The Next 700 CPU Power Models

Maxime Colmant, Romain Rouvoy, Mascha Kurpicz, Anita Sobe, Pascal Felber, Lionel Seinturier

The Next 700 CPU Power Models

Maxime Colmant[a,b], Romain Rouvoy[a,c], Mascha Kurpicz[d], Anita Sobe[d], Pascal Felber[d], Lionel Seinturier[a]

[a] *University of Lille / Inria, France*
[b] *ADEME, France*
[c] *Institut Universitaire de France (IUF)*
[d] *University of Neuchâtel, Switzerland*

Abstract

Software power estimation of CPUs is a central concern for energy efficiency and resource management in data centers. Over the last few years, a dozen of ad hoc power models have been proposed to cope with the wide diversity and the growing complexity of modern CPU architectures. However, most of these CPU power models rely on a thorough expertise of the targeted architectures, thus leading to the design of hardware-specific solutions that can hardly be ported beyond the initial settings. In this article, we rather propose a novel toolkit that uses a configurable/interchangeable learning technique to automatically learn the power model of a CPU, independently of the features and the complexity it exhibits. In particular, our learning approach automatically explores the space of hardware performance counters made available by a given CPU to isolate the ones that are best correlated to the power consumption of the host, and then infers a power model from the selected counters. Based on a middleware toolkit devoted to the implementation of software-defined power meters, we implement the proposed approach to generate CPU power models for a wide diversity of CPU architectures (including Intel, ARM, and AMD processors), and using a large variety of both CPU and memory-intensive workloads. We show that the CPU power models generated by our middleware toolkit estimate the power consumption of the whole CPU or individual processes with an accuracy of 98.5% on average, thus competing with the state-of-the-art power models.

Keywords: power models, energy monitoring, software toolkit, software-defined power meters, open testbed

1. Introduction

Energy represents one of the largest cost factors when operating data centers, largely due to the consumption of air conditioning, network infrastructure, and host machines [1]. To monitor this energy consumption, *power distribution units* (PDU) are often shared amongst several machines to deliver aggregated power consumption reports, in the range of hours or minutes. This node-level power consumption mostly depends on the software systems hosted by the machines and optimizing their power efficiency therefore requires to support process-level power monitoring in real-time, which goes beyond the capacity of PDUs [2]. Among the key components, the CPU is considered as the major power consumer [3] within a node and requires to be closely and accurately monitored.

However, developing power models that can accurately cover different features of a single CPU (*e.g.*, multi-threading, frequency scaling) is a complex task, and supporting the diversity of features on different CPU architectures is even more challenging. In particular, the wide variety of power models proposed in the literature [4–8] along the last years demonstrates that there is no single model that can fit the diversity of CPU architectures while delivering acceptable accuracy and runtime performance. More specifically, we observed that the state-of-the-art in this domain faces several key limitations, such as a simplified CPU architecture [9], the deprecation of the CPU model [10], the unavailability of the selected hardware performance counters [11], the design of hand-crafted power models [12], the unavailability of the considered benchmarks [11], and the limited diversity of tested workloads [13], to name a few.

To overcome these limitations—and rather than proposing yet another power model—we introduce PowerAPI, a framework that adopts a modular approach to automatically learn the power model of a target CPU by exploring its consumption space according to a variety of input workloads. In particular, our toolkit automatically builds the power model of a CPU by characterizing the available power-aware features, and then exploits this model in production to estimate the power consumption. Thanks to PowerAPI, these power estimations can be delivered in real-time both at the granularity of the physical node and of the hosted software processes. Our solution is assessed on 4 different CPUs (Intel Xeon, Intel i3, ARM Cortex, AMD Opteron) and exhibits, on average, an error

Email addresses: maxime.colmant@inria.fr (Maxime Colmant), romain.rouvoy@univ-lille.fr (Romain Rouvoy), pascal.felber@unine.ch (Pascal Felber), lionel.seinturier@univ-lille.fr (Lionel Seinturier)

rate of only 1.5%. Regarding the limitations identified in the previous paragraph, we show that PowerAPI can *i)* benefit from widely available open-source benchmarks and workloads to learn and compare power models, *ii)* build an accurate power model according to the list of available hardware performance counters and power-aware features. We claim that such a toolkit will be able to cope with the continuous evolution of CPU architectures and to cover the "next 700 CPU power models",[1] thus easing their adoption and wide deployment.

Beyond the CPU power models and experiments we report in this article, we believe that our contribution offers an open testbed to foster the research on green computing. Our open-source solution can be used to automatically infer the power models and to support the design of energy-aware scheduling heuristics in homogeneous systems [4, 15–18], as well as in heterogeneous data centers [19], to serve the energy-proportional computing [20–23] and to evaluate the effectiveness of optimizations applied on binaries [24]. It also targets system administrators and software developers alike in monitoring and better understanding the power consumption of their software assets [25–27]. The middleware toolkit we developed, named PowerAPI, is published as open-source software[2] and we build on freely available benchmark suites to learn the power model of any CPU.[3]

The remainder of this article is organized as follows.

Section 2 discusses the state-of-the-art of CPU power models learning techniques and theirs limitations, and the state-of-the-art of monitoring tools used to produce power estimation at different granularities. Section 3 describes in depth the architecture of our middleware toolkit, PowerAPI, for learning the CPU power models and building software-defined power meters. Section 4 proposes a new approach to automatically learn the power model for a given CPU architecture. Section 5 demonstrates the accuracy of various CPU power models that our approach can infer. Section 6 studies the applicability of our approach to model specific processor architectures. Finally, Section 7 concludes this article and sketches some perspectives for further research.

2. State-of-the-Art

In this Section, we describe the previous research works which are close to our contributions. Section 2.1 introduces all recent research approaches for learning power models, while Section 2.2 presents different tools that can estimate or measure the power consumption at different granularities.

[1] Following the same idea of generality as in the seminal paper of Landin: *The Next 700 Programming Languages* [14].

[2] Freely available as open source software from: `http://powerapi.org`

[3] Freely available as Docker containers from: `https://github.com/Spirals-Team/benchmark-containers`

2.1. CPU Power Models

The closest approach to hardware-based monitoring is *Running Average Power Limit* (RAPL), introduced with the Intel "Sandy Bridge" architecture to report on the power consumption of the entire CPU package. As this feature is not available on other architectures and is not always as accurate as one could expect [5], alternative CPU power models are generally designed based on a wider diversity of raw metrics. Along the last decade, the design of CPU power models has therefore been regularly considered by the research community [4–8]. The state-of-the-art in this area has investigating both power modeling techniques and system performance metrics to identify power models that are as accurate and/or as generic as possible.

In particular, standard operating system metrics (CPU, memory, disk, or network), directly computed by the kernel, tend to exhibit a large error rate [6, 8]. Contrary to these high-level usage statistics, *Hardware Performance Counters* (HPCs) are low-level metrics that can be monitored from the processor (*e.g.*, number of retired instructions, cache misses, non-halted cycles). However, modern processors offer a variable number of HPC, depending on architectures and generations. As shown by Bellosa [4] and Bircher [34], some HPCs are highly correlated with the processor power consumption whereas the authors in [31] conclude that several performance counters are not useful as they are not directly correlated with dynamic power. Nevertheless, this correlation depends on the processor architecture and the CPU power model computed using some HPCs may not be ported to different settings and architectures. Furthermore, the number of HPC that can be monitored simultaneously is limited and depends on the underlying architecture [35], which also limits the ability to port a CPU power model on a different architecture. Therefore, finding an approach to select the relevant HPC represents a tedious task, regardless of the CPU architecture.

