Monitoring energy hotspots in software

Noname manuscript No.
(will be inserted by the editor)

Monitoring Energy Hotspots in Software

Energy Profiling of Software Code

Adel Noureddine · Romain Rouvoy · Lionel Seinturier

Received: date / Accepted: date

Abstract Green IT 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. In this paper, we report on a fine-grained runtime energy monitoring framework we developed to help developers to diagnose energy hotspots with a better accuracy.

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 and monitor their variations under stress scenarios.

Keywords Power Model, Power Monitoring, Energy Consumption

Adel Noureddine
INRIA Lille – Nord Europe, University Lille 1 - LIFL CNRS UMR 8022, France E-mail: adel.noureddine@inria.fr

Romain Rouvoy
INRIA Lille – Nord Europe, University Lille 1 - LIFL CNRS UMR 8022, France E-mail: romain.rouvoy@inria.fr

Lionel Seinturier
INRIA Lille – Nord Europe, University Lille 1 - LIFL CNRS UMR 8022, Institut Universitaire de France, France E-mail: lionel.seinturier@inria.fr

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. Predictions for the next 20 years outlook a global rise of energy costs [18], in addition to an expected increase of the carbon footprint of Information and Communications Technology (ICT) in 2020 [22]. Although ICT helps in reducing the energy footprint of other sectors (such as with transportation or in buildings) [22], its power consumption is predicted to rise and double from 168 Gigawatt (GW, or around 7%) to 433 GW (or more than 14.5%) of worldwide power consumption in 2020 [21]. These numbers illustrate the opportunities for efficient ICT solutions to reduce energy consumption, and therefore help in reducing carbon emissions. Rising energy costs in computers and mobile devices require the optimization and the adaptation of computer systems. In this domain, research in Green IT 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 [11], or only offer coarse-grained energy estimation feedback of software [7, 5].

In this article, 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 (at the software level), 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 byte code instrumentation or statistical sampling (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[1]. Our preliminary results demonstrate that we can diagnose power hotspots of Java-based applications at runtime, offering opportunities to reduce their power consumption.

Our work extends our previous propositions and experimentations in measuring the impact of programming languages and algorithms in [12] with a finer grain approach down to the code level. And also, it extends our work on code level monitoring in [13] with new implementations that yields the limitations of our previous approach. In addition, we propose an extension of

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

our preliminary approaches and results on modeling the energy consumption evolution of software code based on input parameters [14].

The remainder of this article is organized as follows. In Section 2, we describe our motivations and the main challenges we tackle. Section 3 describes the formulas and power models we propose to estimate the energy consumption of software. Section 4 describes our approach, the design of our proposed architecture and its implementation. 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. We discuss the results of our experimentations in Section 7, while we discuss inferring the energy variation model of software code in Section 8. Futures directions are discussed in Section 9, while related work is discussed in Section 10. Finally we conclude in Section 11.

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 [11,19,9]. 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 [2] 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 of energy is usually achieved through additional hardware measurement equipment, such as multimeters or specialized integrated circuits (see Section 10). 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 tradeoff between the accuracy requirements and the cost of the software monitoring tool.
- **Fine-grained.** Many of the current approaches (see Section 10) 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 are 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 (see Section 10). 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.
- **Hardware and Software relation.** Energy, in the physical world, is actually consumed by hardware. We consider the energy consumption of software as *the energy consumed by hardware following a request of software*. In particular, if an application requires a certain number of CPU cycles for computing a task, then the energy consumed by the CPU for these cycles is considered the energy consumed by the application. Challenges arise when hardware properties impact energy consumption unrelated to software. For example, if some data were stored in a file or in the RAM memory, then the CPU would require different amount of time for executing the same task, therefore consuming more energy. Other factors also impact energy, such as temperature, hardware characteristics (*e.g.*, silicon capacitance, material imperfections), that go beyond our scope of computer science and into the scope of physics.

Laying these challenges, we propose in the next sections an approach named E-Surgeon, for monitoring and profiling applications at runtime. In particular, we start with describing our power models, then the details of our reference architecture and its implementation.

3 Power Models

We propose a comprehensive power model using our proposed formulae as well as formulae taken from the state-of-the-art. In [19], the energy cost of a software is computed based on the following formula:

$$E_{software} = E_{comp} + E_{com} + E_{infra} \quad (1)$$

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:

$$P_{software} = P_{comp} + P_{com} \quad (2)$$

At this stage, we define 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.

3.1 PowerAPI Power Models

CPU Model

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

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

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[2]) follows the standard equation [15]:

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

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 [17]:

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

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 frequency. 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

[2] Complementary Metal Oxide Semiconductor

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[3]. Thus, the CPU power consumption in a duration d and for a given frequency f, P_{comp}^{f} of formula (2) is equal to :

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

When the processor supports Dynamic Voltage and Frequency Scaling (DVFS), the CPU power consumption for a process P_{comp} is equal to the average of the CPU power of each frequency balanced by the CPU time of all frequencies:

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

3.1.1 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 [6], 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.

3.2 Jalen Power Models

3.2.1 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

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

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:

$$Power_{method}^{CPU} = \frac{Time_{method}^{CPU} \times Power_{thread}^{CPU}}{Duration_{cycle}} \tag{12}$$

3.2.2 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:

$$Power_{method}^{Network} = \frac{Bytes_{method} \times Power_{process}^{Network}}{Bytes_{process}} \tag{13}$$

Where $Bytes_{method}$ is the number of bytes read and written by the method, $Power_{process}^{Network}$ 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:

$$Power_{thread}^{Network} = \sum Power_{methods}^{Network} \tag{14}$$

In the next section, we describe the architecture and the implementation of e-Surgeon.

4 e-Surgeon Design and Approach

In this section, we present our power monitoring approach called e-Surgeon. The e-Surgeon architecture is composed of 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 monitoring methodology of our approach, where the order of each step of the approach is numerically marked in the figure. Blue boxes represent the existing environment, such as applications and hardware, while the green ones represent data and models we use to estimate energy consumption. The remaining boxes represent the each of the steps of our architecture. These steps are implemented in our parts, PowerAPI and Jalen. In details, we start first by monitoring the hardware resources utilization (step 1 in Figure 1), for example, we monitor the actual frequency and voltage of the CPU. In step 2, we use the collected information in step 1 along with constructors documentations about hardware in order to estimate the energy consumption of hardware, using our energy models. In step 3, we apply a similar monitoring strategy as in step 1 in order to monitor resources utilization by applications. Then, step 4 estimates the energy consumption of software using our energy models and the energy consumption calculated at step 2. Finally, we monitor resources utilization at source code level in step 5, and estimate their energy consumption in step 6 using our energy models and the energy consumption calculated at step 4.

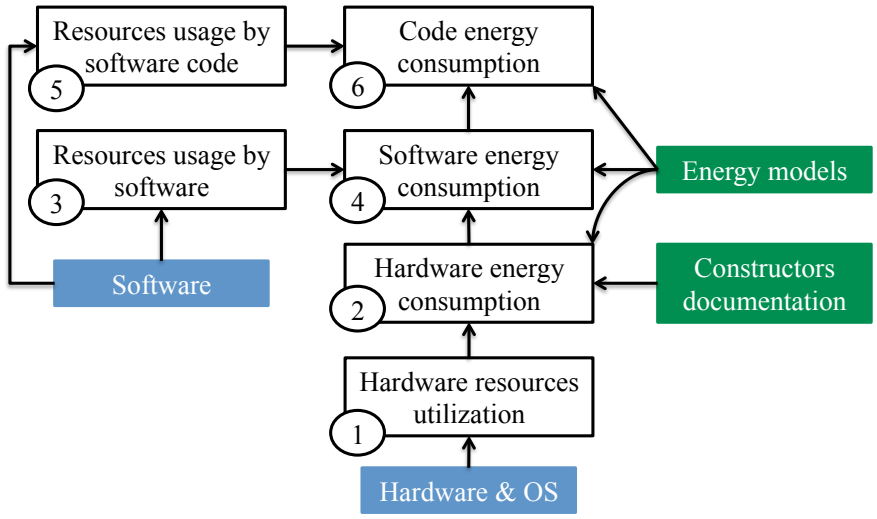

Fig. 1 e-Surgeon monitoring methodology.

4.1 PowerAPI

4.1.1 Architecture

PowerAPI is a system library providing a programming interface (API) to monitor at runtime the power consumption of software at the granularity of system processes. Each process can therefore be monitored for its power consumption with a good estimation in comparison to using hardware power meters. PowerAPI implements steps 1 to 4 in Figure 1. The library also offers energy differentiation values based on hardware resources, such as giving the energy consumed by the process on the CPU, or on the network or on other supported hardware resources. In our approach, we consider an application as a process, therefore if an application is executed in two processes, then the total power consumption of the application is the sum of these two processes. This additional calculation can be done manually, we prefer in our approach to handle processes as independent entities. Note that if a process is using system calls, then the power consumed by these calls is included to the power of the process only if the operating system executes them in the same monitored process. If these calls are executed in a separate process, then they are not included in the results of the main monitored process. In particular, if the application `A` calls a method `sysCall` that is part of the operating system's available API, then the energy consumed by `sysCall` is computed as being consumed by the application `A`, if and only if the OS executes this system call in the same process as the application.

PowerAPI's architecture is modular as each of its components is represented as a *power module* (see Figure 2). It uses an event bus with the publish/subscribe paradigm in order for modules to communicate. First, sensors modules (*e.g.*, S_{CPU}, $S_{Network}$) are responsible for gathering operating system and hardware related information. For example, the CPU sensor gathers the time spent by the CPU, for the monitored process, at each of the processor frequencies (when DVFS is supported). This information is then published to all listening modules in the event bus.

