shows the energy consumption of Tomcat 7 during its execution. Displayed values refer to the last monitoring cycle (here it is 1 second).

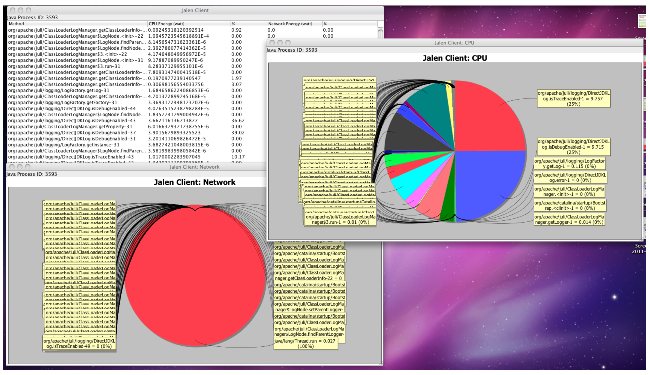

Figure 4: The Jalen Client's Graphical Interface.

5 Experimentation

We run our e-Sugeon prototype on a Dell Precision T3400 workstation with an Intel Core 2 Quad processor (Q6600), running Ubuntu Linux 11.04. We first evaluate the accuracy of our library, and then estimate the overhead of our Java agent. Based on these results, we conduct an analysis of a stress benchmark on a Jetty Web Server [17] in Section 5.3. Our preliminary experimentations shows that the consumed network energy is largely negligible compared to the consumed CPU energy on our test server. This observation is in correlation with related research [26]. Therefore, we will only outline the results of our CPU experimentations. Note that the network energy can have a higher impact on the global energy consumption in a different experimentation set, such as a mobile phone. However, we limit our experimentations on a workstation machine.

5.1 Accuracy

We first assess the accuracy of the results provided by our system library. We compared these values with the actual energy consumption of the computer using a powermeter.

First, we stress the processor using the Linux stress command [20]. Figure 5 depicts the results as an evolution of the CPU energy consumption during time (normalized values). The peaks corresponds to stressing 1, 2, 3, 4, and then 1 and 2 cores, respectively. Then, we compare the values of our library and the powermeter in three additional scenarios: running a video using MPlayer [2] (results in Figure 6), the SunSpider JavaScript benchmark [4] running on Firefox (cf. Fgure 7), a stress test on Tomcat using Apache JMeter [1] (cf. Figure 8), and on Jetty (cf. Figure 9).

The results show minor variations between our calculated values in the library, and the real energy consumption values. Therefore, we can reasonably argue that using software-only approach, we manage to provide values that are accurate enough to be used in energy management software.

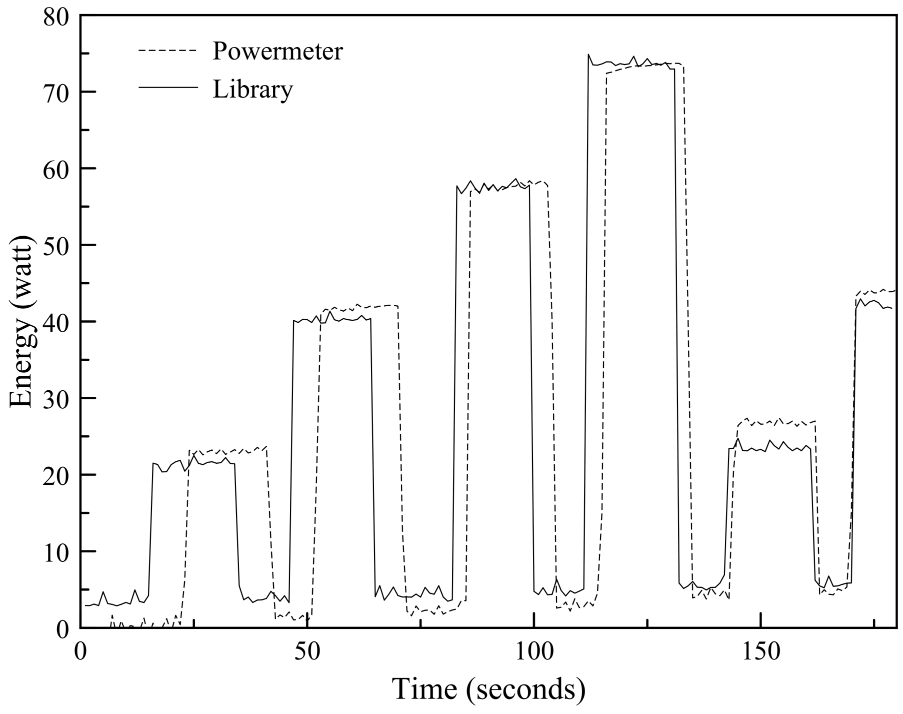

Figure 5: Stressing the processor cores.

