A Preliminary Study of the Impact of Software Engineering on GreenIT

Adel Noureddine, Aurélien Bourdon, Romain Rouvoy, Lionel Seinturier

A Preliminary Study of the Impact of Software Engineering on GreenIT

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

***Abstract*—GreenIT has emerged as a discipline concerned with the optimization of software solutions with regards to their energy consumption. In this domain, most of state-of-the-art solutions offer limited or constraining approaches to monitor the energy consumption of a device or a process. In this paper, we therefore report on a runtime energy monitoring framework we developed to easily report on the energy consumption of system processes. Concretely, our approach adopts an OS-level library, called POWERAPI, which estimates the power consumption of processes according to different dimensions (CPU, network, etc.). In order to better understand potential energy leaks of legacy software, we use this library to study the impact of programming languages and algorithmic choices on the energy consumption. This preliminary study is based on an empirical evaluation of a eight implementations of the Towers of Hanoi problem.**

***Keywords*-Performance; Measurement; Experimentation; Energy; Power Model; Monitoring**

I. INTRODUCTION

Energy-aware software solutions and approaches are becoming broadly available, as energy concerns are becoming mainstream. The increasing usage of computers and other electronic devices (*e.g.*, smartphones, sensors) is continuously impacting our overall energy consumption. *Information and Communications Technology* (ICT) accounted for 2% of global carbon emissions in 2007 [1] or 830 $MtCO_2e$ [2], and is expected to grow to 1,430 $MtCO_2e$ in 2020 [2]. These values illustrate the opportunities for efficient ICT solutions to reduce carbon emissions and energy consumption.

Rising energy costs in computers and mobile devices implies the optimization and the adaptation of computer systems. In this domain, research in GreenIT already proposes various approaches aiming at achieving energy savings in computers and software. In particular, monitoring the energy consumption of the system is a requirement to achieve such savings. However, most of the state-of-the-art approaches either focus on the hardware [3], or offer coarse-grained energy feedbacks of software [4], [5].

In this paper, we therefore propose to gather applications energy feedback information at runtime and with similar accuracy as hardware equipments while using only a software approach. Our approach consists of a system monitoring library (at the operating system level), called POWERAPI. POWERAPI estimates the energy consumption of running processes, in real-time, based on raw information collected from hardware devices (*e.g.*, CPU, network card) through the operating system. We use both state-of-the-art energy models and propose new models to compute the energy consumption of software.

Using this monitoring framework, we compare the energy footprint of eight implementations of the Towers of Hanoi program. Our preliminary results demonstrate that we can clearly observe the impact of different implementations of algorithms on the energy consumption, thus providing the opportunity at a later stage to reduce their energy footprint.

The remainder of this paper is organized as follows. In Section II, we describe our motivations and the main challenges we tackle. Section III describes our approach, the design of our proposed architecture and our energy models. Section IV details the implementation and validation of our prototype. In Section V, we report and discuss the preliminary results we obtained from comparing the Towers of Hanoi implementations. Related work is discussed in Section VI, while we conclude in Section VII.

II. MOTIVATION AND CHALLENGES

According to [6], ICT consumes up to 7% of global power consumption (or 168 GW) in 2008. This number is predicted to grow and double to 433 GW in 2020 or more than 14.5% of worldwide power consumption [6]. With the rise of ICT power consumption, power-aware optimizations and management cannot be done efficiently without accurate power measurements. In particular, we argue that monitoring the power consumption of applications (and not only hardware) has become a requirement for power-aware *Software as a Service* (SaaS) and power-aware software adaptation.

Hardware monitoring is usually achieved through additional hardware measurement equipments, such as multimeters or specialized integrated circuits (cf. Section VI). This approach offers a precise and accurate measurement of the energy consumption of hardware components, but at the cost of an additional investment. Furthermore, it cannot monitor the energy consumption of software components and running applications.

We rather believe that a scalable approach can be achieved by a software-centric approach. Monitoring the energy con-

sumption of software has to yield many challenges in order to build an accurate software-centric approach. In particular, the biggest problem that software monitoring tools face is providing accurate estimations of energy consumption based on collected raw information. Unlike hardware measurements, software approaches use energy models in order to provide an estimation of the energy consumption of software components. However, these estimations tend to have different degrees of accuracy and overhead.