Power modeling often builds on these raw metrics to apply learning techniques—for example based on sampling [13]—to correlate the metrics with hardware power measurements using various regression models, which are so far mostly linear [7].

A. Aroca *et al.* [28] propose to model several hardware components (CPU, disk, network). They use the `lookbusy` tool to generate a CPU load for each available frequency and a fixed number of active cores. They capture *Active Cycles Per Second* (ACPS) and raw power measurements while loading the CPU. A polynomial regression is used for proposing a power model per combination of frequency and number of active cores. They validate their power models on a single processor (Intel Xeon W3520) by using a map-reduce Hadoop application. During the validation, the authors have not been able to correctly capture the input parameter of their power model—*i.e.*, the overall CPU load—and they use an estimation instead. The resulting "tuned" power model with all components together

Author(s)	Processor(s)	Feature(s)	Regression(s)	Benchmarks	Error(s)
A.Aroca *et al.* [28]	Xeon W3530	HW sensors	polynomial, multiple linear	*eval.:* Hadoop App.	$< 7\%$ of total energy
Bertran *et al.* [13]	Core 2 Duo	14 HPCs regrouped by component	multiple linear by component	*sampl.:* μ-benchs *eval.:* SPEC CPU 06	5%
Bircher *et al.* [9]	Pentium 4	μ-ops trace-cache, micro-code ROM	multiple linear	*sampl.:* SPEC CPU 00 *eval.:* SPEC CPU 00	2.5%
Contreras *et al.* [29]	XScale PXA255	5 HPCs	multiple linear	*eval.:* SPEC CPU 00, Java CDC/CLDC	4%
Dolz *et al.* [30]	Xeon E3-1275	3 HPCs HW sensors	linear	*sampl.:* linpack, stream, iperf, IOR *eval.:* Quantum Espresso	3 W 70 W max.
Economou *et al.* [3]	Turion, Itanium 2	HW sensors	multiple linear	*sampl.:* Gamut *eval.:* SPECs, Matrix, Stream	5%
Isci *et al.* [10]	Pentium 4	15 HPCs	multiple linear	*eval.:* μ-benchs, AbiWord, Mozilla, Gnumeric	3 W
Li *et al.* [12]	8-way issue superscalar	IPC	linear	*sampl./eval.:* DB, email, SPEC JVM 98, SPEC INT 95	1% off. 6% run.
Rivoire *et al.* [31]	Core 2 Duo & Xeon, Itanium 2, Turion	HW sensors HPCs	multiple linear	*sampl.:* calibration suite *eval.:* SPECs, stream, Nsort	$< 5\%$
Yang *et al.* [32]	Xeon E5620 & E7530	7components 91 preselected	support vector	*sampl.:* NPB, IOzone, CacheBench *eval.:* SPEC CPU 06, IOzone	4.7%
Zamani *et al.* [33]	Opteron	3 to 5 HPCs	ARMAX	*sampl./eval.:* BT.C, CG.C, LU.C, SP.C	0.1% – 0.5% offline
Zhai *et al.* [11]	Sandy Bridge	non-halted cycles	linear	*eval.:* Google, SPEC CPU 06	7.5%

Table 1: Summary of existing CPU power models.

exhibits an error rate of 7% compared to total amount of energy consumed.

Bertran *et al.* [13] model the power consumption of an Intel Core 2 Duo by selecting 14 HPCs based on an *a priori* knowledge of the underlying architecture. To compute their model, the authors inject both selected HPCs and power measurements inside a multivariate linear regression. A modified version of `perfmon2`[4] is used to collect the raw HPC values. In particular, the authors developed 97 specific micro-benchmarks to stress in isolation each component identified. These benchmarks are written in C and assembly and cannot be generalized to other architectures. The authors assess their solution with the SPEC CPU 2006 benchmark suite, reporting an error rate of 5% on a multi-core architecture.

Bircher *et al.* [9] propose a power model for an Intel Pentium 4 processor. They provide a first model that uses the number of fetched μ-operations per cycle, reporting an average error rate of 2.6%. As this model was performing better for benchmarks inducing integer operations, the authors refine their model by using the definition of a floating point operation. As a consequence, their second power model builds on 2 HPCs: the μ-operations delivered by the trace cache and the μ-operations delivered by the μ-code ROM. This model is assessed using the SPEC CPU 2000 benchmark suite, which is split in 10 groups. One benchmark is selected per group to train the model and the remaining ones are used to assess the estimation. Overall, the resulting CPU power model reports an average error of 2.5%.

In [29], the authors propose a multivariate linear CPU power model for the Intel XScale PXA255 processor. They additionally consider different CPU frequencies on this processor to build a more accurate power model. They carefully select the HPCs with the best correlation while avoiding redundancy, resulting in the selection of only 5 HPCs. In their paper, they also consider the power drawn by the main memory using 2 HPCs already used in the CPU power model. However, given that the processor can only monitor 2 events concurrently, they cannot implement an efficient and usable runtime power estimation. They test their solution on SPEC CPU 2000, Java CDC, and Java CLDC, and they report an average error rate of 4% compared to the measured average power consumption.

The authors in [30] propose an approach to build linear power models for hardware components (CPU, memory, network, disk) by applying a per component analysis. Their technique uses 4 benchmarks during the training phase and collect various metrics gathered from hardware performance counters, OS statistics, and sensors. They build a correlation matrix from all gathered metrics (including power measurements) and then apply a clustering algorithm on top of it. The power models reported from this state-of-the-art work are manually extracted from these groups. Without considering the power models which include directly power measurements, the best one exhibits an absolute error of 3 W on average with a maximum absolute error of 70 W.

Economou *et al.* [3] model the power consumption of 2 servers (Turion, Itanium) as a multiple linear regression that uses various utilization metrics as input parameters. The authors use the CPU utilization, the off-chip memory access count, the hard-disk I/O rate, and the network I/O rate. The input parameters are learned by using Gamut that emulates applications with varying levels of CPU, memory, hard disk, and network utilization. In order to retrieve raw power measurements, the authors uses board-level modifications and 4 "power planes" (extracted from the paper), which is a rather heavy and constraining mechanism for end-users and represents a major hardware investment. On average, their power models exhibit an error rate of less than 5% (varying between 0% and 15% in all cases) when using SPEC benchmarks, matrix and stream.

Isci and Martonosi [10] use an alternative approach to estimate the power consumption of an Intel Pentium 4 processor. They isolate 22 processor subunits with the help of designed micro-benchmarks and live power measurements. For each subunit, they use simple linear heuristics, which can include one or more HPCs. For the others (trace cache, allocation, rename...), they use a specific piecewise linear approach. They selected 15 different HPCs to model all subunits, some of them are reconfigured or rotated when needed. At the end, they express the CPU power consumption as the sum of all subunits. They train their model on designed micro-benchmarks, SPEC CPU 2000 and some desktop tools (AbiWord, Mozilla, Gnumeric) and they report an average error of 3 W.

Li and John [12] rely on per OS-routines power estimation to characterize at best the power drawn by a system. They simulate an 8-way issue,[5] out-of-order superscalar processor with function unit latency. The authors use 21 applications, including SPEC JVM 98 and SPEC INT 95. During their experiments, they identify *Instruction Per Cycle* (IPC) to be very relevant to model the power drawn by the OS routines invoked by the benchmarks. The resulting CPU power model exhibits an average error of up to 6% in runtime testing conditions.