5.2 Overhead

We calculate the CPU overhead of our Jalen java agent using several programs: a CPU-intensive application, for example a recursive version of the Towers of Hanoi program, and a CPU and network-intensive application: the Tomcat application server. We develop, in addition to our agent, a thread-only version of the agent. This version uses the same methods delegations for calculating the network energy, however it does only calculate the CPU energy consumption per thread. In this case, no bytecode instrumentation is needed. We first calculate the overhead of our thread-only agent that does not use bytecode instrumentation. Then, we use our methods-level agent in order to estimate the overhead of the bytecode instrumentation, thus the overhead of our full agent.

Thread-only agent

We first calculate the CPU overhead of our agent on a CPU-intensive application. We iterate 50 times the program with and without injecting our thread-only version of our Jalen agent. On each loop, we set the number of towers to 10. On average, the program takes 22.64 ms to execute, while it took 22.74 ms when we added our agent. The average overhead is therefore negligible at 1.79%.

We also calculate the overhead of our thread-only agent on Tomcat 7 application server. Results of 10 executions show that our bytecode instrumentation

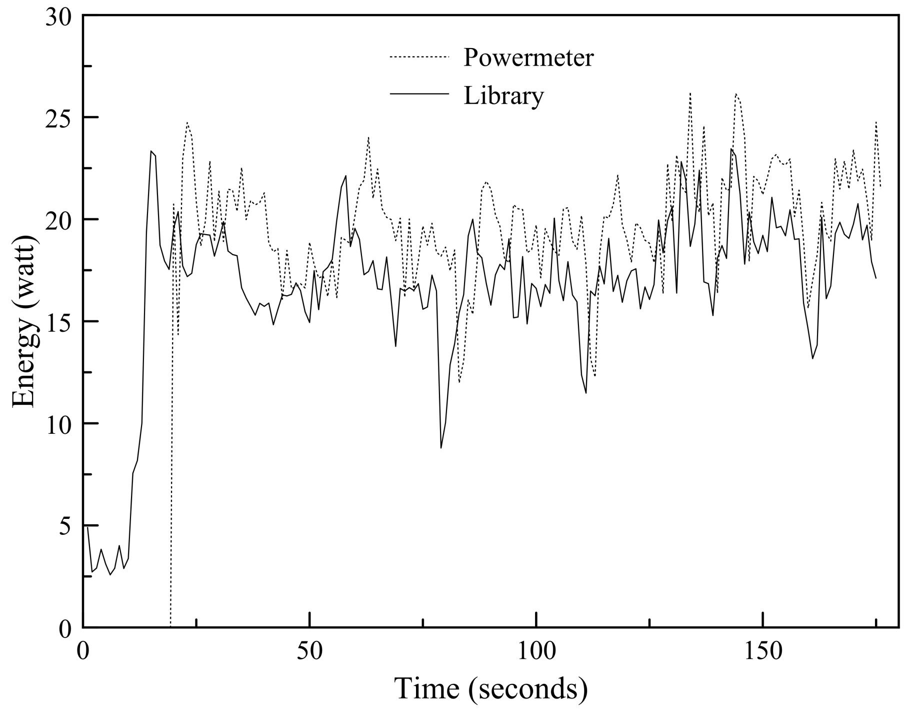

Figure 6: Running a video under MPlayer.

has practically no impact on the server. On average, we even have better results when running Tomcat with our agent than without it (716 ms compared to 724 ms!). However, when comparing server startup time on each loop, we find that our agent does not have any real impact. The overhead was alternating with some executions being more efficient when running the agent, while other was worst.

Methods-level agent

However, when running the overhead benchmarks using our methods-level version of our Jalen agent, the observed overhead was more consequent. The Towers of Hanoi program took 115.92 ms to execute compared to 24.14 ms without the agent, resulting in an overhead of 380.12%. This high overhead can be explained by the usage of thread-safe maps and lists, in particular the usage of `CopyOnWriteArrayList`, which is responsible for nearly half of the overall overhead (the overhead dropped 40% when we used an `ArrayList` instead).