The second step is where formulas modules listen to the published sensors data, and estimate the power consumed for the monitored process. Formulas modules use hardware characteristics (such as the CPU voltage for each frequency, which are often provided by hardware constructors) in the calculation as seen in Section 3.

Then, listeners modules listen to the event bus and gather the published data of formulas modules. A *CPU File* listener module is therefore responsible for writing to a file the CPU power consumption of the monitored process. The *All Graphic* listener displays a graph that is updated with the power consumption of the monitored process for all supported hardware components (*e.g.*, CPU, Network, etc.). The API allows users to query the different modules and get information about sensors, formulas, or directly get the energy consumption through listeners. Next, we describe the implementation of PowerAPI.

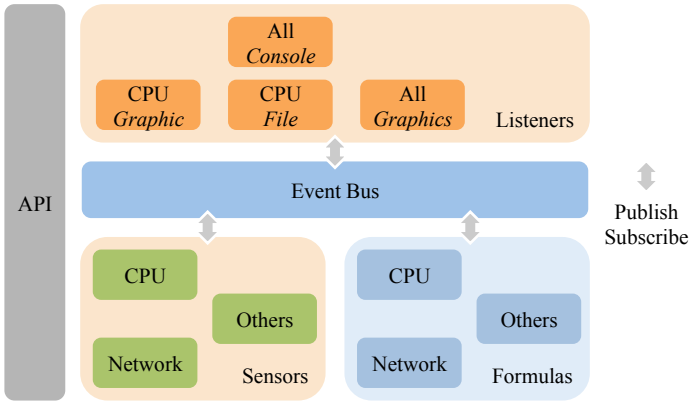

Fig. 2 PowerAPI Reference Architecture. Each group of modules is displayed in a different color for clarity.

4.1.2 Implementation

PowerAPI is implemented in Scala [4] and is based on an event-driven architecture. PowerAPI is based on a modular and asynchronous event-driven architecture using the Akka library [5]. Its architecture is centralized around a common event bus where each module can publish/subscribe to sending events. One particularity of this architecture is that each module is in passive state and reacts to events sent by the common event bus. One objective of PowerAPI is to provide a simple and efficient way to estimate the energy consumption of a given process. Simple, because API is close to the user requirement, and efficient, because the library is an actor-based framework in which the user builds the library by choosing modules to consider for the user's particular requirements. PowerAPI is thus limited to the user's needs, avoiding any extra computational cost.

PowerAPI can be used either as a standalone application (for example, running the program in a terminal), or as an API in other software. In the latter case, the application loads and starts modules on demand. For example, a Java application requiring monitoring the power consumption of a process for the CPU component will load the CPU sensor and formula modules as provided in the following listing:

```
PowerAPI.startEnergyModule(powerapi.formula.cpu.dvfs.CpuFormula.class);
```

Then, the application asks PowerAPI to start monitoring the process by providing its Process ID (PID) as follows:

```
PowerAPI.startMonitoring(new powerapi.core.Process(PID),
        Duration.create(500, "milliseconds"), null,
        powerapi.listener.cpu.file.CpuListener.class);
```

[4] http://www.scala-lang.org/

[5] http://akka.io/

PowerAPI will then load all modules, and starts monitoring the process *PID* while writing to a file the power values each 500 milliseconds.

We provide a modular library where only parts 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

4.2.1 Architecture

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. Jalen implements steps 5 and 6 in Figure 1. Several profiling techniques can be used, such as byte code instrumentation or sampling the application. Each of these methods has benefits and drawbacks. Statistical sampling 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. Instrumentation injects profiling code into the application's code (or byte code), 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. This overhead depends on the added code and may be constant or may vary between executions (*e.g.*, whether the added code execute method related tasks or not, such as filling a map with data). 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

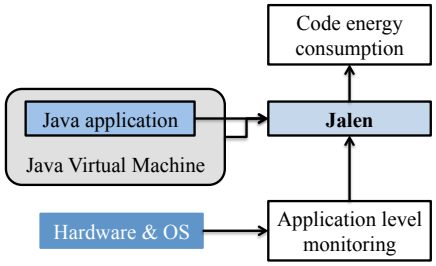

Fig. 3 The Jalen Architecture.

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. 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). Figure 3 depicts the JALEN's architecture for Java applications.

4.2.2 Implementation

We implement JALEN in the Java programming language, and as part of the E-SURGEON architecture. The availability of a virtual machine helps also in retrieving vital information for our model. Therefore, we decide to take advantage of what a virtual machine can offer in term of specifications (in particular, accurate CPU utilization of threads, a middle layer between software code and hardware, byte code instrumentation and injection, etc.).

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 implemented as a Java agent that hooks to the Java Virtual Machine during its start, and starts monitoring and collecting energy related information of the monitored application. We develop two versions, each with different approaches on collecting and correlating information. The first version uses byte code instrumentation, while the second uses statistical sampling. Each holds advantages and disadvantages when monitoring energy consumption of software, and can be better applied than the other in certain contexts. Next, we explain in details each of these implementations.

4.2.3 Instrumentation Version

The instrumentation version of JALEN uses byte code instrumentation techniques, in order to collect resources usage information. In particular, we use ASM [10, 1] to inject monitoring code into the methods of legacy applications. ASM [6] is a *Java byte code manipulation and analysis framework*. The instrumentation process goes as follows and is described in Figure 4.

Byte code injection

First, we inject monitoring code at the beginning and the end of each instrumented method. The latters are instrumented based on their name, class, package or other characteristics such as their number of parameters. This filtering is specified in the settings of JALEN agent.

[6] ASM name is a reference to the asm keyword in C, and does not mean anything [10].

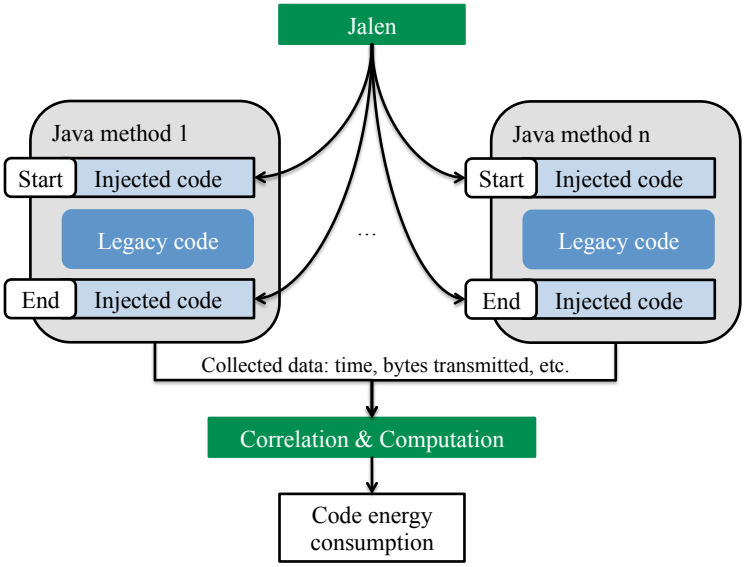

Fig. 4 The JALEN instrumentation implementation.

The injection can either be at runtime, where the JALEN agent injects the additional code when a class is first loaded; or offline, where a special tool is used to inject code to the .class files of the program. The first approach is therefore an agent that instruments byte code at runtime and estimates the energy consumption, while the second approach is composed of two tools: an application that instruments byte code offline, and an agent that estimates the energy consumption at runtime. Both versions inject the same code and provide the same calculations. The main advantage of the second approach is its reduced overhead (as the instrumentation is done offline), and its ability to instruments all classes including the ones loaded with a different class loader at runtime (in which case, the first approach, as implemented, fails to instruments them).

The additional code has a constant overhead because the code performs similar tasks for all methods. In particular, it collects the methods name, its execution time based on injected counters, and it's position on the call tree. An example of byte code instrumentation is shown in the following listing for the `solveHanoi` method in a Towers of Hanoi program:

```
public static void solveHanoi(int arg0, char arg1, char arg2,
        char arg3, PrintStream arg4) {
    MethodStats.onMethodEntry(3, "/TowersOfHanoi", "solveHanoi");
    if (disks >= 1) {
      solveHanoi(disks - 1, fromPole, withPole, toPole, ps);
      moveDisk(fromPole, toPole, ps);
      solveHanoi(disks - 1, withPole, toPole, fromPole, ps);
    }
    MethodStats.onMethodExit(3);
}
```

The source code for the two added method calls, `onMethodEntry` and `onMethodExist` is provided in the following listing:

```
public static void onMethodEntry(int id, String className,
         String methodName) {
        StringBuilder fullMethodName = new StringBuilder(className)
        .append('.').append(methodName).append('-')
        .append(Thread.currentThread().getId());
        MethodStats.addNewMethod(id, fullMethodName.toString());
        if (depth < MAX_CALL_DEPTH) {
                startTimes[depth] = System.nanoTime();
                stack[depth++] = MethodStats.getMethodInfo(id);
        }
}

public static void onMethodExit(int id) {
        if (depth == 0)
                return;
        depth--;
        MethodInfo mi = stack[depth];
        if (mi == null)
                return;
        Long executionTime = System.nanoTime() - startTimes[depth];
        mi.nbCalls++;
        mi.allTime += executionTime;
        if (depth > 0) {
                MethodInfo mip = stack[depth - 1];
                if (mip == null)
                        return;
                mip.addChildTime(mi, executionTime);
                mi.netTime = mi.allTime - mi.childrenTotalTime;
        }
}
```

Collected information

The code injected at the beginning of methods collects information such as the full name of the method, the timestamp of method's execution and the depth in the method call tree (to detect children method, *i.e.*, methods that are stared by an instrumented method). This information is useful for acknowledging where energy is being consumed (*e.g.*, the energy spend by a method excluding the energy spend by its children methods). Depth, which starts at 0, is incremented at each method entry, and decremented on method exit. Therefore, in an execution sequence, this value represent where the current monitored method is in the call tree.