Rivoire *et al.* [31] propose an approach to generate a family of high-level power models by using a common infrastructure. In order to choose the best input metrics for their power models, they compare 5 types of power models that vary on their input metrics and complexity. The first 4 power models are defined in the literature and use basic OS metrics (CPU utilization, disk utilization) [31, 36, 37]. They propose the fifth power model that uses HPCs in addition to CPU and disk utilization. The last proposed power model exhibits a mean absolute error lesser than

[4] `http://perfmon2.sourceforge.net`

[5] As mentioned in the aforementioned publication.

4% over 4 families of processors (Core 2 Duo, Xeon, Itanium, Turion) when using SPECfp, SPECint, SPECjbb, stream, and Nsort. The authors do not detail the underlying architectures of the testbed CPU, thus making a fair comparison difficult.

iMeter [32] covers not only CPU, but also memory and I/O. To get a practical model, the authors need to select the proper number of counters. After benchmarking VMs under different loads, they empirically extract 91 out of 400 HPCs. In a second step, a principal component analysis (PCA) is applied to identify a statistical correlation between the power consumption and the performance counters. With this method, highly correlated values are clustered into a smaller set of principal components that are not correlated anymore. The selection of the principal components depends on the cumulative contribution to the variance of the original counters, which should reach at least 85%. The final model is derived by the usage of support vector regression and 3 manually selected events per principal component [38] and reports an average error of 4.7%.

In [33], the authors study the relationship between HPC and power consumption. They use 4 applications from NAS parallel benchmarks (BT.C, CG.C, LU.C, SP.C) running on 8 threads in a dual quad-core AMD Opteron system. Given the limitation on the events that they can open simultaneously, the authors first show that the measurement variability over different executions is not very significant, enabling different runs for sampling all events. This article proposes a deep analysis for HPC selection (single or multiple). The authors demonstrate that a collinearity can exist between events and then propose a novel method to find the best combination of HPC with good performance. They use the ARMAX technique to build their power models. They evaluate their solution by producing a model per application and exhibit a mean absolute error in signal between 0.1%-0.5% for offline analysis.

HaPPy [11] introduces an hyperthread-aware power model that uses only the *non-halted cycles* event. The authors distinguish between different cases where either single or both hardware threads of a core are in use. This power model is linear and contains a ratio computed according to their observations. They demonstrate that when both threads of a core are activated, they share a small part of non-halted cycles. The authors extend the `perf`[6] tool to access to RAPL. Their model is tested on a Intel "Sandy Bridge" server with private benchmarks provided by Google, which cannot be reused, and 10 benchmarks taken from SPEC CPU 2006. To assess their power estimation, they used the RAPL interface reachable on this server. Compared to RAPL, they manage to have an average error rate of 7.5%, and a worst case error rate of 9.4%. Nevertheless, as demonstrated in [5], these error rates can be exceeded in scenarios where only single cores of a CPU are monitored.

[6] `https://perf.wiki.kernel.org`

2.2. Software-Defined Power Meters

Software-defined power meters are customizable and adaptable solutions that can deliver raw power consumptions or power estimation at various frequencies and granularity, depending on the requirements. Power estimation of running processes is not a trivial task and must tackle several challenges. Several solutions are already proposed by the state-of-the-art:

pTop [39] is a process-level power profiling tool for Linux or Windows platforms. pTop uses a daemon in background for continuously profiling statistics of running processes. pTop keeps traces about component states and stores temporarily the amount of energy consumed over each time interval. This tool displays, similarly to the `top` output, the total amount of energy consumed—*i.e.*, in Joules—per running process. The authors propose static built-in energy models for CPU, disks, and network components. An API is also provided to get energy information of a given application per selected component.

PowerScope [40] is a tool capable of tracking the energy usage per application for later analysis and optimizations. Moreover, it maps the energy consumption to procedures within applications for better understanding the energy distribution. The tool uses 2 computers for offline analysis: one for sampling system activities (*Profiling* computer) and another for collecting power measurements from an external digital multimeter (*Data Collection* computer). Once the profiling phase completed, the *Profiling* computer is then used to compute all energy profiles for later use.

PowerTOP [41] is a Linux tool for finding energy-consuming software on multiple component sources (*e.g.*, CPU, GPU, USB devices, screen). Several modes are available, such as the calibrate mode to test different brightness levels as well as USB devices, or the interactive mode for enabling different energy saving mechanisms not enabled by default. This software-defined power meter can only report on power estimation while running on battery within an Intel laptop, or only usage statistics otherwise.

SPAN [42] is designed for providing real-time power phases information of running applications. It also offers external API calls to manually allow developers to synchronize the source-code applications with power dissipation. The authors first design micro benchmarks for sampling the only HPC used—*i.e.*, IPC—and they gather the HPC data and raw power measurements for computing the parameters of their hand-crafted power models. They can next use SPAN for real-time power monitoring and offline source-code analysis.

Different limitations can be extracted from these solutions and the state-of-the-art. Most solutions are monolithic, created for specific needs and cannot be easily tuned or configured for assembling new kind of power meters [39–42]. Power models used by such power meters cannot be

used or adapted to modern architectures because they have been designed for a specific task [42] or depend on specific metrics [39] that cannot trustfully represent recent power-aware features. They also lack modularity [42] and may require additional investments [40].

In order to detect energy-consuming applications and to apply critical energy decisions at runtime, one rather needs a middleware toolkit that is fully modular, configurable, and adaptive to report on power estimation or power measurements at high frequency on heterogeneous systems. For promoting the usage of such a solution as a good alternative of physical power meters, the proposed solution has to be freely available for the community.

Synthesis and Contribution. In this article, we clearly differ from the state-of-the-art by providing an open source, modular, and completely configurable implementation of a power modeling toolkit: PowerAPI. For the purpose of this article, we base our analysis on standard benchmarks (*e.g.*, PARSEC, NAS NPB) to provide an open testbed for building and comparing CPU power models. We believe that our implementation can be easily extended to other domain or synthetic workloads [43] depending on requirements (cf. Section 6.1). To the best of our knowledge, PowerAPI is the first solution that embeds a configurable and extensible learning approach to identify the power model of a CPU, independently of the features it exhibits. Once learned, the power models are then used by PowerAPI to accurately produce process-level power estimation in production at high frequencies. Unlike the existing approaches already published in the literature, we rather propose an interchangeable and configurable approach that is *i)* architecture-agnostic and *ii)* processor-aware.

3. PowerAPI, a Middleware Toolkit to Learn Power Models

We propose and design PowerAPI, an open-source toolkit[7] for assembling software-defined power meters upon need. PowerAPI is built on top of Scala and the actor programming model using the Akka library. Akka is an open-source toolkit for building scalable and distributed applications on the JVM, which pushes forward the actor programming model as the best programming model for concurrency. The software components of PowerAPI are implemented as actors, which can process millions of messages per second [44], a key property for supporting real-time power estimation. PowerAPI is therefore fully asynchronous and scales on several dimensions—*i.e.*, the number of input sources, the requested monitoring frequencies, and the number of monitoring targets.

More specifically, the PowerAPI toolkit identifies 5 types of actor components:

Clock actors are the entry point of our architecture and allow us to meet throughput requirements by emitting ticks at given frequencies for waking up the other components.

Monitor actors handle the power monitoring request for one or several processes. They react to the messages published by a clock actor, configured to emit tick messages at a given frequency. The monitor is also responsible for aggregating the power estimation by applying a function (*e.g.*, SUM, MEAN, MAX) defined for the monitoring when needed.