We also test our agent with Tomcat 7 application server. On 5 loops, our agent has an overhead of 246.41%. Tomcat needed in average 878.2 ms to start without the agent, while this number grow up to 3042.2 ms on average.

Even though these numbers appear to be high, they should be leveraged by what similar profilers provide. For example, *Java Interactive Profiler* (JIP) is reported as having a *very low overhead* [6]. But still, we calculate an average

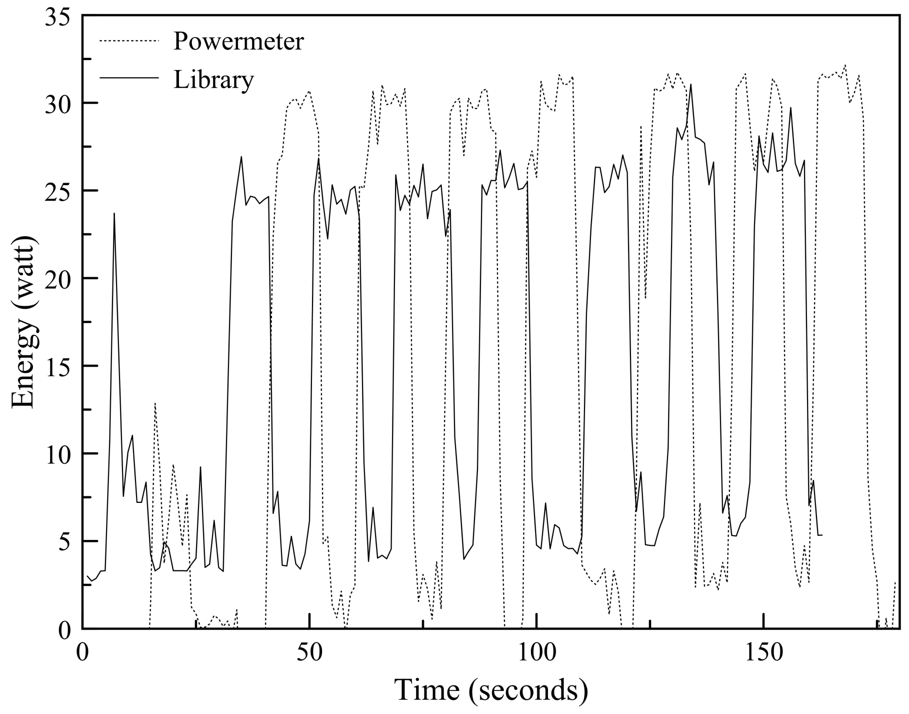

Figure 7: Running the SunSpider JavaScript benchmark.

overhead of 167.46% (with an average execution time of 64.97 ms) with the Towers of Hanoi program with 10 towers. We also calculated the overhead of the VisualVM Profiler [29]: 84.85% (with an average time of 49.93 ms), and the overhead of the Oktech Profiler [23]: 55.5% (41.22 ms). Figure 10 summarizes these numbers. Our code is therefore twice slower than the available tools but we can provide, at runtime and in real-time, detailed information of the energy consumption of methods.

5.3 Jetty Web Server Experimentation

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 11. The graph portrays the top 10 most energy-consuming methods in the X axis (out of 726 instrumented methods). The right Y axis (thus the bars) represents the energy consumed during our execution in percentage of the total energy consumed. The left Y axis (thus the line) represents the number of invocation of the method. We run this experiment several times, and although we had difference in the watt energy values, we notices that the global and proportional percentage is stable. Note also that the provided values are