In addition to accuracy, the overhead of the monitoring platform is to be optimized in order to limit its impact. The overhead depends both on the degree of accuracy needed and on the cost of the monitoring tool and the monitored applications. This leads to a difficult tradeoff between the accuracy requirements and the cost of the software monitoring tool.

Laying these challenges, we propose in the next section an approach named POWERAPI for monitoring applications and processes at runtime.

III. POWERAPI DESIGN AND APPROACH

In this section, we present the POWERAPI general architecture and we describe the approach we use for defining our energy models. The architecture is based on a modular approach, mixing power monitoring tools with energy models in order to provide energy information of software.

A. Architecture

POWERAPI architecture is a modular architecture built for agile software programming. The core of the architecture are *power modules*. POWERAPI is constructed as separate modules that can be started or stopped at runtime upon needs. A set of OS-dependent *sensor* modules (*e.g.*, CPU, network) collect raw information about hardware resource utilization, either directly from the devices or through the operating system. This information is then exposed to another set of OS-independent *formula* modules that use our power models (cf. Section III-B) to compute the power consumption of each hardware component. These modules also compute the power consumption of running processes and applications per hardware resource. A local database is also used to store configurations about the hardware resources and used to auto-calibrate POWERAPI depending on the environment. Finally, all these modules are managed by a *life cycle* module. The latter allows to start, stop, add, remove or modify modules depending on monitoring needs and commands sent by applications. Figure 1 depicts the overall architecture of POWERAPI.

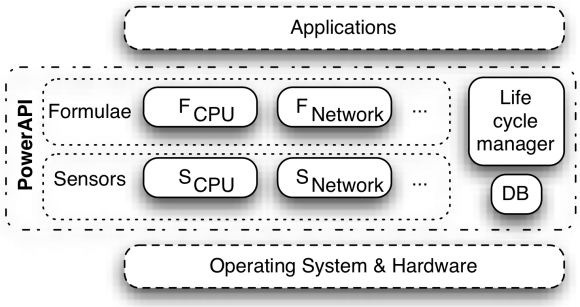

Figure 1. POWERAPI Reference Architecture.

B. Power Models

We propose a comprehensive energy model using our own proposed formulae and formulae taken from the state-of-the-art. In [7], the energy cost of software is computed based on the following formula:

$$E_{software} = E_{comp} + E_{com} + E_{infra}$$

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 is included in the computational cost of our power models and in our prototype. From this, we can abstract our global power formula as follows:

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

At this stage, we developed 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 will detail the CPU and network energy models we use in POWERAPI.

1) CPU Model: The CPU power consumed by a specific process (in our case we use process $PIDs$) can be represented as:

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

Where $P_{CPU}^{PID}(d)$ is the CPU power consumed by the specific *PID* during a given duration *d*, $P_{CPU}(d)$ is the global 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.

Global CPU power: The overall power consumption for the majority of modern processors (CMOS[1]) follows the standard equation:

$$P_{CPU}^{f,v} = c \times f \times V^2 \quad (2)$$

where f is the frequency, V the voltage and c a constant value depending on the hardware materials (such as the capacitance). 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 characteristics of the CPU and allows accurate power consumption monitoring.

According to formula 2, calculating the overall CPU power for a given time is equal to calculating 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 of the cases this value is not available. 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 by [8]:

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

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.

For the dynamic part, the frequency f is associated to a specific voltage V. One or more frequencies can be associated to a specific voltage. Lowering the voltage results in changing frequency. The other way around is also valid. Frequencies used by a processor are provided by the operating system APIs, while voltages are given by manufacturers.

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

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

[1]Complementary Metal Oxide Semiconductor

Our approach is inspired by well-known tools, such as the top linux program[2].

Thus, P_{comp} of formula 1 and for the CPU power consumption in a duration d is equal to :

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

2) Network Model: The network power of a process is calculated using a formula similar to the CPU power formula. We base our model on available information whether they are collected at runtime or provided by manufacturers' documentations. As a first step, we focus on Ethernet network cards. A similar model using a linear equation can be applied for wireless network cards [9], 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, etc.). Our network power model is therefore defined as:

$$P_{network}^{PID} = \frac{\sum_{i \in states} t_i \times P_i \times d}{t_{total}}$$

where P_i 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 with the network card.