Sensor actors connect the software-defined power meters to the underlying system in order to collect raw measurements of system activity. Raw measurements can be coarse-grained power consumption reported by third-party power meters and embedded probes (*e.g.*, RAPL), or fine-grained CPU activity statistics as delivered by the process file system (`ProcFS`). Sensors are triggered according to the requested monitoring frequency and forward raw measurements to the appropriate formula actor.

Formula actors use the raw measurements received from the sensor to compute a power estimation. A formula implements a specific power model [6, 8] to convert raw measurements into power estimation. The granularity of the power estimation reported by the formula (machine, core, process) depends on the granularity of the measurements forwarded by the sensors.

Reporter actors finally give the power estimation computed by the aggregating function to a Display object. The Display object is responsible to convert the raw power estimation and the related informations (*e.g.*, the timestamp, the monitoring id or the devices) into a suitable format. The delivered report is then exposed, for instance via a web interface, via a virtual file system (*e.g.*, based on FUSE),[8] or can be uploaded into a database (*e.g.*, InfluxDB).[9]

Sensor and Formula actors are tightly coupled and we group them as a PowerModule that links the input metrics and the power model.

The overall architecture of PowerAPI is depicted in Figure 1. Several PowerModule components can be assembled together for grouping power estimation from multiple sources. One can see that PowerAPI is fully modular, can be used to assemble power meters upon needs and to fulfill all monitoring requirements. We can also note that PowerAPI is a non-invasive solution and does not require costly investments or specific kernel updates.

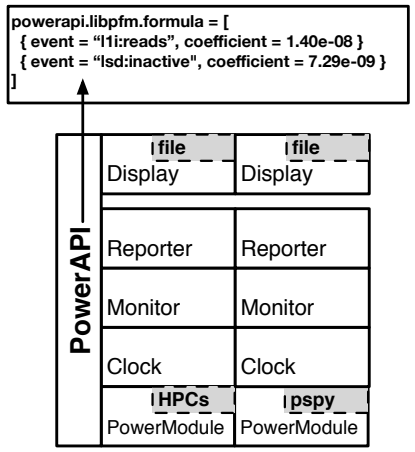
Three instances of software-defined power meters are also depicted in the figure. On the left side (Figure 1a), one can find an instance of PowerAPI especially configured to learn the CPU power models. This instance is composed of two PowerModule components, one for retrieving

[7] Freely available from: `http://powerapi.org`

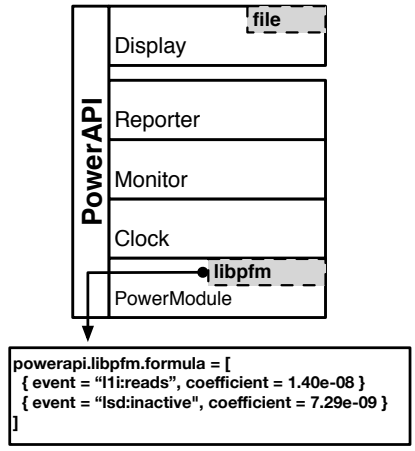
[8] `https://github.com/libfuse/libfuse`

[9] `https://www.influxdata.com/time-series-platform/influxdb`

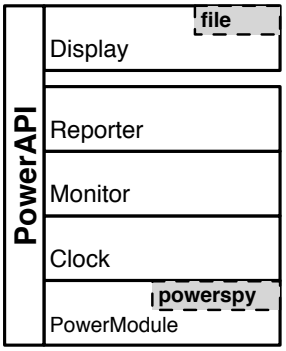
Figure 1: Overview of the PowerAPI's architecture with 3 instances of software-defined power meter built with it. The leftmost instance (Figure 1a) shows the component assembly used in Section 4 to learn CPU power models, while the middle and the rightmost instances (Figures 1b and 1c resp.) represent the assemblies made in Section 5 to compare real-time power estimation with measurements.



(a) Software-defined power meter used for learning CPU power models.



(b) Software-defined power meter that uses the learned power model for real-time power estimation.



(c) Software-defined power meter that collects power measurements from PowerSpy.

raw accurate CPU metrics via `libpfm`, and another, for retrieving the power measurements from a Bluetooth power meter. The data are then forwarded to several files to be later processed by our learning approach explained below. The resulting power model is then written inside a configuration file that can be used later by a new instance of PowerAPI to estimate the power consumption in production. In the middle (Figure 1b), another instance of PowerAPI is configured to use the aforementioned power model for producing fine-grained power estimation. And finally, on the right side (Figure 1c), an instance of PowerAPI is configured to retrieve the power measurements from a bluetooth power meter, PowerSpy, in real-time.

In this article, we use PowerAPI *i)* to learn the power models presented in Section 4, *ii)* to validate our learning approach and compare it against the state-of-the-art in Section 5, and *iii)* to build several software-defined power meters in Section 6.

4. Learning CPU Power Models

While state-of-the-art CPU power models demonstrate that achieving accurate power estimation is possible (cf. Table 1), most of the past contributions are barely generalizable to processors or applications that were not part of the original study. Therefore, instead of trying to design yet another power model, we rather propose an approach capable of learning the specifics of a processor and building the fitting CPU power model. Hence, our solution intends to cover evolving architectures and aims at delivering a reusable solution that will be able to deal with current and future generations of CPU architectures.

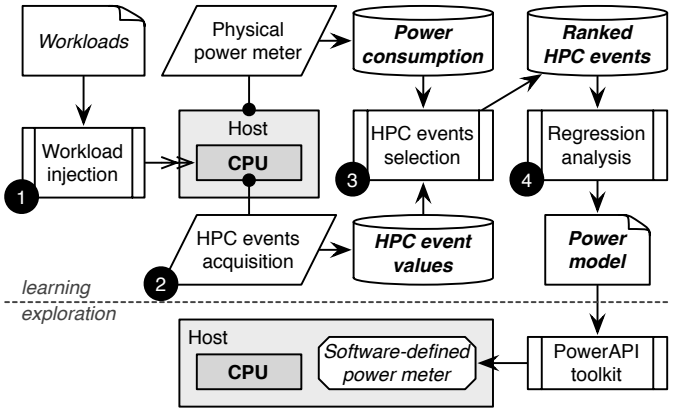
As reported by [6], the CPU load does not accurately reflect the diversity of the CPU activities. In particular, to faithfully capture the power model of a CPU, the types of tasks that are executed by the CPU have to be clearly identified. We therefore decide to base our power models on *hardware performance counters* (HPCs) to collect raw, yet accurate, metrics reflecting the types of operations that are truly executed by the CPU. However, the number and the nature of HPC events provided by the CPU strongly vary according to the processor type.

Table 2: Examples of PMUs detected for 4 processors from 3 manufacturers, including the numbers of generic counters and available events.

Manuf.	CPU	PMU	#Gen.	#Ev.
Intel	Xeon W3520	nhm	4	338
		nhm_unc	8	176
	i3 2120	snb	4	336
		snb_unc_cbo0	2	19
		snb_unc_cbo1	2	18
AMD	Opteron 8354	fam10h_barcelona	3	421
ARM	Cortex A15	arm_ac15	6	67

More specifically, a CPU can expose several *performance monitoring units* (PMUs) depending on its architecture and model. For example, 2 PMUs are detected on an Intel Xeon W3520: nehalem and nehalem uncore, each providing 2 types of HPCs that cover either *fixed* or *generic* HPC events. A fixed HPC event can only be used for one predefined event, usually cycles, bus cycles, or instructions retired, while a generic HPC can monitor any event. If there are more events monitored than available counters for a PMU, the kernel applies multiplexing to alter the frequency and to provide a fair access to each HPC

Figure 2: Overview of the 2-phase proposed approach.



event. When multiplexing is triggered, the events cannot be monitored accurately anymore and estimation are returned instead.