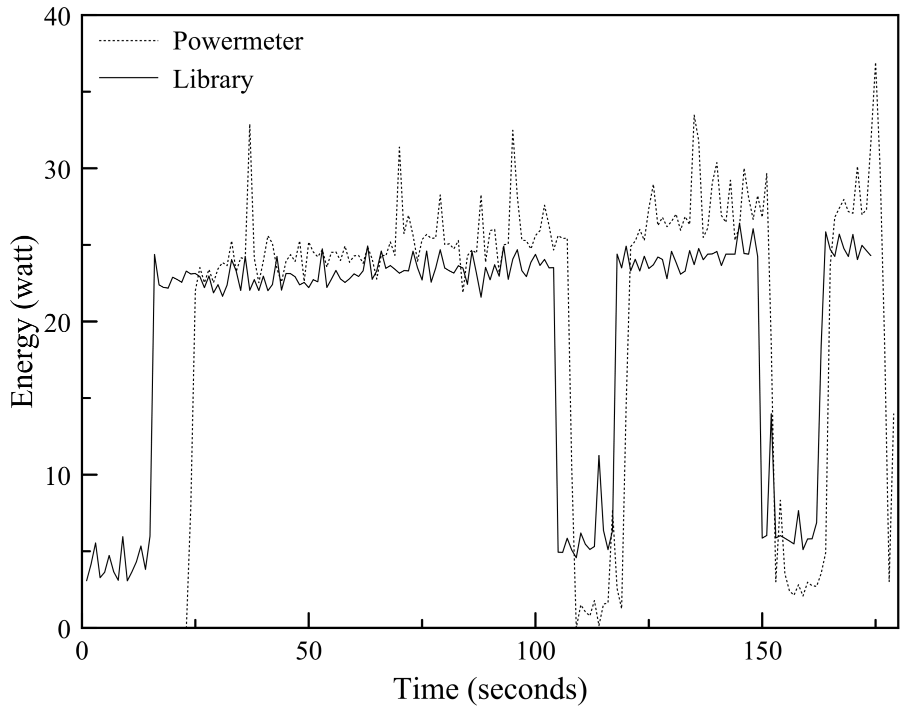

Figure 8: Running stress tests on Tomcat using JMeter.

an aggregation of the execution the methods on all threads.

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

We also analyzed the energy performance of the top 10 methods. We calculate the energy consumption per method invocation. The results are presented in Figure 12. We observe that `org/eclipse/jetty/io/AbstractBuffer.putIndex` has a better performance per invocation. This method consumed 2.89% of total energy during the tests, but was invoced 42293 times (the most in the top 10). Thus, this method has a energy per invocation of 5.81% of the top 10. On the other hand, `org/eclipse/jetty/io/ByteArrayBuffer.get` has a performance of 23.18% with 52159 invocations and 14.22% of global energy consumed.

The results for the top 10 most consuming classes (out of 146) are presented in Figure 13. We note that the 2 classes (`org/eclipse/jetty/io/AbstractBuffer` with 39.22% and `org/eclipse/jetty/io/ByteArrayBuffer` with 23.89%) consumes alone more than 60% of the total energy (63.11%).

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 energy footprint of this method. By keeping track of

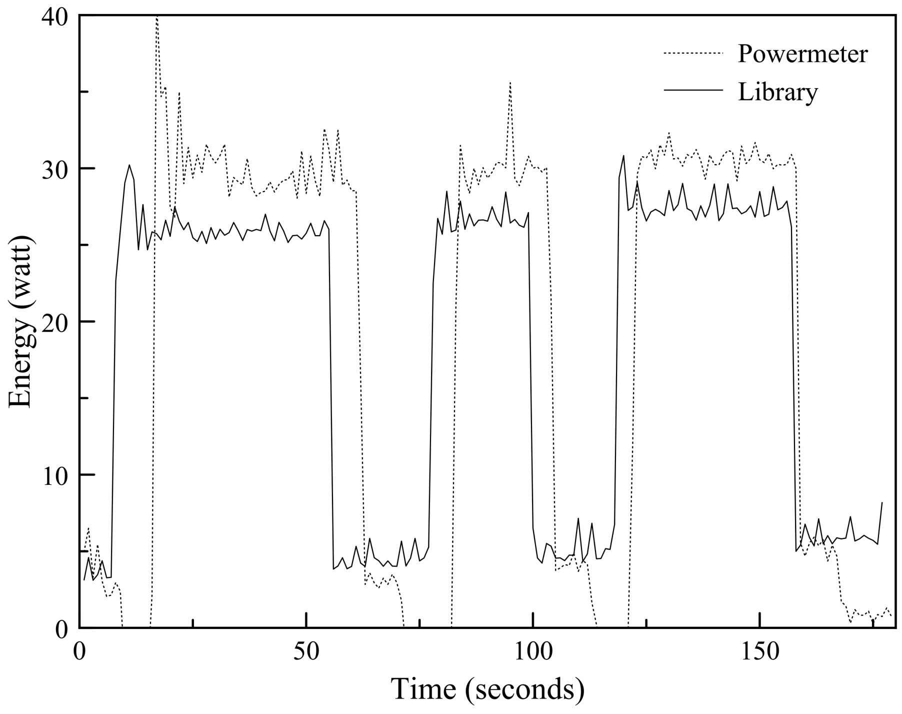

Figure 9: Running stress tests on Jetty using JMeter.