Call tree

The code injected at the end of methods also collects the timestamp of method's end, and takes into account the restoration of the call tree up one level (when a method ends, the *hand* is given back to its parent method, or to *main* method). This is because Java uses *stack frames* in its Java stack. Stack frames *contain the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's Java stack. When the method completes, the virtual machine pops and discards the frame*

for that method. [20]. On method exit, we calculate the execution time of the method excluding its children methods. The latter is calculated on the method exit of each child, *i.e.*, when a method exits, we check whether is has father (`depth - 1`), then add its own execution time to its father's `MethodInfo` object (this object contains statistics about the method). As such, the execution time of each method is calculated by subtracting the total execution time of the method (between entry and exit) and the execution time of all of its children. Therefore, our implementation monitors the energy consumption of a method excluding its called methods (*e.g.*, children methods).

Network information

Network information are gathered by using a delegator to route all method call of sockets methods to a custom implementation where we add counters to count the number of bytes send and received by each method. We use a delegator class to route calls from the class `SocketImpl`[7] 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.

Applying energy models

Periodically, on each monitor cycle, or at the end of the program's execution, the JALEN agent processes the gathered data on each method invocation. It applies our energy models and provides the energy consumed by each method on the terminal or saved in a file.

4.2.4 Statistical Sampling Version

The statistical sampling version of JALEN collects information about running methods from the JVM, and correlates them with our energy models. This version follows a sampling strategy that is outlined in Figure 5:

- We first follow a two cycle approach: an *application* monitoring cycle where power consumption of software is gathered from POWERAPI; and *code* monitoring cycles where statistical information is collected on each running method.
- During the *application* monitoring cycle (typically at each 500 ms or 1 second), we monitor the energy consumption of the entire application using the approach implemented in POWERAPI. This provides us with the energy consumption of the application, which we will use as the total energy consumption of all executed methods.

[7] `http://docs.oracle.com/javase/7/docs/api/java/net/SocketImpl.html`

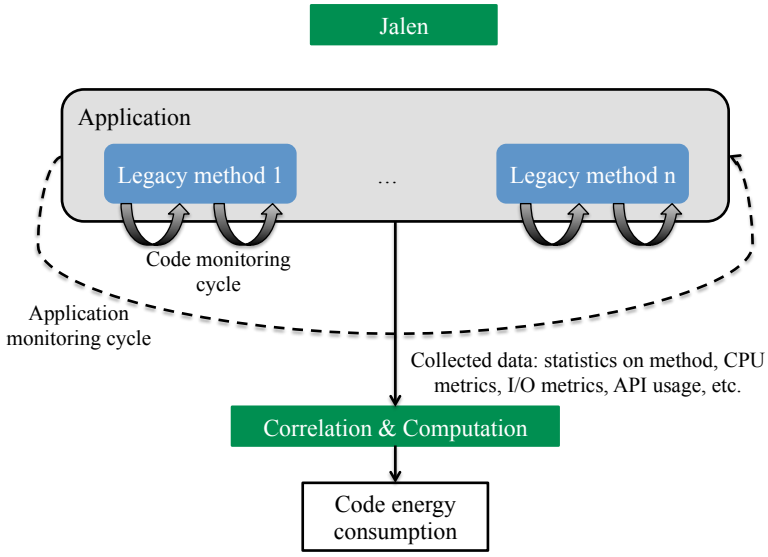

Fig. 5 The JALEN sampling implementation.

- The next step is to distribute the application energy consumption between the application's methods based on their execution. Concretely, we detect the method currently being executed by looking at the JVM's stack trace. We frequently detect and calculate how many times each method is on the top of the stack, typically each 10 ms (*e.g.*, the *code* monitoring cycle).
- We then correlate theses statistics with the CPU time of threads (gathered from the JVM), in order to estimate the energy consumption of methods.
- For example, two methods `Blue` and `Red` are executing for 10 seconds, and the *application* cycle is 1 second and the *code* cycle is 10 milliseconds. Each of these methods have different execution times and CPU utilization, therefore both methods are scheduled and executed accordingly (for example, method `Blue` waits for a network answer, thus the JVM executes `Red` during the wait). The method `Blue` is detected at the top of the stack 10 times during the *code* cycles while `Red` is detected 12 times. Therefore, the method `Blue` is awarded 45% of the energy, while the method `Red` is awarded 55% of the energy consumed during the *application* monitoring cycle (see Figure 6).
- For network energy, we detect and count the calls to Java's JDK methods responsible for network (such as java.net methods), and apply a similar energy distribution as described in the previous steps excluding all methods that do not call Java's JDK network methods.

In the next section, we validate our E-SURGEON approach, and run experimentations using our tools.

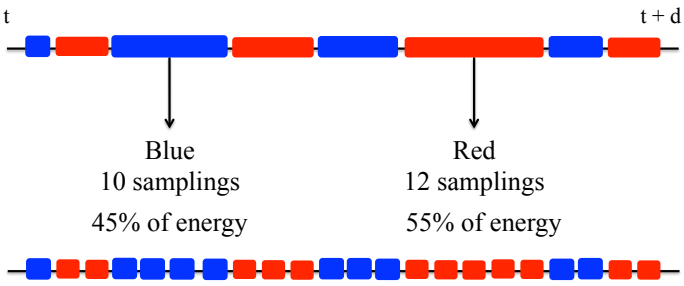

Fig. 6 The JALEN sampling approach.

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 (see Section 5.1), and then evaluate our JALEN Java agent (see 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[8], a Bluetooth powermeter. We compare the values of our library and the powermeter in a stress test on JETTY Web Server using APACHE JMETER[9] (see Figure 8), and using the Linux STRESS command[10] (see Figure 7). 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.

In particular, the powermeter monitors the energy consumption of the entire computer, including all of its hardware components, all applications and the operating system. In order to compare its values with our approach, we subtract the energy consumption during idle time from the energy monitored when running our experimentations. Concretely, on idle time, we measure the energy consumed by the computer using the powermeter (*e.g.*, $E\_idleMeter$). Then

[8] `http://www.alciom.com/en/products/powerspy2.html`
[9] `http://jmeter.apache.org`
[10] `http://linux.die.net/man/1/stress`

we run our experiments and measure again the energy of the computer (*e.g.*, $E\_totalMeter$). The difference between these two values is the energy consumed by the application (*e.g.*, $E\_appMeter = E\_totalMeter - E\_idleMeter$). However, this value corresponds to the energy for all hardware components stressed by the application (*e.g.*, CPU, memory, etc.). As we are only monitoring select hardware components, such as only the CPU or the network card, we decide to normalize the powermeter values based on the offset it has in comparison with our library (*e.g.*, $E\_offset = E\_totalMeter - E\_appLibrary$). Therefore, we verify that the values provided by our library are lower than the $E\_appMeter$, and we argue that they represent a better estimation to be used as a normalizing offset with the values of the powermeter. Ideally, a comprehensive model that estimates the energy consumption of all hardware components involved with an application execution should provide a value equal to $E\_appMeter$. However, as we do not have such model (our models are for the CPU and network so far), we tried to limit the impact of other components. We are planning, as a future work, to improve our normalizing approach by adding additional models for other hardware components. Finally, we calculate the average of the difference between the values of the powermeter and those of our library (*e.g.*, $AVG(E\_offset)$). Then we subtract for each measured value of the powermeter this calculated average ($E\_appMeterNormalized = E\_totalMeter - AVG(E\_offset)$).

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 (with variations of up to 8% or 2 watts on average for individual measurements when the Bluetooth synchronization lag is corrected). The error grows to nearly 3% in the Jetty stress test (with variations of 13% or 3-4 watts on average for individual measurements). 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.

5.1.2 Impact of frequencies variation and multi-core

In order to understand the impact of changing frequencies and multi-cores, we run experimentations on two popular video players, MPlayer and VLC. MPlayer's results (see Figure 9) are particularly interesting as they outline the effects of DVFS. Our energy model takes into account this variability of frequencies and voltages in the CPU, thus the variability of energy consumption in complex software. Also, the Linux stress experiment in Figure 7 shows the importance of multi-core in energy consumption. For a similar period of time, energy consumption varies greatly when running one, two, three or all the cores of the CPU.

On another example, we compared the energy consumption of VLC player decoding a video, and an execution of the Tower of Hanoi program, both

running on a Dell OptiPlex 745 workstation computer with an Intel Core 2 Duo processor (E6600). The results in Figure 10 show the impact of running applications under multiple frequencies (*i.e.*, 1.6 GHz and 2.4 GHz). Although both executions of VLC the two frequencies run the same program and decode the same video, the difference in energy consumption shows clearly the energy impact of DVFS as an approach to lower energy consumption (*i.e.*, 3325 and 1716 joules, respectively).