As shown in Table 2, the number of generic counters supported and the number of available events varies considerably across architectures and even among models of the same manufacturer.

The approach we propose consists of two phases: an *offline learning* phase and an *online exploitation* phase, as depicted in Figure 2. The learning phase analyses the power consumption and triggered HPC events of the target CPU, in order to identify the key HPC events that impact the power consumption. These key events are combined in a power model—*i.e.*, a formula—which is then used during the online exploitation phase to deliver real-time power estimation by feeding a *software-defined power meter*, built with PowerAPI, that embeds the inferred CPU power model (cf. Figure 1b).

4.1. Learning Phase

During the learning phase, our goal is to automatically classify the HPC events in order to identify those that are best characterizing the CPU activity and are tightly correlated with its power consumption. In the following paragraphs, we describe each of the steps illustrated in Figure 2.

Input workload injection. To explore the diversity of activities of a CPU, we consider a set of representative benchmark applications covering the features provided by a CPU. In particular, to promote the reproducibility of our results, we favor freely available and widely used benchmark suites, such as PARSEC [45]. We publish several of these open source benchmarks, as well as PowerAPI, as Docker containers that can be easily deployed from DockerHub to host machines in order to quickly learn their power model using our toolkit.[10] However, this choice does not prevent from including additional (micro-)benchmark suites or any other sample workloads to study their impact on the inferred power models. During this step, PowerAPI takes care of launching the selected workloads several times—3 times here—and in isolation for reducing the potential noise that can be experienced during the learning phase and thus improve the reliability of the monitoring phase.

Acquisition of raw HPC counters Technically, the CPU cannot monitor hundreds of HPC events simultaneously [35], without triggering an event multiplexing mechanism, which degrades the accuracy of the collected metrics. Thus, we have to split the list of available events into subsets of events to avoid this multiplexing mechanism that might cause inaccuracies in our process. Based on the information gathered in Table 2—the numbers are automatically computed in the beginning of our approach—we compute the number of events that can be monitored in parallel. For a given CPU, the number of workload executions w to be considered is therefore defined as:

$$w = \left\lceil \sum_{p \in PMUs} \frac{|E_p|}{|C_p|} \right\rceil \times |W| \times i \qquad (1)$$

where E is the set of events made available by the processor for a given PMU, C is the set of generic counters available for a PMU, W is the set of input workloads, and i is the number of sampling iterations to execute.

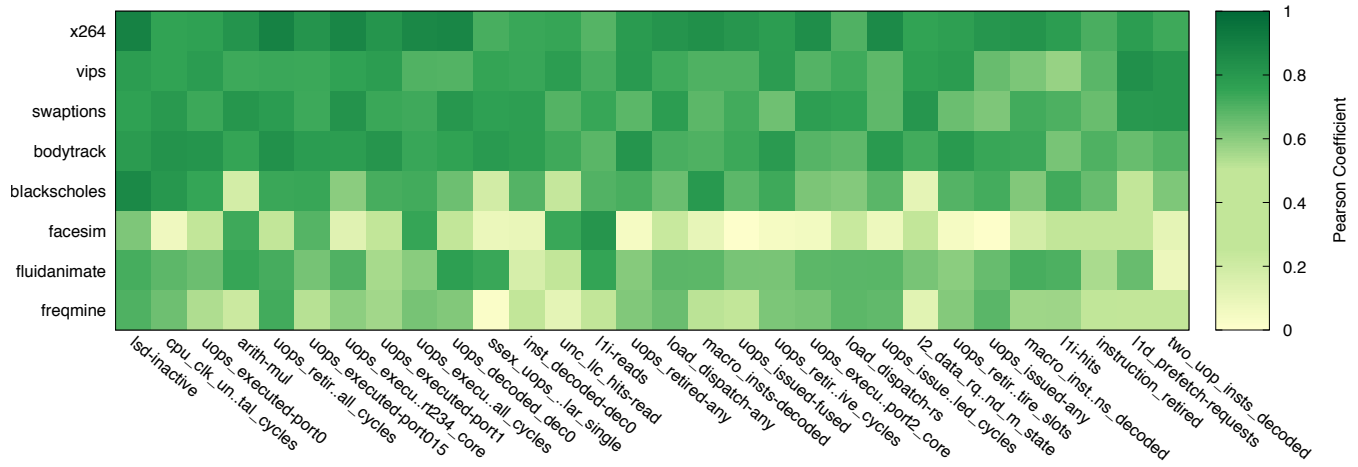
Combining HPC events and sample applications may quickly lead to the comparison of thousands of candidate metrics. Hence, a filtering step is introduced to guarantee an acceptable duration for the learning phase. Our approach therefore proposes an automated way to focus on the most relevant events. In the first step, each workload is only executed for 30 seconds (configurable) while collecting values from HPC events and from a power meter. We then filter out the most relevant HPC events by applying the Pearson correlation coefficient. We compute the Pearson correlation coefficient $r_{e,p}$ for each workload between the n values reported by each monitored HPC event e and the collected power consumption p:

$$r_{e,p} = \frac{\sum_{i=1}^{n} (e_i - \overline{e})\ (p_i - \overline{p})}{\sqrt{\sum_{i=1}^{n} (e_i - \overline{e})^2}\ \sqrt{\sum_{i=1}^{n} (p_i - \overline{p})^2}} \qquad (2)$$

Selection of relevant HPC events As next step, we eliminate the HPC events that have a median correlation coefficient ($\tilde{r}$) below a configurable threshold. In particular, we consider that any coefficient below

[10] Open source benchmark containers: `https://github.com/Spirals-Team/benchmark-containers`

Figure 3: Pearson coefficients of the Top-30 correlated events for the PARSEC benchmarks on the Intel Xeon W3520.



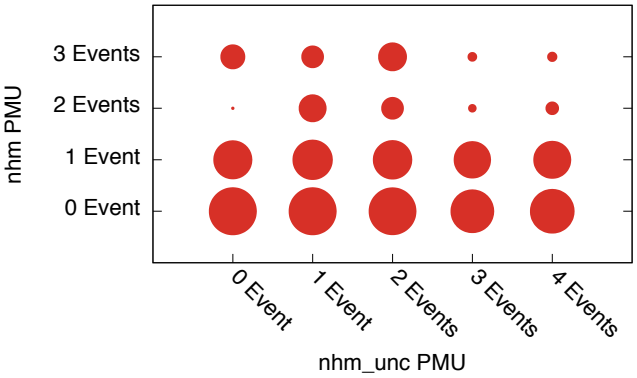
0.5—empirically found here—clearly indicates a lack of correlation between the considered event (e) and power consumption (p). During this step, we can quickly filter out hundreds of uncorrelated—and therefore irrelevant—events, resulting for instance in 253 left out of 514 events on an Intel Xeon W3520. The reduced set of HPC events is then used to relaunch all the workloads, but this time with the default runtime of the selected benchmarks. At the end of the full execution, we rank the remaining HPC events for all the workloads based on their newly calculated median correlation with the power consumption, as depicted in Figure 3. This figure illustrates the distribution of Pearson coefficients for the 30 best events, which varies for each of the workloads (W) taken from the PARSEC benchmark suite on the Intel Xeon W3520 processor. One can clearly distinguish the benchmarks that stimulate all selected HPC events (*e.g.,* `x264`, `vips`) from the ones whose power consumptions match only some specific events (*e.g.,* `freqmine`, `fluidanimate`).