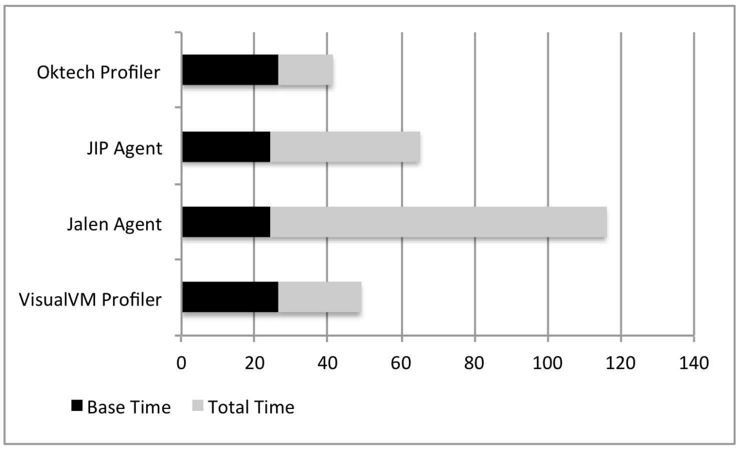

Figure 10: CPU overhead of instrumentation tools with Towers of Hanoi program.

the energy footprint of classes and methods,we think that coding completion systems (like the one available in Eclipse) could be extended to recommend the developer to use energy-efficient implementation of standard classes (*e.g.*, `List`, `Set`, `Map`).

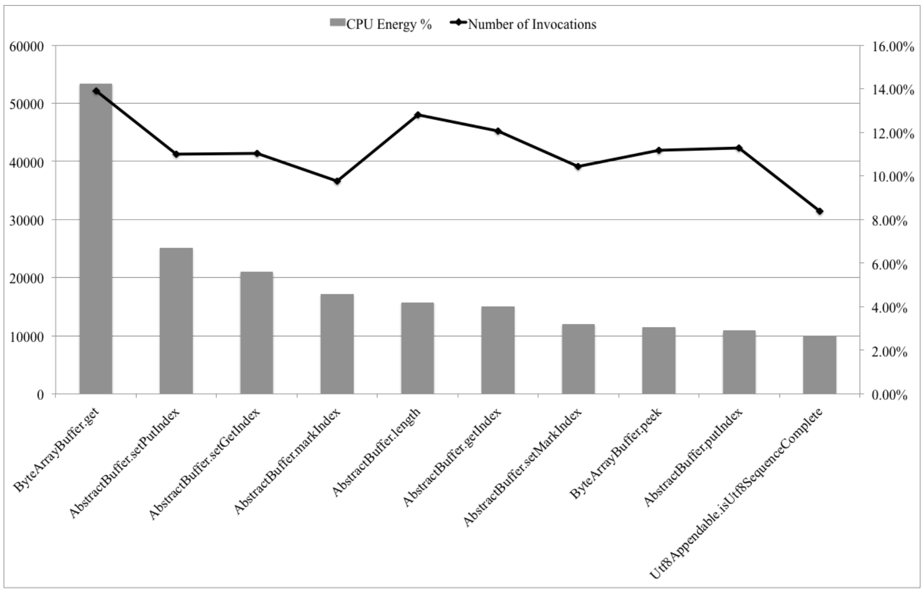

Figure 11: Cumulated energy consumption of Jetty methods under JMeter stresses (top 10 most energy-consuming methods).