In contrast, executing a CPU intensive application, such as the Towers of Hanoi Java program, while forcing a specific frequency of the CPU also outlines the impact of DVFS on energy consumption. The results show the

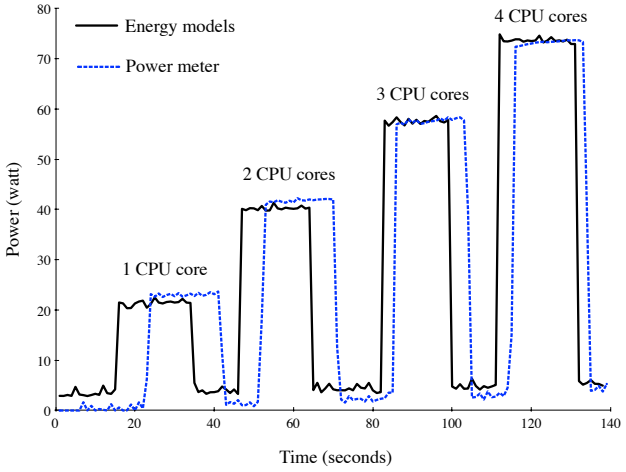

Fig. 7 Stressing the processor cores with the STRESS command.

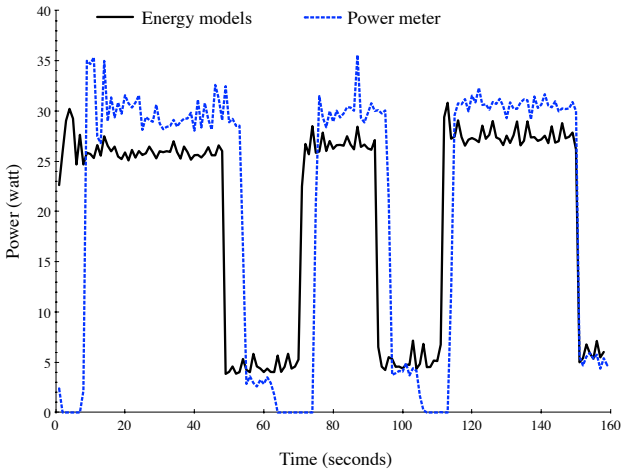

Fig. 8 Running stress tests on JETTY using JMETER.

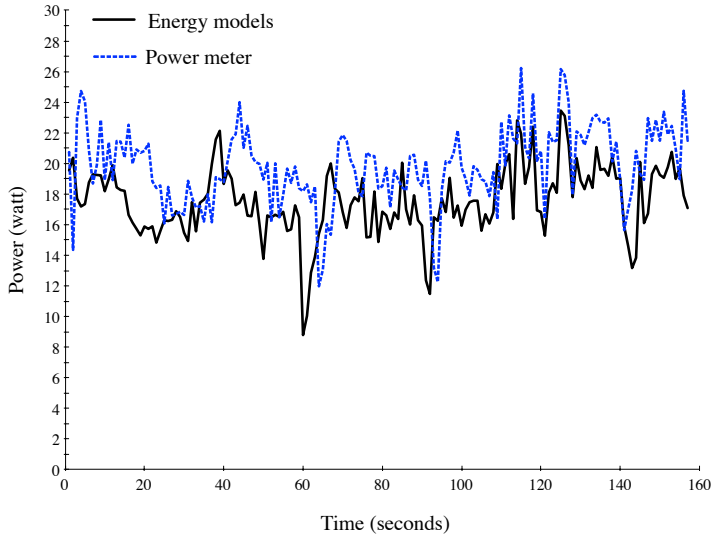

Fig. 9 Running stress tests on MPlayer.

difference in CPU power consumption and execution time while running a special version of the Tower of Hanoi algorithm (that writes to a file each step of the algorithm) on two different frequencies. On the faster 2.4 GHz frequency, the program consumes more energy per second and executes faster (totalling 3948 joules), while on the lower 1.6 GHz, it executes longer but with lower power consumption (totalling 5092 joules). In this example, running faster even at a higher frequencies results in better energy consumption than running at a lower speed but nearly 40% longer.

These results show the validity of our approach and that a simple time profiler is not enough to get energy insights because its does not take into account DVFS and multi-core CPUs.

5.1.3 Network Power

We run a network stress test using Iperf[11] and measure 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

[11] `http://www.manpagez.com/man/1/iperf`

on our test server. This observation is in correlation with the literature [17]. Therefore, we mostly outline the results of our CPU experimentation.

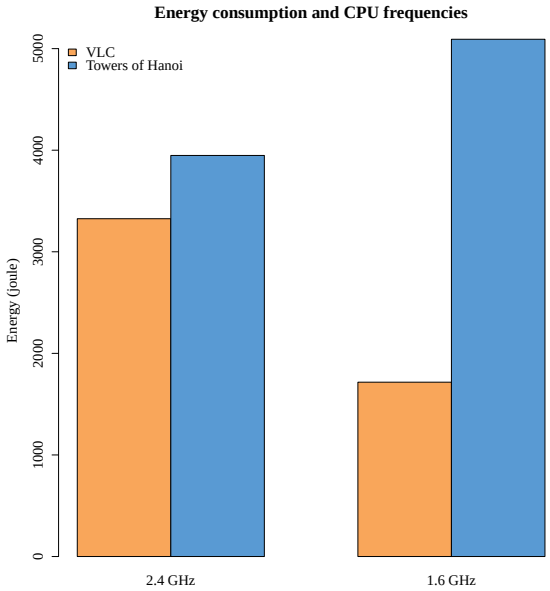

Fig. 10 An example of the impact of CPU frequencies on energy consumption.

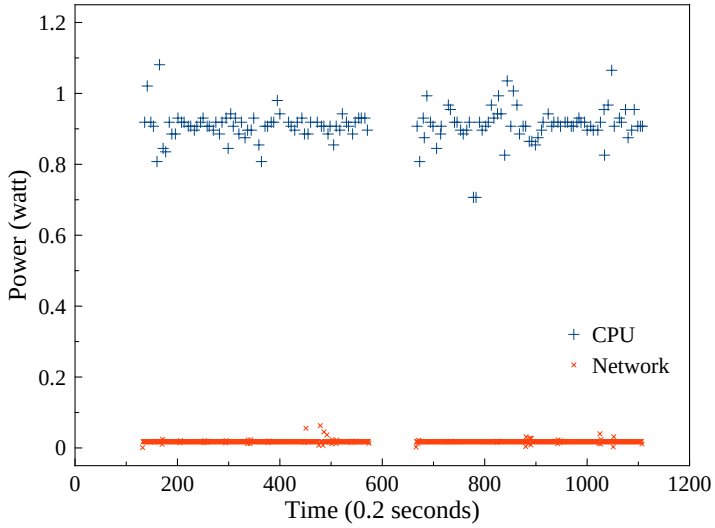

Fig. 11 CPU and network power consumption in IPERF stress test.

5.2 Jalen Validation

Both versions of Jalen, byte code instrumentation and statistical sampling, uses PowerAPI as an application level library. We run two sets of experiments: first, we measure the overhead and compare it to the overhead of software profilers (also due to the absence of similar code level energy profilers); second, we assess the accuracy by comparing the energy evolution with the CPU time evolution of CPU intensive applications running at 100% CPU, and with comparisons with another software profiler. The latter is relevant because we identified in [12] that for software that run at 100% utilization of the CPU all the time, there is a linear relation between their energy consumption and their execution time.

We run our experiments on a Dell OptiPlex 745 workstation with an Intel Core 2 Duo 6600 processor at 2.40 GHz and running Lubuntu Linux 13.04 64 bits, version 1.6-SNAPSHOT of PowerAPI, and Java 7. Energy data are calculated each 500 milliseconds. Sampling interval is at 10 ms.

5.2.1 Accuracy

As no other software profiler provides energy consumption of software code, we validate the accuracy of our approach by comparing energy consumption provided by our agent with CPU time. Therefore, we use the same CPU-intensive application, the recursive Java version of the Towers of Hanoi program, in order to demonstrate that the results provided by Jalen are accurate.

We compare the energy information provided by the instrumentation version of Jalen with the estimated CPU time of methods. Method `TowerOfHanoi.moveDisk` consumes 83.34% of the CPU and of its energy, while `TowerOfHanoi.solveHanoi` consumes 16.58%. Finally, the main method consumes 0.06% of the energy. Results in Figure 12 show similar match between CPU time and energy, which is what we anticipated as time and energy are linear in this context.

The sampling version, on the other hand, does not use CPU time in order to estimate the energy consumption of software code. Therefore, we decide to compare it to HPROF profiler [12], a software CPU profiling tool that also uses statistical sampling in estimating CPU usage of software code. The Java 2 Platform Standard Edition (J2SE) provides HPROF by default, as a command line tool. This tool estimates the CPU utilization percentage of all methods executing in the JVM. In contrast, Jalen can estimate the energy consumption of all methods, but also filter this estimation to a selection of methods (for example, limiting the estimation to the Tower of Hanoi's methods while excluding calls to the Java JDK's methods).

We compare the energy consumption provided by Jalen with the output information provided by HPROF. The results reported by HPROF show that `java.io.FileOutputStream.writeBytes` method uses 97.33% of the CPU

[12] `http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html`

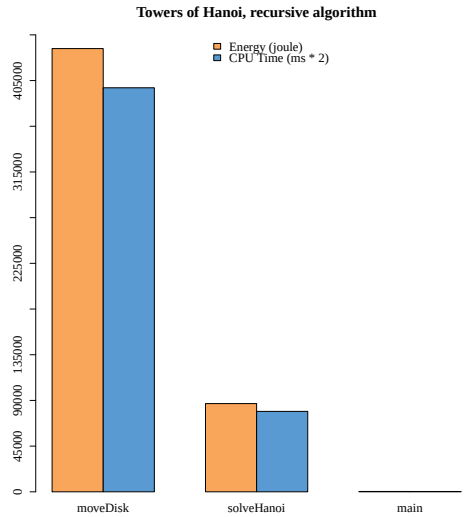

Fig. 12 Comparison between energy consumption and CPU time of the recursive version of the Towers of Hanoi program.

during the execution of the program. Sampling version of JALEN provides an energy consumption of this method at 96.05%, thus a variation of 1.3% between JALEN and HPROF.