Based on these models, our formula 1 is now equal to:

$$P_{software} = 0.7 \times TDP \times f \times V^2 \times \frac{t_{CPU}^{PID}}{t_{CPU}}(d) + \frac{\sum_{i \in states} t_i \times P_i \times d}{t_{total}}$$

IV. Implementation

PowerAPI is implemented as a system level modular library. We implemented so far the CPU and network modules and associated power models.

Our system-level library aims to provide power information per PID for each system component (CPU, NIC[3], etc.).

The library is therefore based on a modular approach where each system component is represented as a *power module*. *Power modules* operate independently of each other and are composed by two sub-modules: *formula* and *sensor*. These sub-modules communicate using the *Service-Oriented Architecture* (SOA) paradigm, and are deployed in an OSGi [4] gateway. In particular, we use *Service-Oriented Framework* (SOF[5]) to implement the various modules of PowerAPI in C++.

[2]http://linux.die.net/man/1/top
[3]Network Integrated Card
[4]formerly Open Service Gateway Initiative
[5]http://sof.tiddlyspot.com

The sensor sub-module is responsible for gathering hardware and operating system related information for the module. For example, it gathers the number of bytes transmitted by the network card, and the time spent by the CPU for each of the processor frequencies (when DVFS is supported). This sub-module is OS-dependent. We implemented sensor sub-modules for the CPU and NIC on a GNU/Linux operating system. In particular, our implementation exploits system information available in the *procfs* [10] and *sysfs* [11] file systems.

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

Additionally, our library supports the lifecycle management of its power modules. The latter can be started, stopped and their parameters changed at runtime, using a *modules manager*. The benefit of this modular approach is to offer flexibility while monitoring the system.

Validation

We validate the accuracy and precision of POWERAPI prototype on a Dell Precision T3400 workstation with an Intel Core 2 Quad processor (Q6600), running Ubuntu Linux 11.04. Our goal is to evaluate implementations of a CPU intensive algorithm, Towers of Hanoi. Therefore, we will only outline the results of the CPU module.

We compared the power values provided by POWERAPI with the actual power consumption of the computer using a bluetooth powermeter, PowerSpy[6].

First, we stress the processor using the Linux stress command[7]. Figure 2 depicts the results as an evolution of the CPU power consumption along time (normalized values). The peaks correspond to stressing 1, 2, 3 and 4 cores, respectively. Figure 3 shows the accuracy of our library where we outline the measured values using the powermeter and the values provided by POWERAPI, excluding the preliminary synchronization values. We also compare the values of our library and the powermeter when viewing a video using MPlayer[8] (results in Figure 4). Note that due to synchronization time lag between the Bluetooth powermeter and our library, values are shifted for a few seconds in the beginning of the monitoring. The time lag disappears after a couple of minutes.

The results show minor variations between the estimations computed by our library, and the real power consumption values. The margin of error is small in the core stressing and MPlayer scenarios at around 0.5% of the normalized and averaged values. Therefore, we can reasonably argue that using software-only approach, we provide values that are accurate enough to be used by energy management software.

[6]http://www.alciom.com/en/products/powerspy2.html
[7]http://linux.die.net/man/1/stress
[8]http://www.mplayerhq.hu

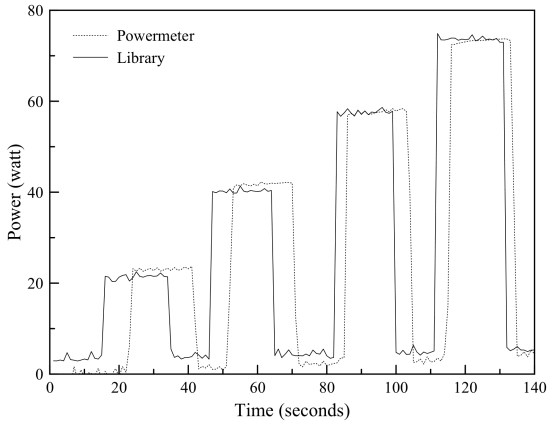

Figure 2. Stressing the processor cores using the STRESS command.