Power model inference by regression We finally apply a regression analysis to derive the CPU power model from the previously selected HPC events. In particular, we consider that the power model of a host machine, $P(host)$, is defined as:

$$\begin{aligned} P(host) &= P_{idle}(host) + P_{active}(host) \\ &= \sum_{c \in C} P^c_{idle}(host) + \sum_{p \in P(host)} \sum_{c \in C} P^c_{active}(p) \end{aligned} \quad (3)$$

where $P_{idle}(host)$ refers to the idle consumption of the host machine—*i.e.*, its power consumption when the node does not host any activity—and $P_{active}(host)$ refers to the dynamic consumption due to the activity of each hosted process p. Both of these idle and dynamic consumptions encompass all the hardware components c supported by the host machine C (CPU, motherboard, fans, memory, disk, etc.). As part of this article, we consider both CPU- and memory-intensive workloads, thus modeling the power consumption of these components (CPU, memory) and their side-effects on related components (motherboard, fans, etc.).

Figure 4: Average error per combination of events for R_3 (`freqmine`, `fluidanimate`, `freqmine`) on an Intel Xeon W3520 (log scale). The larger the circle, the higher the average error.



While the state-of-the-art has investigated several regression techniques, from linear [12, 30], to polynomial ones [5, 28], this article reports on the adoption of the robust ridge regression [46, 47], which belongs to the family of multivariate linear regressions. This technique has been chosen to eliminate outliers and to limit the effect of collinearity between variables—*i.e.*, HPC events in our case. Unlike least squares estimates for regression models, the robust regression better tolerates non-normal measurement errors, which may happen when combining power measurements and HPC events.

The computation of the multiple linear regression should

balance the gain in terms of estimation error with the cost of including an additional event into the CPU power model. To design the CPU power model as accurately as possible, we consider a training subset R_n of n benchmarks ($\forall n < |W|, R_n \subseteq W$)—composed from those exhibiting the lowest median Pearson coefficients—as input for our regression process. From R_n, we compute a CPU power model for each combination of HPC events, by taking into account the limited number of events that can be monitored concurrently. For each training set R_n, and from all the computed power models, we only keep the one with the smallest regression error. Finally, we compare the CPU power model obtained for each R_n and we pick the one that minimizes the absolute error between the regression and the remaining benchmarks, which have not been included in the training set ($E_n = W \setminus R_n$).

As an illustration, Figure 4 depicts the distribution of the average error per CPU power model built for R_3 (`freqmine`, `fluidanimate`, and `facesim`) depending on the number of HPC events included in the model. A larger circle means a larger error. One can clearly see that a CPU power model that combines several HPC events may exhibit a larger error than one that uses a lower number of events. As an example, on the Intel Xeon W3520, the CPU power model composed of 2 events taken from the PMU `nhm` (see Table 2) emerges from this analysis and reports an average error of 1.35%, which corresponds to 1.60 W.

All the above steps allow PowerAPI to compute a CPU power model that can be effectively used in production to estimate the power consumption in real-time. For the purpose of this article, we configure PowerAPI with a frequency of 1 Hz during the learning phase. As a matter of comparison, this approach takes 34 hours approximately on an Intel i3 2120 CPU (373 events available) whereas it takes 16 hours approximately on an ARM Cortex A15 CPU (67 events available).

As already mentioned, this approach is not tightly coupled to the adoption of a robust ridge regression, but extremely modular since we already learned and implemented alternative CPU power models using PowerAPI [5, 26], as reported in Section 4.

The resulting CPU power models are then used as input parameters of a *software-defined power meter* that can estimate the power consumption of the CPU in real-time, thanks to PowerAPI. In the context of this article, we use a `libpfm`[11] sensor actor on the host nodes to collect the HPC events. `libpfm` can list and access available HPC counters on most of modern architectures, regardless of the operating system. The sensor actor forwards the collected HPC counters to a formula actor, which embeds the learned CPU power model. Finally, a reporter actor formats the output of the power model and communicates to various outputs to be configured by the user (*e.g.*, a web interface or a file).

[11] `http://perfmon2.sourceforge.net`

Table 3: Processor architecture specifications.

Manuf.	Intel	Intel	AMD	ARM
CPU	Xeon	i3	Opteron	Cortex
Model	W3520	2120	8354	A15
Freq.	2.66 GHz	3.10 GHz	2.2 GHz	2.32 GHz
Design	4 cores × 2 threads	2 cores × 2 threads	16 cores	4 cores + 1 lp core
TDP	130 W	55 W	95 W	< 15 W[a]
SMT	✓	✓	✗	✗
DVFS	✓	✓	✗	✗
Turbo	✓	✗	✗	✗

[a] Empirically found.

As shown in Figure 1, the PowerAPI toolkit therefore provides a flexible way to assemble software-defined power meters on demand. The experiments and results described later in this article are all based on various software-defined power meters built with this middleware toolkit.

5. Assessing CPU Power Models

To assess the versatility of our approach, we consider several heterogeneous processor architectures that exhibit different characteristics and combinations of power-aware features, as illustrated in Table 3 and described below.

Simultaneous Multi-Threading (SMT) is used to separate each core into several hyper-threads. The technology allows the processor to seamlessly support *thread-level parallelism* (TLP) in hardware and share more effectively the available resources. Performance gains strongly depend on software parallelism, and for a single-threaded application it may be more effective to actually disable this technology.

Dynamic Voltage/Frequency Scaling (DVFS) allows a processor to adjust its clock speed and run at different frequencies or voltages upon need. The OS can increase the frequency to quickly execute operations or reduce it to minimize dissipated power when the processor is under-utilized.

TurboBoost (Turbo) can dynamically increase the processor frequency beyond the maximum bound, which can be greater than the *thermal design power* (TDP), for a limited period of time. It therefore allows the processor cores to execute more instructions by running faster. TurboBoost is however only activated when some specific conditions are met, notably related to the number of active cores and the current CPU temperature. It also depends on the OS, which may request to trigger it when some applications require additional performance.

C-states (CS) were introduced to save energy and allow the CPU to use low-power modes. The idea is to lower the clock speed, turn off some of the units on the processor, and reduce the power consumed. The

more units are shut down, the higher are the power savings. Different types of C-states are available: core (individual hardware core's view, CC-state), processor (global hardware core's view, PC-state), or logical (logical core's view, CC-state). They are numbered starting at 0, which corresponds to the most operational mode (100 %). Higher the index is, deeper the sleep state is and higher the time required to wake-up an unit is.

For assessing the CPU power models, we use PowerAPI and compare the power estimation resulting from the learned models with raw power measurements from a wall-plugged Bluetooth power meter, PowerSpy.[12] Depending on the country and thus the current frequency, the PowerSpy power meter samples the overall power consumption of a system between 45 and 65 Hz with a maximum error rate of 1%. The readings of the power meter are collected simultaneously to the HPC events with PowerAPI at a frequency of 1 Hz.

We use the well-known PARSEC [45] v2.1 benchmark suite to evaluate our approach. PARSEC includes emerging applications in *recognition, mining, and synthesis* (RMS) as well as systems applications that mimic large-scale multi-threaded commercial programs. This benchmark suite is diverse in terms of working set, locality, data sharing, synchronization, and off-chip traffic, thus making it well-designed to stress multi-core architectures.