However, JALEN can also filter methods, therefore, when excluding JDK's methods, the results show that `TowersOfHanoi.moveDisk` method consumes 99.92% of the energy. This is because `TowersOfHanoi.moveDisk` calls `java.io.FileOutputStream.writeBytes` method (and other methods, such as `java.io.BufferedWriter.write` or `java.io.Writer.write`) in order to write the program's results to a file. In addition, `TowersOfHanoi.moveDisk` itself have a net energy consumption of 0.73% (HPROF reports 0.13% for this method alone).

5.2.2 Overhead

The overhead of any energy profiler, or any software profiler, for that matter, is crucial to its usability. In order to acknowledge the overhead of our agent during execution, we calculate the time per individual request in Tomcat 7.0.42 using ApacheBench 2.3. On 10,000 requests, the base mean time per request is at 4.157 ms in average. However, when using the sampling version, the mean time per request is at 4.289 ms in average. HPROF also have a similar overhead at around 4.336 ms in average. The instrumentation version has a time per request of 9.532 ms in average. The overhead of the sampling version is therefore at 3.17%, while the instrumentation version of JALEN has an overhead of 129.29% in comparison to the base Tomcat (see Figure 13).

Although the overhead percentage of the instrumentation version is high, it is similar to other software profilers that use also byte code instrumentation.

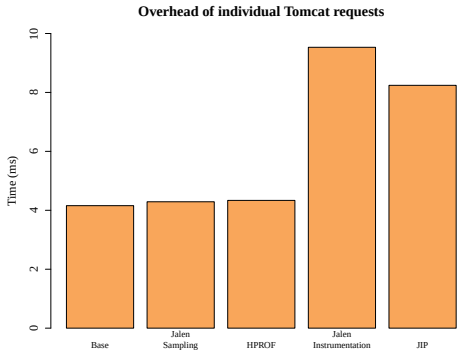

Fig. 13 Overhead of individual Tomcat requests using ApacheBench.

The Java Interactive Profiler or JIP [13] is a software profiler that uses similar byte code instrumentation of methods, however it does not produce energy related information. JIP 1.2 has a time per request of 8.241 ms in average in our experiment, thus an overhead of 98.24%. These metrics show the cost of instrumentation, and that our instrumentation version has an overhead similar to other software profilers that use byte code instrumentation.

5.2.3 Impact of Byte Code Instrumentation

These numbers validate the accuracy of our statistical sampling, but they additionally validate the impact of byte code instrumentation. We note that special care is required when analyzing information provided by the instrumentation version of JALEN. Instrumentation adds a fixed and constant overhead to all instrumented methods, therefore it is more visible (in total percentage) on small methods, or frequently executed methods. Values should be normalized by removing this constant overhead by a factor of the number of times the method is called.

To outline this situation, we compare the energy consumption of Google Guava's `Joiner.join` method using the instrumentation version of JALEN (see Figure 15), and the statistical sampling one (see Figure 14). The figure reports the energy distribution of the methods called by `Joiner.join` method on multiple executions. We increment the method's string parameter value and measure the energy consumption thereafter. The figure therefore shows the energy consumption of the same method execution but with a varying value of its string parameter. Both instrumentation and statistical sampling versions show similar energy trendline and linear evolution of the energy consumption. However, in the instrumentation version, small methods have high energy consumption, in particular, `Preconditions.checkNotNull` method. This latter is called four times in each join call (once in `Joiner.iterable`, once in `Joiner.appendTo`, and twice in `Joiner.toString`), and does nothing

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

other than comparing if a parameter is equal to null. However, due to instrumentation, `checkNotNull` has an energy consumption ranging from 3% to 10% of the total energy when varying the size of the strings to join.

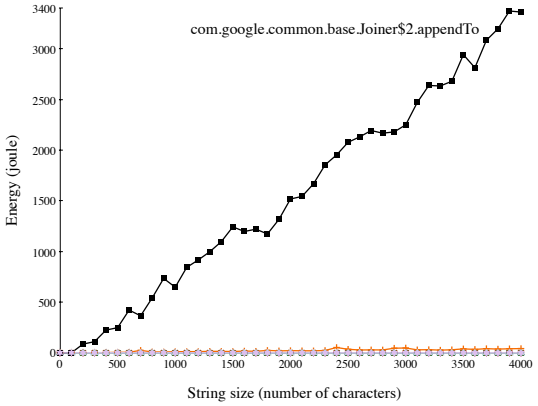

Fig. 14 Energy consumption of methods called by Google Guava's join method when varying its string parameter size, using statistical sampling version.

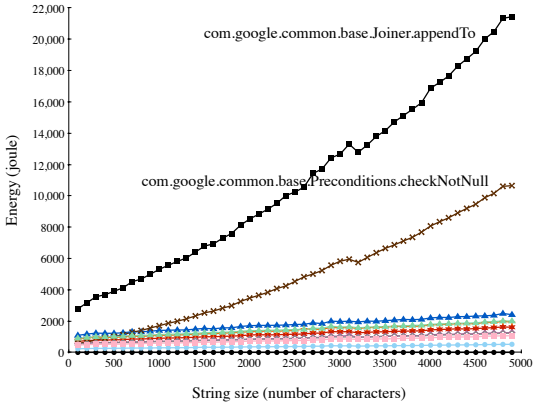

Fig. 15 Energy consumption of methods called by Google Guava's join method when varying its string parameter size, using instrumentation version.

5.2.4 Impact of Different Machines

Hardware components consume electrical energy. Our approach associates the energy consumption of hardware to the software code that initiated the task for hardware components. Therefore, energy consumption is highly dependent

on hardware components. To illustrate the impact of changing machines to energy consumption of software, we run the Xalan benchmark in the Dacapo benchmark suite [4] on two host configurations: a Dell OptiPlex 745 workstation with an Intel Core 2 Duo 6600 processor at 2.40 GHz and running Lubuntu Linux 13.04 64 bits; and a MacBook Pro 5,3 with an Intel Core 2 Duo T9900 processor at 3.06 GHz and running Mac OS X 10.7.5. We use version 9.12 of Dacapo, version 1.6-SNAPSHOT of PowerAPI, and Java 7 on both configurations, with a sampling interval at 10 ms and energy data are calculated each 500 ms.

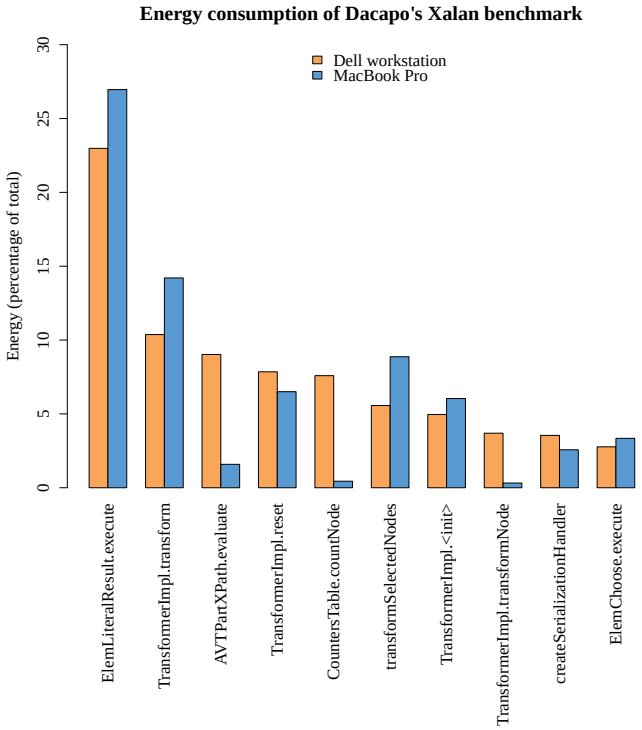

Fig. 16 Percentage of CPU energy consumption of the top 10 most energy consuming methods of Xalan Dacapo benchmark, on a Dell workstation and on a MacBook Pro.

The results in Figure 16 of the first 10 methods show a similar energy consumption trend. Both experiments outline `org.apache.xalan.templates.ElemLiteralResult.execute` and `org.apache.xalan.transformer.TransformerImpl.transform` as the most consuming methods. The results show also similar energy percentage values for these two methods, at 22.98% and 10.37% on the Dell workstation, and 26.95% and 14.02% on the MacBook Pro, respectively. On the other hand, raw energy values in joules are different. While at the Dell workstation, `templates.ElemLiteralResult.execute` consumes 55.9 joules, it consumes

31.5 joules on the MacBook Pro (25.24 joules and 16.6 joules for `transformer.TransformerImpl.transform`, respectively).

These results outline the importance of using percentages when comparing energy consuming of software code. This is mainly due to the different hardware that machines use, thus consuming different amount of energy while still keeping similar energy trends and distribution in software.

Due to the high overhead of the instrumentation version of JALEN and the noise introduced by byte code instrumentation, we decide to use the statistical sampling version of JALEN in the remainder of this chapter. 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 where the energy is being spend in software, or energy hotspots. This detection allows developers and other users to understand where and how the energy is consumed, and also to detect abnormal functioning in applications (*e.g.*, energy bugs).

We illustrate our approach with an example of a complex application: Jetty web server. We use version 9.0.4.v20130625 of Jetty distribution. As with our previous experiences, we run our experiments on a Dell OptiPlex 745 workstation with an Intel Core 2 Duo 6600 processor at 2.40 GHz and running Lubuntu Linux 13.04 64 bits, version 1.6-SNAPSHOT of PowerAPI, and Java 7. Energy data are calculated each 500 milliseconds, and sampling interval is at 10 ms.