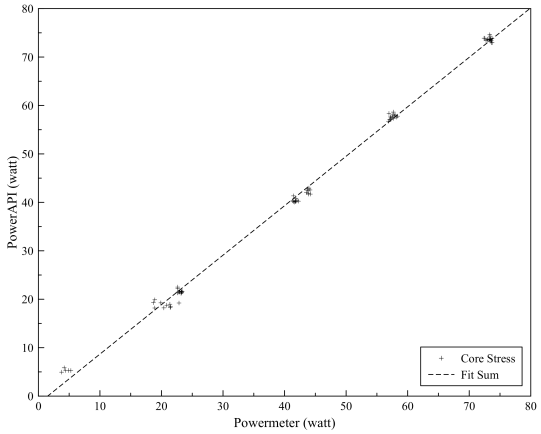

Figure 3. The accuracy of POWERAPI when stressing the processor cores.

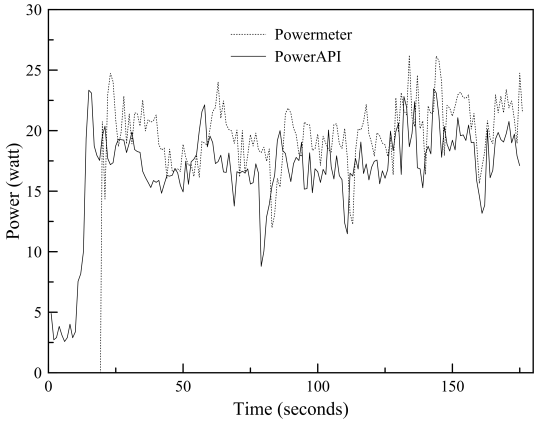

Figure 4. Running a video under MPlayer.

V. Impact of Algorithms and Programming Languages

We run our PowerAPI library on a number of implementations of the Towers of Hanoi mathematical program, all taken from the hanoimania page[9]. We measured the power consumption and execution time of eight implementations: three benchmarks of the iterative implementation in C++ including one using the O2 and O3[10] *GNU Compiler Collection* (GCC) compiler options, three benchmarks of the recursive implementation in C (including O2 and O3), and recursive versions of Java, Python, OCaml, Pascal, Prolog and Perl programming languages.

PowerAPI provides power values (in watt), and we configured our library to take measurements at a 200 ms interval or 5 Hz, and we run the Towers of Hanoi program with 30 disks. Figures 5 and 6 reports on the energy consumption (calculated in joule using the execution time) of the different implementations.

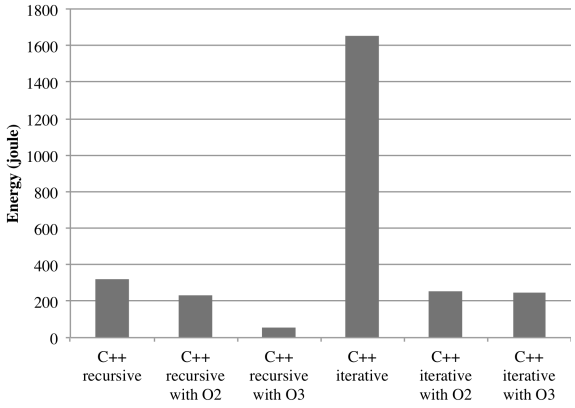

Figure 5. Energy consumption of the recursive and iterative implementations of Tower of Hanoi program in C++.

Overall, we found that all implementations use up to 100% of one of the processor cores. In average, this equals to around 18 watts in our host configuration (ranging from 17.5 to 18.2 watts). However, the execution time varies greatly between the implementations and the algorithms.

Recursive and iterative versions: We first compare the difference between the recursive and iterative versions of the Towers of Hanoi program. The recursive algorithm implemented in C++ consumed in average 322.23 joules while its iterative version consumed 1656.26 joules, more than 400% increase. When using the O2 optimization option during compilation, both versions exhibit a similar energy consumption (on average 7% difference), however with the O3 option, the iterative version does not save any energy. The recursive version shows a 78% decrease in energy consumption using the O3 option in comparison with the O2 option. O2 turn on more than 50 optimization flags in GCC and G++ compilers, while O3 adds 6 more flags including *Predictive Commoning* optimization (to eliminate redundancies across the iterations of a loop[11]) that recursive algorithms benefit from in comparison to iterative ones. Our results show that the recursive version of the Towers of Hanoi program is more energy efficient than its iterative version.

Implementations in programming languages: In addition to energy efficiency differences in algorithm implementations, we observe that the energy consumed by the same algorithm varies from language to language. Both C and C++ recursive versions show similar results (55.45 joules for C and 53.20 joules for C++ with O3). The C recursive version consumed nearly 268% more energy than its Java counterpart. Compiling the program with the O2 optimization option allowed a gain of around 20%, but still 188% more energy consuming than Java. However, when compiling the program with all optimizations (option O3), energy consumption was greatly reduced to become even better than Java (37% decrease of C compared to Java).