The remainder of this section details the CPU power models that are automatically learned by our approach for each of the 4 CPUs introduced in Table 3. Each power model described below is embedded within a formula actor, part of a software-defined power meter, for producing real-time power estimation. Their accuracy is then evaluated with regards to physical power measurements collected from PowerSpy.

As previously described in Section 4.1, we split the set of 8 benchmarks (W) into two sets: *i)* those used to learn the CPU power model (R), and *ii)* the remaining ones used for the purpose of validation (E). Given that we focus on CPU- and memory-intensive systems in this article, we report on the power drawn by a system, which is described as follows: $P = P_{idle} + P^{CPU}_{active}$, where P_{idle} corresponds to the overall static power consumption and P^{CPU}_{active} to the dynamic power consumption drawn by the CPU.

5.1. Intel Xeon W3520

System configuration
This server is configured to run a Linux Ubuntu 14.04 (kernel 3.13) with the vanilla configuration.

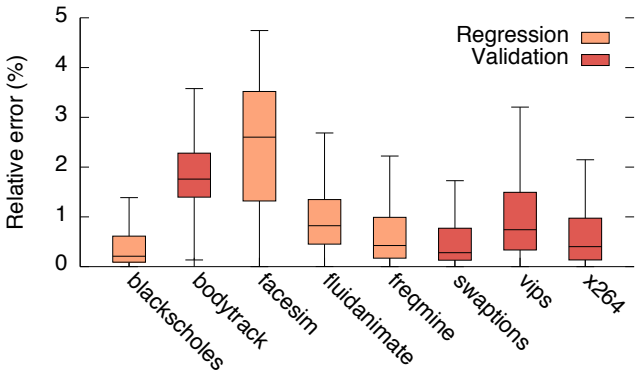
CPU power model definition
The CPU power model automatically computed by PowerAPI with the training subset of benchmarks, R_4 (blackscholes, facesim, fluidanimate, freqmine), comprises 2 HPC events from the PMU nhm ($e1$ = l1i:reads[13], $e2$ = lsd:inactive[14]):

$$P_{idle} = 92\,\text{W}$$
$$P^{CPU}_{active} = \frac{1.40 \cdot e_1}{10^8} + \frac{7.29 \cdot e_2}{10^9}$$

To assess the effectiveness of the robust ridge regression, we inspect the *eigenvalues* of corresponding correlation matrix. Very low values (close to zero, 10^{-3}) in the resulting matrix denote a collinearity between variables. The selected events have *eigenvalues* of 1.5 and 0.5, confirming the non-collinearity of the HPC events included in this CPU power model.

CPU power model accuracy

Figure 5: Relative error distribution of the PARSEC benchmarks on the Xeon processor ($P_{idle} = 92\,\text{W}$).



Our approach automatically isolates the idle consumption of the node whose relationship to TDP is defined in [25, 48] as $P \simeq P_{idle} + 0.7 \times TDP$. Regarding the dynamic power consumption of the host machine, Figure 5 reports an average relative error of 1.35 % (1.60 W) for the subset of benchmarks that were not included in the sampling set (bodytrack, swaptions, vips, x264). In particular, this figure reports on the comparison between the power estimation produced with PowerAPI and a PowerSPY Bluetooth power meter while running the PARSEC benchmark suite. The software-defined power meter built with PowerAPI has been configured with a frequency of 1 Hz on an Intel Xeon W3520.

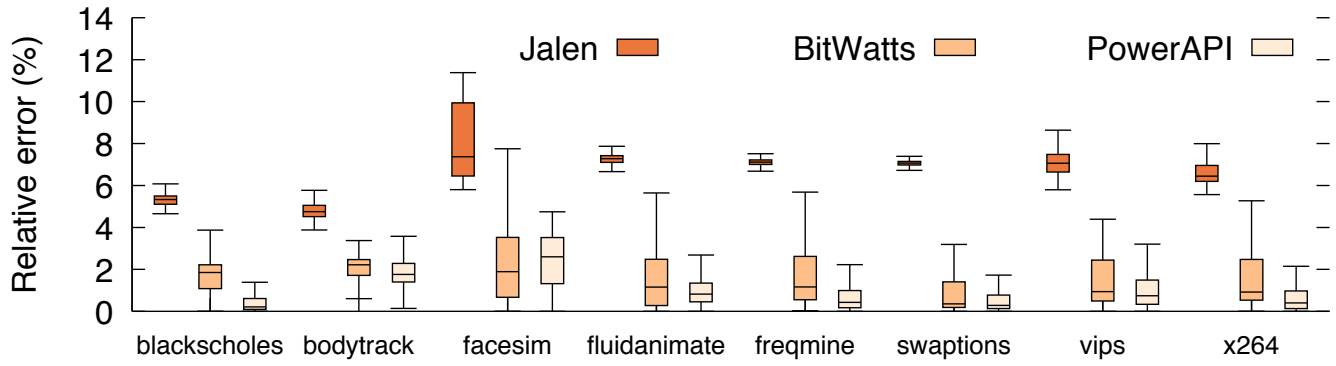
To confirm the accuracy of the generated CPU power model, we propose to compare these results with state-of-the-art power models that we can reproduce. In particular,

[12] `http://www.alciom.com/en/products/powerspy2-en-gb-2.html`

[13] l1i:reads counts all instruction fetches, including uncacheable fetches that bypass the L1I

[14] lsd:inactive counts all cycles when the loop stream detector emits no uops to the RAT for scheduling

Figure 6: Comparison of the errors exhibited from the power models learned with the approaches presented in [25] (Jalen), [5] (BitWatts), and the one presented in this article (PowerAPI), for the Xeon W3520 processor $P_{idle} = 92\,\text{W}$.



we selected 2 CPU power models [5, 25] that can be deployed on this server configuration. The CPU power model Jalen proposed in [25] is an architecture-agnostic power model that estimates the power consumption from a linear regression combined with the processor's TDP value. The CPU power model BitWatts described in [5] is an Intel-specific power model that is built from a polynomial regression applied on a set of manually selected HPC events. Thanks to PowerAPI, we reused and/or developed the corresponding Sensor and Formula actors (cf. Section 3) to collect power estimations for the purpose of this comparison. As demonstrated in Figure 6, our learning approach clearly improves the accuracy of these state-of-the-art CPU power models on such a server configuration for 6 workloads out of 8, while it performs similarly to BitWatts for the remaining ones (bodytrack and facesim). By outperforming Jalen and providing more accurate estimations than BitWatts, PowerAPI therefore demonstrates the effectiveness of our approach. Nonetheless, accuracy is not the only objective of PowerAPI and in the following sections, we demonstrate that our approach succeeds to automatically build similar CPU power models for any CPU architectures, including those that are not covered by the state-of-the-art.

5.2. Intel i3 2120

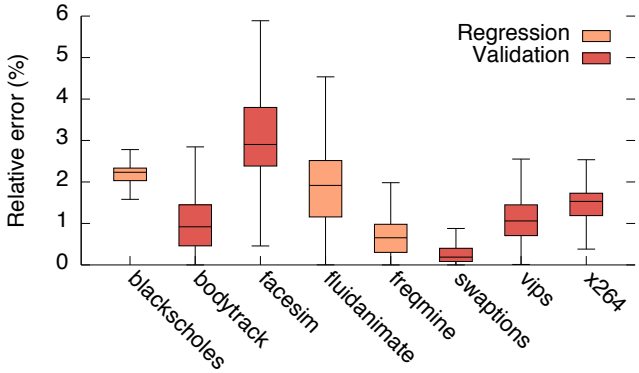
System configuration
This server is configured to run a Linux Ubuntu 14.04 (kernel 3.13) with the vanilla configuration.