Jetty web server is a lightweight application server and javax.servlet container. It is an example of real world complex application, counting 105,156 source lines of code (SLOC) of Java in the version we use for our study. We stress Jetty's asynchronous REST web application example (async-rest) using ApacheBench. The latter uses 25 concurrent users with 100,000 requests. We run the experiment 5 times, for around 205 seconds in total execution time (the first run at 54 seconds, then the others run at 37 seconds in average due mainly to the Java JVM's JIT functionality).

Results are presented in Figure 17. The graph portrays the top 10 most consuming methods in term of CPU energy consumption in the X-axis. The Y-axis represents the energy consumed during the execution of the experiment in percentage of the total energy consumed at all measured Jetty methods. The second set of bars in the Y-axis represents the number of invocations of the methods. We normalize this number by dividing it by 1000 in order to have a better overview in the graph. This invocation is gathered from the statistics JALEN's sampling version also provides.

The first observation is that the 10 most energy consuming methods of Jetty in this experiment consume the vast majority of the energy, 92.18%. Specifically, two methods consume nearly 60% of the energy:
`util.BlockingArrayQueue.poll` (29.92%) and

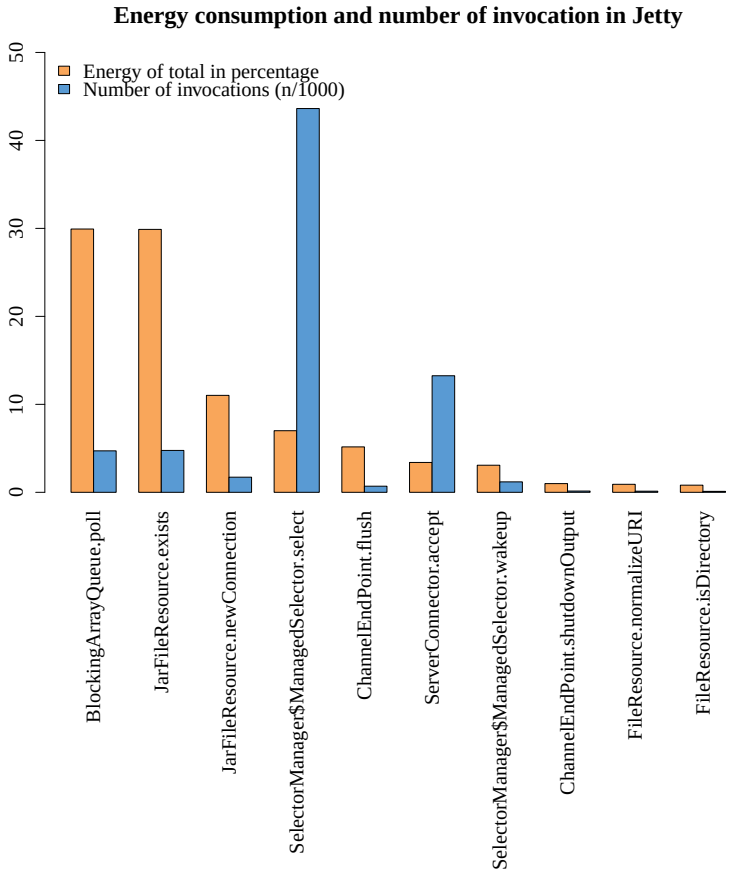

Fig. 17 Energy consumption of the 10 most energy consuming methods of Jetty in our experiment.

`util.resource.JarFileResource.exists` (29.88%).
Five other methods (`util.resource.JarFileResource.newConnection`, `io.SelectorManager$ManagedSelector.select`, `io.ChannelEndPoint.flush`, `server.ServerConnector.accept`, and `io.SelectorManager$ManagedSelector.wakeup`) consume between 3% and 11%, while the energy consumption of the remaining methods is negligible (less then 1%).

In contrast, the same methods are also the most invoked. Nevertheless, two methods have a high invocation number with lower energy consumption. This is the case for `io.SelectorManager$ManagedSelector.select` and `server.ServerConnector.accept methods`. The former is the most invoked, 43,624 times and consumes 7% of the total energy (or 487.34 joules on our configuration). The latter is invoked 13,250 times and consumes 3.38% of the total energy (or 236.28 joules).

In order to understand better the energy hotspots in software, we introduce energy per invocation (epi) unit, which is the energy consumed by one invocation of a method, and is calculated by dividing the energy consumption by the number of invocation. The two most invoked methods have therefore a low epi in comparison with less invoked methods (and sometimes less energy consuming methods). Figure 18 outlines the epi of the 10 most energy consuming methods in our experiment. We observe that in average, the epi of most methods is between 0.4 and 0.5 joule, with the notable exception of the two most invoked methods.

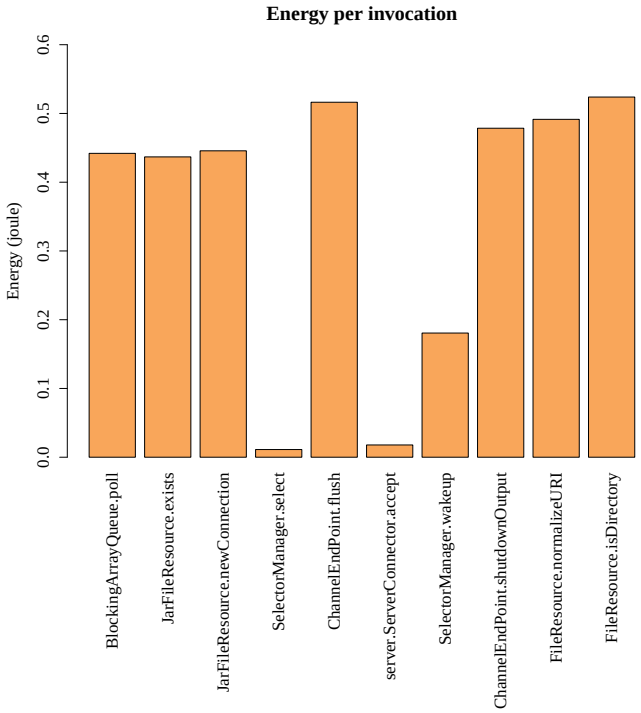

Fig. 18 Energy per invocation (epi) of the 10 most energy consuming methods of Jetty in our experiment.

In addition to detecting hotspots at the methods level, our approach can detect most energy consuming classes. Figure 19 outlines the 6 most consuming classes of Jetty during our experimentation. These 6 classes consume together 96.02% of the total energy consumed by Jetty classes. The remaining 129 classes consume the rest, 3.97%. We observe that two classes consumes more than 70% of the energy: `util.resource.JarFileResource` (40.93%, 2 methods invoked) and `util.BlockingArrayQueue` (30.07%, 4 methods invoked). These two classes are averaged sized classes with the former counting 291 SLOC and the latter 691 SLOC.

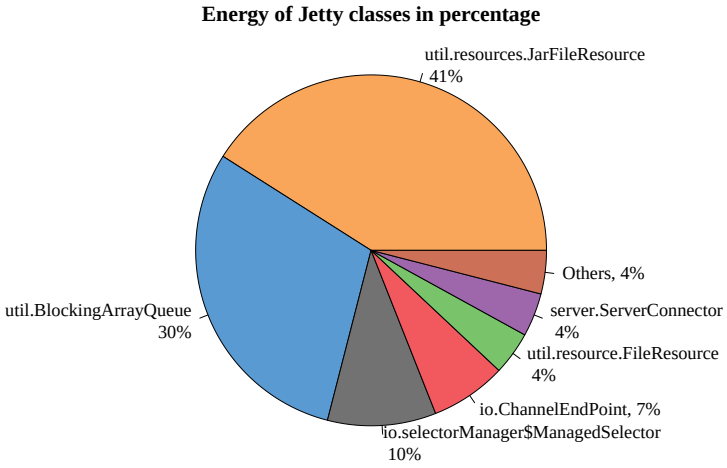

Fig. 19 Energy consumption in percentage of the 6 most energy consuming classes of Jetty in our experiment.

Our benchmark stress scenario can explain these results. The former stresses Jetty's asynchronous rest web application example. This web application *uses Jetty asynchronous HTTP client and the asynchronous servlets 3.0 API, to call an eBay restful web service* as explained in [14]. When the initial request passes to the servlet, it is detected as the first dispatch, thus the request is suspended and a queue list (to accumulate results of requests) is added as a request attribute. This explains the energy consumption of `util.BlockingArrayQueue` class and its methods. After the suspension, *the servlet creates and sends an asynchronous HTTP exchange for each request*, and when all responses are received, the results are retrieved and a response is generated. The calls for `util.resource.JarFileResource` class, and its `exists` method which checks whether a represented resource jar exists, and its `newConnection` method that is used for connecting to JAR resources, are explained by the need to access Jetty's own jar files and the web application's jar files. The experiment is run multiple times, and the asynchronous example is a relatively a small example, therefore this jar access is notable in term of total energy consumption percentage.

We believe that this information can help the developers to investigate alternative implementations of the most consuming classes and methods in order to reduce their power footprint. 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.

[14] http://webtide.intalio.com/2013/04/async-rest-jetty-9/

7 Discussions

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.

7.1 Prefer Percentages Over Energy Raw Values

First, our aim was to observe trends in energy consumption (which is the reason of using percentages when comparing methods and classes). Energy consumption of software code is measured in joules, and is the energy consumed by hardware due to tasks initiated by software. As such, changing hardware will also change the raw energy values even for the same software code. The same experiment run on different hardware produces different energy values due to the physical nature of hardware components (for example, running the same program on a laptop or on a server).