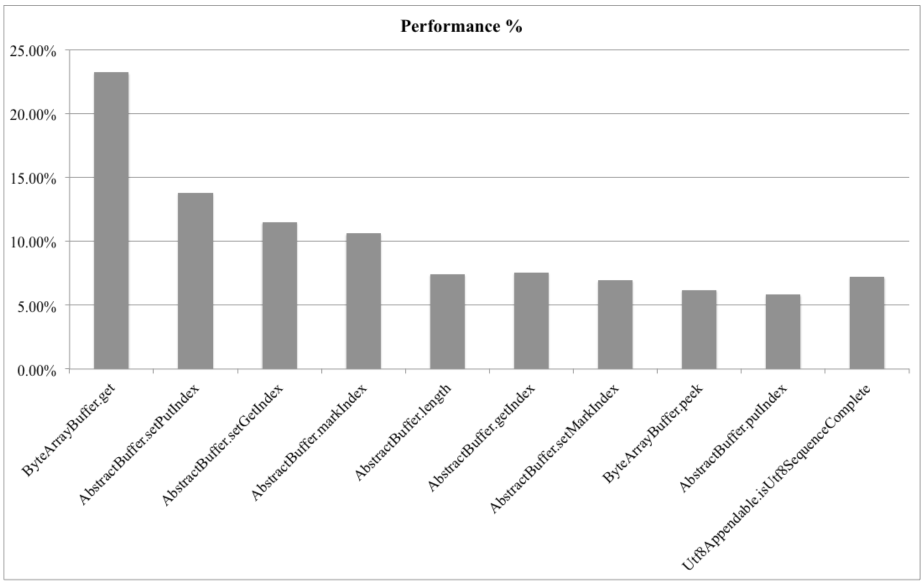

Figure 12: Methods energy performance in percentage of the top 10 most energy-consuming methods (lower is better).

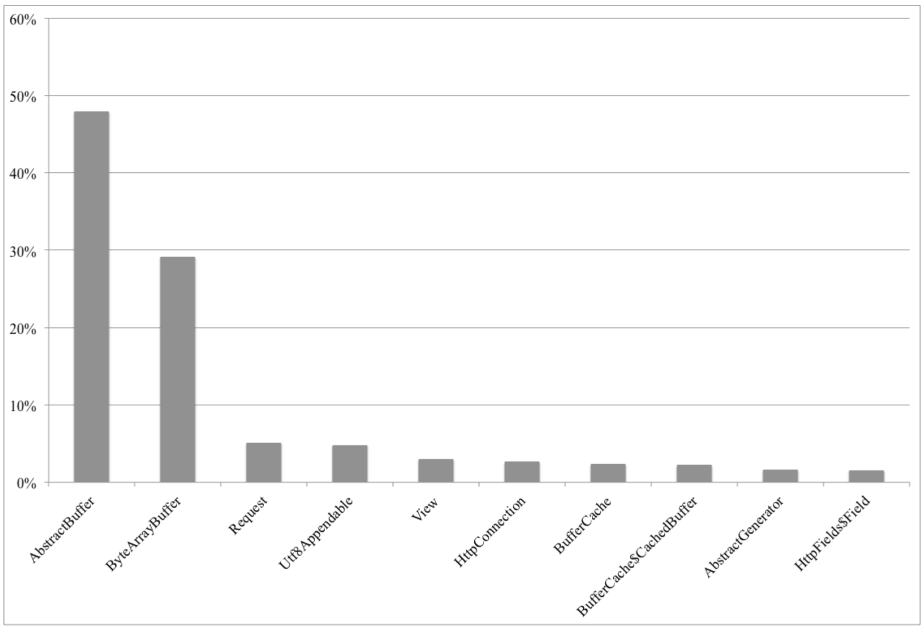

Figure 13: Cumulated energy consumption of Jetty classes under JMeter stresses (top 10 most energy-consuming classes).

6 Related Works

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 [22], 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. Devices, such as AlertMe Smart Energy [7], monitor home devices and allow users to visualize their energy consumption history through application services, such as the now defunct Google Powermeter [16]. However, these approaches do not adapt the system autonomously: the user takes the decision, while our approach opens up solutions for adapting the system with limited user intervention.

The previous monitoring approaches allow getting 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 formulas as in [27] and [18]. In [27], the authors propose formulas 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. In [18], 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). In [13], 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 precise than energy estimation, although it still needs a hardware investment.

In [24], Petre presents an energy model using the middleware language MIDAS [25] that classifies energy as data, code, or computation unit resources. Computation units are distinguished into 3 types: *software*, *hardware*, and *electrical sockets*. However, this model is too generic for providing energy values. It is rather used as an energy-aware add-on to the MIDAS platform.