Contrary to some beliefs, our results show that Java, using the default options, is quite energy efficient in comparison to other programming languages. Using only default values, the Java recursive version is the more energy efficient in our tests. Only the C recursive version with the O3 option performs better.

We also observed the energy efficiency of the Pascal language, being on par with the default C or C++. However, other results are more interesting: Perl is the most consuming (25 516 joules, executing for 23 minutes), OCaml being second (with 17 852 joules and 16 minutes), then Python (9 450 joules and less than 9 minutes), and finally Prolog (3 673 joules with 3.5 minutes).

The difference between the most energy consuming (Perl) and the most energy efficient (C++ with O3) is quite high: 25 463 joules (or 47 863% increase). It is important to note here that our comparison tests are for one CPU-intensive application (Towers of Hanoi). Other tests are needed in order to make a proper conclusion about the energy efficiency of a programming language compared to another. However, our results offers a preliminary indication of energy-efficiency trends in programming languages for CPU intensive applications. Nevertheless, implementing additional tests is one of our future work directions.

It is also important to note that although a CPU time profiler can offer an idea about the energy efficiency of a Tower of Hanoi program (due to the fact that the algorithm uses 100% of the processor nearly all the time), it is not always the case. Our test on a video playback on MPlayer (cf. Figure 4) shows a variation of power consumption of

[9] http://www.kernelthread.com/projects/hanoi/

[10] http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[11] http://gcc.gnu.org/wiki/PredictiveCommoning

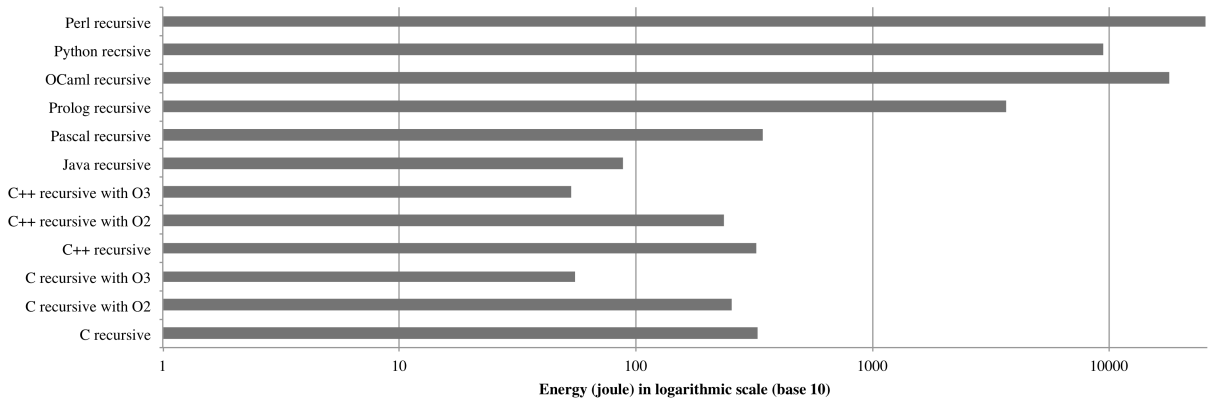

Figure 6. Energy consumption of the recursive implementation of Tower of Hanoi program in different languages (using a base 10 logarithmic scale).

more than 130% increase. In this situation, a CPU time profiler is rendered less useful for energy monitoring. When adding other hardware resources (such as the video card or the memory), a comprehensive power model therefore becomes crucial for energy monitoring.

VI. Related Works

Monitoring energy consumption of hardware components usually requires an hardware investment, like a multimeter or a specialized integrated circuit. For example in [3], the energy management and preprocessing capabilities is integrated in a dedicated ASIC (*Application Specific Integrated Circuit*). It continuously monitors the energy levels and performs power scheduling for the platform. However, this method has the main drawback of being difficult to upgrade to newer and more precise monitoring and it requires that the hardware component be built with the dedicated ASIC, thus making any evolution impossible without replacing the whole hardware.