However, percentages values do not change with hardware, as software code is running the same tasks and using hardware resources accordingly. Our experiment in Figure 16 illustrates the limited impact of changing hardware on the percentage distribution of the energy consumption in software code. The goal of our approach is to observe trends in energy consumption and profile applications to detect energy hotspots. Therefore, we argue that using percentages when comparing energy consumption of methods and classes is more useful and representative than raw values. Our approach is thus 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.

7.2 Overhead and Instrumentation Noise Are Not to be Ignored

Second, one major advantage of using the statistical sampling version of JALEN is its negligible overhead cost. Our results show a low overhead for sampling (at around 3%), in comparison to the high overhead of instrumentation (at around 130%). This overhead cripples any *real world* uses of instrumentation for energy measurements. However, we found that sampling provides also accurate enough values for detecting energy trends and energy hotspots. Therefore, using statistical sampling offers the best tradeoff between accuracy of results and low overhead.

Besides execution overhead, byte code instrumentation *pollutes* the energy results of each method by the energy cost of the instrumentation code itself. Without normalizing the results and removing this additional cost (that is dependent on the energy cost of the instrumentation code, and on the number of times it is executed), energy results are not correctly reported as we report

in our experimentation. The energy consumption of small methods is reported as higher than it is, and frequently executed methods have a high overhead due to the additional cost of multiple execution of the instrumentation code.

7.3 The Need for Energy Variation Modeling

Our experiments report on the energy distribution and hotspots of software code in a specified context. We are able to detect and identify the most energy consuming methods or classes. However, this identification is for a fixed set of parameters and configuration. Our approach is similar to *screenshots*, where we take an energy snapshot of software code. Although this is useful for energy debugging and optimization, it lacks execution variability. For example, we provide the energy variation model of methods based on the variation of their parameters. This provides a relational table between methods and their energy model, thus allowing developers to the best energy efficient method for their programs. Developers will have empirical energy models that will allow them to choose the most power efficient software library and configurations for their usage in their software. In the next section, we illustrate and detail our approach for this research direction.

8 Inferring Energy Consumption of Software Libraries

Measuring energy consumption of software provides a snapshot of the energy profile of such software under one particular execution. The energy reported by our E-Surgeon approach is static, *e.g.*, values are related to a specific execution of software in a specific configuration. Changing a parameter in a method or modifying input parameters will therefore require a new execution of the application in order to get its energy consumption. We propose to infer the energy variation model of software based on their input parameters. We first present a motivation scenario using the RSA asymmetric algorithm [16], then we detail our approach for inferring energy variation based on input parameters, and present our framework called Jalen Unit.

8.1 RSA Asymmetric Algorithm

We take an RSA asymmetric encryption/decryption algorithm [16] and measure its energy consumption while varying the length of the RSA public and private keys. The algorithm generates an RSA key, then encrypt and decrypt 10 times a random BigInteger with a bit length of 10,000. We use our PowerAPI library to measure the energy consumption of the RSA algorithm.

The experimentation is done on a Dell OptiPlex 745 with an Intel Core 2 Duo 6600 processor at 2.40 GHz and running Lubuntu Linux 13.04, version 1.6-SNAPSHOT of PowerAPI, the statistical sampling version of Jalen and

Java 7. We collect energy data each 500 milliseconds for PowerAPI, and the sampling rate for JALEN is at 10 milliseconds.

The results, in Figure 20, show an exponential rise in the energy consumption of the RSA algorithm when increasing the RSA key length.

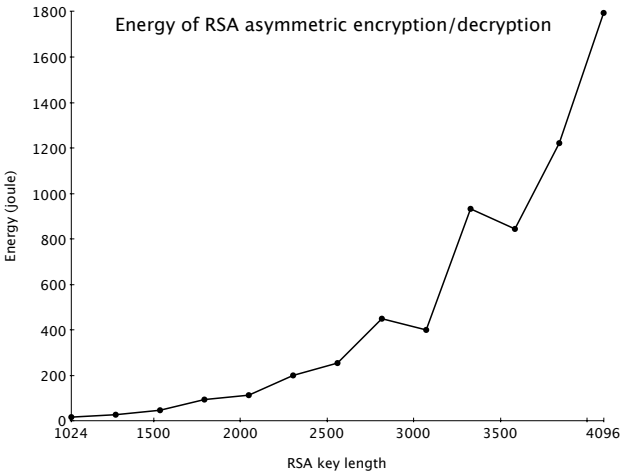

Fig. 20 Evolution of the energy consumption of RSA asymmetric encryption/decryption according to key length.

Even though these numbers show the evolution of the energy consumption of the RSA algorithm, we want to understand which portion of the code is responsible for the exponential increase. Therefore, we use JALEN to measure the energy consumption of the classes and methods of the RSA algorithm. Results, in Figure 21, show that two methods are responsible for the majority of the energy consumption: `java.math.BigInteger.oddModPow`, and `java.math.BigInteger.montReduce`. From these methods, `oddModPow` have a clear exponential increase, while `montReduce` follows a logarithmic growth.

RSA encryption/decryption algorithm is an exponential one as described in [16]. Therefore, our experiment results provide additional validation to our measurement approach. In particular, the method responsible for the exponential growth in energy consumption in our implementation of RSA algorithm is the method that does the exponential calculation, `oddModPow`.

These libraries are used by other software and therefore, improvement in their energy efficiency would benefit to a large pool of applications. Thus, what is the power consumption model of software libraries? How much a library call consumes? What is the impact of varying invocation parameters in a method of an API? Which library providing the same functionalities is more energy efficient?

These questions motivate us to propose a new approach called JALEN UNIT [14]. It is our energy framework that generates energy models for software code based on empirical benchmarks.

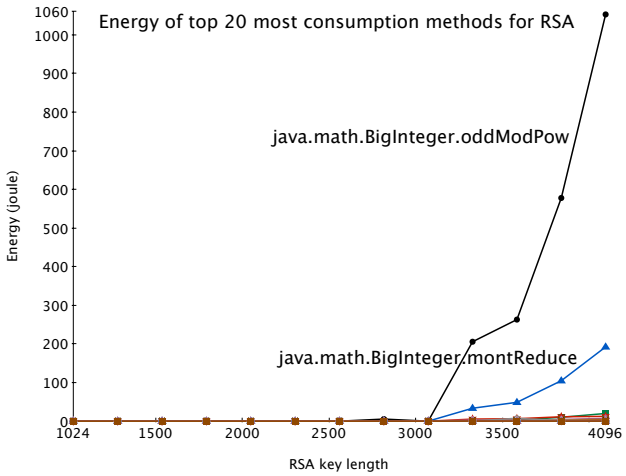

Fig. 21 Evolution of the energy consumption of RSA asymmetric encryption/decryption according to key length.

8.2 Jalen Unit

Jalen Unit provides benchmarks for modeling the energy consumption of software methods through automatic benchmarking. For instance, it generates individual benchmarks for each method in a software library, and for each of its parameters. These benchmarks stress the method based on a set of input values for its parameters. These values are determined through different injectors, and multi-parameters methods are managed through different strategies. Next, all generated benchmarks are executed. For each, we measure its energy consumption, then the results are aggregated and analyzed to produce the method's energy profile and variation model.

Concretely, Jalen Unit cycles through every package, class, and method in a Java library. For each method and each of its parameters, an energy benchmark is created following a variation strategy for the benchmarked parameter. The benchmark is then executed and Jalen is used to measure its energy consumption. Finally, energy data for the benchmark and the variation of parameters is reported in an output file that is later plotted as a graph. The variation strategies of each parameter are done through injectors implemented for Java primitive and object types. The framework can, therefore, be extended with application-specific injectors describing alternative variation strategies. Java objects can be benchmarked automatically if their injector model is implemented in the framework.

The initial implementation of Jalen Unit provides injectors for primitive types: `Integer`, `Double`, `Long`, `Float`, `Boolean`, and `Character`, in addition to the `String` class. We prefer to implement our own injector instead of using existing injectors, such as YETI [3] which performs random testing, because we want to provide different strategies for benchmarking and testing methods.

This provides a good strategy for detecting abnormal behavior in software code, such as exceptions or huge CPU load for certain values. However, it does not offer a comprehensive strategy for evaluating the energy variation of methods by input parameters. For example, we develop an injector for integers where the integer values evolve with an increment, from a start value to a final value (*e.g.*, integer values from 10 to 100 with a hop of 10 leads to 10 benchmarks with values of 10, 20, ... to 100). Another injector for integers evolves the integer randomly using the `Math.random` method in Java. Although integers are all of the same size, changing their value impacts the execution of methods, therefore their energy consumption. For example, an integer parameter that is used as a final value to a for-loop may have a high impact because increasing its value implies that tasks are being executed for longer period of time and consuming more energy.

Injectors for other types also implement different variation strategies, such as varying the length of a string parameter randomly, or from a start value to a final value, or choosing the characters of the string from a subset of the alphabet. The variation strategies are endless, and offer the advantage of better flexibility and extendibility of the framework. This flexibility is also useful for domain specific applications, where random testing is not representative of the *real world* workload. By providing an extensible framework and providing freedom of choice in method variation model strategies, we propose a solution that can be customized for specific needs. Therefore, better representative energy variation models can be empirically achieved.

Concretely, an injector is a Java class implementing the `Iterator` and `JalenModel` interfaces. The latter adds additional methods to the iterator `next` and `hasNext` methods, such as a `getDefaultValue` method that returns an object of a default value of the injector. The following listing provides an excerpt of code of the default integer injector (syntax modified and shortened for space concerns):

```
public IntegerModel(int start, int end, int inc);
public boolean hasNext()
        return this.current <= this.end;
public Object next() {
        int result = this.current;
        this.current += this.inc;
        return result;
}
public Object getDefaultValue()
        return this.start;
```

Multi-parameters methods are managed by varying one parameter at a time, while the others use a default value. Others strategies are possible, such as varying multiple parameters while fixing the values of some, or modifying all parameters randomly. We are aware that more comprehensive strategies are required for a refined energy variation model, therefore our multi-parameters strategy is just an initial implementation for handling the complexity of multiple parameters.