System Level Tools

pTop [10] is a process-level power profiling tool. Similar to the Linux *top* program [21], 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. A Windows version is also available, so called *pTopW*, and offers similar functionalities, but using Windows APIs.

In addition to *pTop*, several utilities exist on Linux for resource profiling. For example, *cpufrequtils* [28], 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* [19] that is used to get devices' and partitions' input/output (I/O) performance information, as well as CPU statistics. Other utilities [15] 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 energy information.

Our approach is more flexible and fine-grained than *pTop*. Not only we offer process-level energy information, but we also go deep into the application in order to profile and report thread and method-level energy 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. Our flexible and modular approach therefore offers these functionalities, and extends them to not only OS processes, but also to Java-based applications profiling.

Application Profiling Tools

Several open-source or commercial Java profiling tools already propose some statistics of Java applications. Tools, such as VisualVM [29], *Java Interactive Profiler* (JIP) [6], JProfiler [11], or the Oktech Profiler [23], offer coarse-grained information on the application and fine-grained resource utilization statistics. However, they fail in providing energy 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 real-time 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 energy consumed. On the other hand, these tools have a smaller overhead than our prototype. This is due, in part, to source code optimizations made into these tools by dozens of developers in several years. Nevertheless, our approach can be used to extend these tools with our energy models.

7 Conclusion and Future Works

In this paper, we present the e-Surgeon architecture. It allows to gather and calculate the energy consumption at threads and methods level. Its modular architecture allows real-time context-based adaptations of the monitoring environment itself, leveraging performance and accuracy at the wish of the application or the user. We also propose energy models to calculate the energy consumption. Our models use and extend the state-of-the-art models and formulas, and port them to fine-grained context. Our initial results show the potential of our approach for diagnosing, at runtime, energy leaks of Java-based applications. As for future work, we plan to: *i)* propose more energy models for other hardware resources (in particular, memory and disk); *ii)* optimize our implementation prototype, in particular the overhead of the methods-level version; and *iii)* extend our experimentation to more applications, such as the Eclipse platform and the JOnAS Java EE Application Server.

References

[1] Apache JMeter. http://jmeter.apache.org.

[2] MPlayer. http://www.mplayerhq.hu.

[3] Service Oriented Framework. http://sof.tiddlyspot.com.

[4] SunSpider JavaScript Benchmark. http://www.webkit.org/perf/sunspider/sunspider.html.

[5] The Green Challenge for USI 2010. https://sites.google.com/a/octo.com/green-challenge.

[6] The Java Interactive Profiler. http://jiprof.sourceforge.net.

[7] AlertMe. http://www.alertme.com/smart_energy.

[8] ASM. http://asm.ow2.org.

[9] Benchtest.com. Temperature, Watts and C/W Calculators. http://benchtest.com/calc.html.

[10] Thanh Do, Suhib Rawshdeh, and Weisong 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.

[11] ej-techonologies. JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html.

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

[13] Jason 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.

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

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

[16] Google Powermeter. http://www.google.com/powermeter.

[17] Jetty Web Server. http://www.eclipse.org/jetty.

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

[19] Linux User's Manual. iostat. http://linux.die.net/man/1/iostat.

[20] Linux User's Manual. stress. http://linux.die.net/man/1/stress.

[21] Linux User's Manual. top. http://linux.die.net/man/1/top.

[22] Dustin McIntire, Thanos Stathopoulos, and William 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.

[23] OKTECH-Info Kft. OKTECH Profiler. http://code.google.com/p/oktech-profiler.

[24] Luigia Petre. Energy-Aware Middleware. In *ECBS'08: Proceedings of the 15th Annual International Conference and Workshop on the Engineering of Computer Based Systems*, pages 326–334. IEEE, 2008.

[25] Luigia Petre, Kaisa Sere, and Marina Waldén. A Language for Modeling Network Availability. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 639–659. Springer Berlin, Heidelberg, 2006.

[26] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos 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.

[27] Chiyoung Seo, Sam Malek, and Nenad 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.

[28] The Linux Kernel. cpufrequtils. http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufrequtils.html.

[29] VisualVM. http://visualvm.java.net.

[30] Molly 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.

[31] CPU World. Therman Design Power (TDP). http://www.cpu-world.com/Glossary/T/Thermal_Design_Power_(TDP).html.

Contents

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399