On the other hand, an external monitoring device provides the same accuracy as ASIC circuits and does not prohibit energy monitoring evolutions. Devices, such as AlertMe Smart Energy[12], monitor home devices and allow users to visualize their energy consumption history through application services, such as the now defunct Google Powermeter[13]. However, these approaches do not adapt the system autonomously: the user takes the decision, while our approach opens up solutions for adapting applications with limited user intervention.

The previous monitoring approaches allow getting energy measures about hardware components only. However, knowing the energy consumption of software services and components requires an estimation of that consumption. This estimation is based on calculation formulae as in [7] and [12]. In [7], 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. In [12], the authors take into account the cost of the *wait* and *idle* states of the application (*e.g.*, an application consumes energy when waiting for a message on the network). In [4], the authors propose a tool, PowerScope, for profiling energy usages of applications. This tool uses a digital multimeter to sample the energy consumption and a separate computer to control the multimeter and to store the collected data. PowerScope can sample the energy usage by process. This sampling is more precise than energy estimation, although it still needs a hardware investment.

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. A Windows version is also available, so called pTopW, and offers similar functionalities, but using Windows APIs.

In addition to pTop, several utilities exist on Linux for resource profiling. For example, cpufrequtils[14], in particular cpufreq-info to get kernel information about the CPU (*i.e.*, frequency), and cpufreq-set to modify CPU settings, such as the frequency. *iostat*[15] that is used to

[12]http://www.alertme.com/smart_energy
[13]http://www.google.com/powermeter
[14]http://kernel.org/pub/linux/utils/kernel/cpufreq/cpufrequtils.html
[15]http://linux.die.net/man/1/iostat

get devices' and partitions' input/output (I/O) performance information, as well as CPU statistics. Other utilities [13] also exist with similar functionalities, such as SAR, MPSTAT, or the system monitoring applications available in Gnome, KDE or Windows. In [14], the authors propose GREEN TRACKER, a tool that presents an insight of the energy consumption of different software groups by estimating their CPU utilization. However, all of these utilities only offer raw data (*e.g.*, CPU frequency, utilized memory) and do not offer power information.

Our approach is more flexible and evolutive than PTOP. In addition to offering process-level energy information, POWERAPI 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.

VII. CONCLUSION AND FUTURE WORKS

In this paper, we present the POWERAPI architecture. It allows gathering and calculating the power consumption of processes and applications. We also propose energy models to calculate the energy consumption. Our models use and extend the state-of-the-art models and formulae. Our initial results show the potential of our approach for comparing, at runtime, the power consumption of applications and algorithms. As for future work, we plan to: *i)* propose more energy models for other hardware resources (in particular, memory and disk); *ii)* propose power monitoring at a finer grain, such as Java threads and methods; *iii)* extend our experimentation to more applications and algorithms, such as JETTY application server or network intensive applications; and *iv)* use power-aware information to adapt applications at runtime based on energy concerns.

ACKNOWLEDGMENT

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

REFERENCES

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

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

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

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

[5] T. Do, S. Rawshdeh, and W. Shi, "pTop: A Process-level Power Profiling Tool," in *HotPower'09: Proceedings of the 2nd Workshop on Power Aware Computing and Systems*, 2009.

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

[7] C. Seo, S. Malek, and N. Medvidovic, "An energy consumption framework for distributed java-based systems," in *ASE'07: Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 421–424.

[8] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis, "JouleSort: a balanced energy-efficiency benchmark," in *SIGMOD'07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007, pp. 365–376.

[9] L. Feeney and M. Nilsson, "Investigating the energy consumption of a wireless network interface in an ad hoc networking environment," in *INFOCOM 2001: 20th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, 2001, pp. 1548–1557.

[10] Erik Mouw, "Linux Kernel Procfs Guide," http://www.compsoc.man.ac.uk/~moz/kernelnewbies/documents/kdoc/2.5/procfs-guide.pdf, 2001.

[11] Patrick Mochel, "The sysfs Filesystem," http://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf.

[12] A. Kansal and F. Zhao, "Fine-grained energy profiling for power-aware application design," *SIGMETRICS Perform. Eval. Rev.*, vol. 36, no. 2, pp. 26–31, 2008.

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

[14] N. Amsel and B. Tomlinson, "Green tracker: a tool for estimating the energy consumption of software," in *CHI EA'10: Proceedings of the 28th of the international conference extended abstracts on Human factors in computing systems*. ACM, 2010, pp. 3337–3342.