Benchmarks are then run and their energy consumption is measured using JALEN. Finally, the generated energy results are aggregated and the energy variation model of methods is inferred.

8.3 Limitations

Our framework allows the generation of an energy variation model for methods based on the values of their parameters. Although our results are promising [14], some limitations are to be noted and offer additional challenges for future directions.

First, our approach benchmarks methods individually, therefore the interactions between methods and parameters are not fully studied using our framework. The impact of methods interactions is complex and interesting in order to have a better understanding of energy consumption in software, and will be addressed as a future work.

Second, our framework offers manual analysis of energy variation results. In particular, we would like to add automatic analysis of the measured empirical data. In an ideal situation, a mathematical formula, or set of formulas, would be generated automatically in order to relate the energy consumption of software methods with the values of their input parameters.

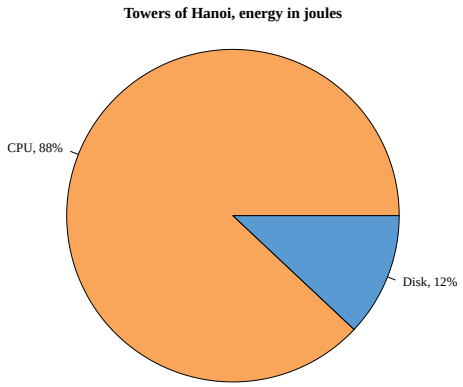

Fig. 22 CPU and disk energy consumption of the Towers of Hanoi algorithm.

Finally, our framework infers energy variation models based on CPU energy. However, additional hardware components have a non-negligible energy consumption. In our experimentations, we detected that Ethernet network energy is negligible compared to the CPU (see Figure 11). On the other hand, our initial work in adding additional sensors shows that the energy consumption of the hard disk is not negligible. Figure 22 shows that energy consumed by the Towers of Hanoi program is divided as 88% for the CPU (doing the calculations, CPU cycles for preparing to write each step of the algorithm to a file), and 12% for the disk (actual writing of each step of the algorithm to a

file stored on the hard disk). The figure outlines that other components then the CPU have a non-negligible impact on energy consumption, even though we measured a CPU-intensive application. On the other hand, our experience in Figure 22 also uses non-negligible amounts of data in the memory. Our approach and model do not yet acknowledge the impact of energy consumption of memory access. But it is also crucial to note that the location where the application's data are stored and accessed may have an impact on energy consumption. Whether the computation requires access to the main memory in RAM or the CPU memory cache, the energy consumed by the system, but also by the CPU alone, may be different. The time to read a data stored on the hard disk, RAM memory, or the CPU memory cache is variable, and the more time the CPU waits the more energy it will be consuming. Therefore, it is important to study the effects of other hardware components, not just on energy consumption, but also on the energy variation model of methods in relation to the values of their input parameters.

9 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 our approach to measure hardware components that contribute to a high percentage in energy consumption of applications [17], 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.

In addition, our current approach lacks intelligent analysis of the results. In particular, it only provides energy information about software code while the developer has to interpret and analyze the results. Monitoring applications at runtime allow users and developers to acknowledge the energy cost of their applications. Software can then be diagnosed and its energy efficiency improved. However, what if developers had tools to empirically measure the energy consumption of their code, and get empirical data about the energy evolution trends of their code? These data can therefore be used to diagnose the code to detect energy bugs, understand the energy distribution of the application, or establish an energy profile or classification of software.

Another challenging future work is to provide an abstraction of the energy consumption in relation to hardware. The energy consumption of an application depends on multiple factors, including the hardware configuration itself,

but also the execution scenario. For example, if certain data is stored on the processor cache, then energy consumption may be lower as there will be less energy lost while waiting to read the data. In addition, using a different set of values for an application's input parameters would lead to a different energy consumption. In our work, we propose a first step in this direction by using percentages instead of raw values, and by inferring the energy consumption of software libraries through empirical benchmarking.

10 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.

10.1 Energy Metering and Modeling

Monitoring energy consumption of hardware components usually requires a hardware investment, like a multimeter or a specialized integrated circuit. For example in [11], 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 [19] and [9].

In [19], 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 [9], 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 spent running on a resource (*i.e.*, CPU, network card). In [7], 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.

10.2 System Level Tools

pTop [5] 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[15], in particular cpufreq-info to get kernel information about the CPU (*i.e.*, frequency), and cpufreq-set to modify CPU

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

settings such as the frequency. IOSTAT[16] that is used to get devices' and partitions' input/output (I/O) performance information, as well as CPU statistics. Other utilities [8] 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.

10.3 Application Profiling Tools

Several open-source or commercial Java profiling tools already propose some statistics of Java applications. Tools, such as VISUALVM[17], JAVA INTERACTIVE PROFILER (JIP)[18], or the OKTECH PROFILER[19], 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.

11 Conclusion and Future Work

In this paper, we report on the E-SURGEON runtime energy monitoring solution. It allows gathering and calculating 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, energy hotspots of Java-based applications. In particular, our approach detects methods and classes responsible for the most energy consumption. We define energy per invocation as a more representative energy measurement unit, and we argue to use percentage values over raw energy values. Finally, our work opens windows into energy evolution modeling, where we empirically model the evolution of energy consumption based on input parameters.

[16] `http://linux.die.net/man/1/iostat`
[17] `http://visualvm.java.net`
[18] `http://jiprof.sourceforge.net`
[19] `http://code.google.com/p/oktech-profiler`

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.

References

1. Asm. http://asm.ow2.org/
2. The Green Challenge for USI 2010. `http://sites.google.com/a/octo.com/green-challenge`
3. York Extensible Testing Infrastructure. https://code.google.com/p/yeti-test/
4. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, pp. 169–190. ACM Press, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1167473.1167488
5. Do, T., Rawshdeh, S., Shi, W.: 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 (2009)
6. Feeney, L., Nilsson, M.: 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, vol. 3, pp. 1548–1557 (2001). DOI 10.1109/INFCOM.2001.916651
7. Flinn, J., Satyanarayanan, M.: 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, p. 2. IEEE Computer Society, Washington, DC, USA (1999)
8. Gite, V.: How do I Find Out Linux CPU Utilization? `http://www.cyberciti.biz/tips/how-do-i-find-out-linux-cpu-utilization.html`
9. Kansal, A., Zhao, F.: Fine-grained energy profiling for power-aware application design. SIGMETRICS Perform. Eval. Rev. **36**(2), 26–31 (2008). DOI 10.1145/1453175.1453180
10. Kuleshov, E.: Using the ASM framework to implement common java bytecode transformation patterns. In: AOSD'07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development. Vancouver, Canada (2007)
11. McIntire, D., Stathopoulos, T., Kaiser, W.: 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, pp. 576–577. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1236360.1236448
12. Noureddine, A., Bourdon, A., Rouvoy, R., Seinturier, L.: A preliminary study of the impact of software engineering on greenit. In: 1st International Workshop on Green and Sustainable Software (GREENS'12), pp. 21–27 (2012). DOI 10.1109/GREENS.2012.6224251
13. Noureddine, A., Bourdon, A., Rouvoy, R., Seinturier, L.: Runtime monitoring of software energy hotspots. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pp. 160–169. ACM, New York, NY, USA (2012). DOI 10.1145/2351676.2351699. URL `http://doi.acm.org/10.1145/2351676.2351699`
14. Noureddine, A., Rouvoy, R., Seinturier, L.: Unit testing of energy consumption of software libraries. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC 2014. ACM, New York, NY, USA (2014)

15. Pouwelse, J., Langendoen, K., Sips, H.: Dynamic voltage scaling on a low-power microprocessor. In: MMSA'00: Proceesings of the 2nd International Symposium on Mobile Multimedia Systems and Applications, pp. 157–164. Delft, The Netherlands (2000). URL http://www.pds.ewi.tudelft.nl/~koen/papers/mmsa.ps.gz
16. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978). DOI 10.1145/359340.359342. URL http://doi.acm.org/10.1145/359340.359342
17. Rivoire, S., Shah, M.A., Ranganathan, P., Kozyrakis, C.: JouleSort: a balanced energy-efficiency benchmark. In: SIGMOD'07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp. 365–376. ACM, New York, NY, USA (2007). DOI 10.1145/1247480.1247522
18. Ruhl, C., Appleby, P., Fennema, J., Naumov, A., Schaffer, M.: Economic development and the demand for energy: A historical perspective on the next 20 years. Energy Policy **50**(0), 109 – 116 (2012). DOI 10.1016/j.enpol.2012.07.039. URL http://www.sciencedirect.com/science/article/pii/S0301421512006313
19. Seo, C., Malek, S., Medvidovic, N.: 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, pp. 421–424. ACM, New York, NY, USA (2007). DOI 10.1145/1321631.1321699
20. Venners, B.: Inside the Java Virtual Machine, 1st edn. McGraw-Hill Professional (1999)
21. Vereecken, W., Van Heddeghem, W., Colle, D., Pickavet, M., Demeester, P.: Overall ict footprint and green communication technologies. In: ISCCSP'10: Proceedings of the 4th International Symposium on Communications, Control and Signal Processing, pp. 1 –6 (2010). DOI 10.1109/ISCCSP.2010.5463327
22. Webb, M.: 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)