CPU power model definition
The resulting CPU power model computed by PowerAPI with a training subset, R_3, is composed of 2 HPC events from PMU snb (e_1 = idq:empty[15], e_2 = uops_dispatched:stall_cycles[16]) and 1 HPC event from PMU snb_unc_cbo0 (e_3 = unc_clockticks[17]):

$$P_{idle} = 30\,\text{W}$$
$$P^{CPU}_{active} = \frac{1.12 \cdot e_1}{10^8} + \frac{4.55 \cdot e_2}{10^9} + \frac{6.89 \cdot e_3}{10^{10}}$$

CPU power model accuracy
Although it uses a similar configuration and the same subset of the 4 benchmarks described in Section 5.1, the resulting CPU power model strongly differs from the power model computed for the Xeon. Yet, Figure 7 reports a relative error of 1.57 % (0.71 W), on average, which confirms the accuracy of our CPU power models for Intel architectures, beyond the Intel-specific models previously published [5].

Figure 7: Relative error distribution of the PARSEC benchmarks on the i3 processor ($P_{idle} = 30\,\text{W}$).



[15] idq:empty counts cycles the *Instruction Decode Queue* (IDQ) is empty.

[16] uops_dispatched:stall_cycles counts number of cycles no *micro-operation* (μops) were dispatched to be executed on this thread.

[17] unc_clockticks is not officially documented.

Beyond Intel architectures, PowerAPI can also address alternative CPU processors provided by AMD and ARM.

5.3. AMD Opteron 8354

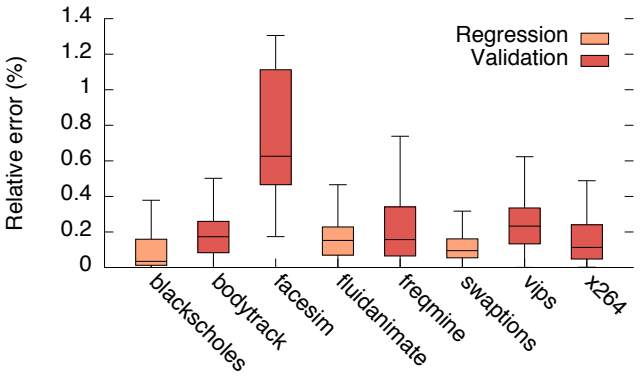
System configuration
This server is configured to run a Linux Ubuntu 14.04 (kernel 3.13) with the vanilla configuration.

CPU power model definition
The resulting CPU power model computed by PowerAPI from the training subset, R_3, is composed of 3 HPC events from the PMU fam10h_barcelona (e_1 = probe:upstream_non_isoc_writes[18], e_2 = instruction_cache_fetches[19], e_3 = retired_mmx_and_fp_instructions:all[20]):

$$P_{idle} = 390\,\mathrm{W}$$
$$P^{CPU}_{active} = -\frac{2.63 \cdot e_1}{10^4} + \frac{8.20 \cdot e_2 + 3.16 \cdot e_3}{10^9}$$

CPU power model accuracy
Figure 8 reports a relative error of 0.20 % (0.81 W), on average. While most of the works in the state-of-the-art focus on Intel architectures (cf. Table 1), the accuracy of the CPU power model we generate for the AMD Opteron configuration assesses our capability to cover alternative CPU architectures. Given the particularly high value of the idle consumption ($390\,W$) imposed by the infrastructure setup (cluster configuration), this relative error raises up to 8.45 % ($0.77\,W$) if we compute this relative error by only considering the dynamic power consumption—*i.e.*, by subtracting P_{idle} from raw measurements.

Figure 8: Relative error distribution of the PARSEC benchmarks on the Opteron processor ($P_{idle} = 390\,\mathrm{W}$).



While PowerAPI can accurate power estimations for power-consuming servers, it can also be used to estimate the power consumption of energy efficient systems running atop of ARM processors.

5.4. ARM Cortex A15

System configuration
Our last configuration is a Jetson Tegra K1[21] with Linux Ubuntu 14.04 (kernel 3.10). The processor has 4 plus 1 cores on its chip, designed and optimized by NVIDIA. The 4 cores have a standard behavior, while the additional core is designed to be energy efficient. These cores are exclusive—*i.e.*, we cannot use both configurations together. By default, the 4 cores are enabled, and only a manual action can put the processor in low power mode. We first use this default behavior for the purpose of validation.

CPU power model definition
The resulting CPU power model computed by PowerAPI from R_4 is composed of 3 HPC events from the PMU arm_ac15 (e_1 = cpu_cycles[22], e_2 = inst_spec_exec_integer_inst[23], e_3 = bus_cycles[24]):

$$P_{idle} = 3.5\,W$$
$$P^{CPU}_{active} = \frac{1.18 \cdot e_1}{10^9} + \frac{1.26 \cdot e_2}{10^{10}} + \frac{1.84 \cdot e_3}{10^{11}}$$

CPU power model accuracy
Figure 9 reports a relative error of 2.70 % (0.17 W), on average. This error rate has to be balanced with the low idle consumption of this CPU, compared to previous configurations. Nonetheless, this CPU power model demonstrates that PowerAPI can also generate accurate CPU power models for embedded systems, thus going beyond standard server settings.

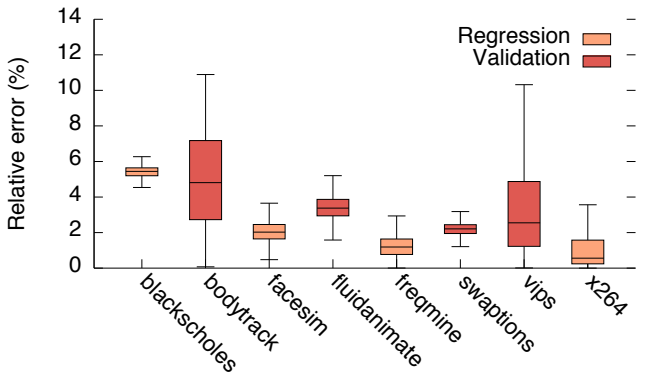
5.5. Synthesis

From all the above experiments and observations, we can assess that the generated CPU power models perform well on a variety of representative architectures, including Intel, ARM, and AMD. Our solution does not rely on a specific processor extension (*e.g.*, RAPL) and can use specific workloads during the learning phase to build domain-specific CPU power models. On average, our solution exhibits a relative error of 1.5 % (0.8 W), not only clearly outperforming the state-of-the-art, but also offering a reproducible approach for comparing CPU power models.

The closest method to ours, described in [3], builds a CPU power model for the whole testbed system and use it then inside their software solution, Mantis, for power estimation. Their power model is composed of 4 metrics,

[18] probe:upstream_non_isoc_writes is not officially documented.
[19] instruction_cache_fetches is not officially documented.
[20] retired_mmx_and_fp_instructions:all counts different classes of *Floating Point* (FP) instructions.
[21] https://developer.nvidia.com/jetson-tk1
[22] cpu_cycles is not officially documented.
[23] inst_spec_exec_integer_inst counts the integer data processing instructions speculatively executed.
[24] bus_cycles is not officially documented.

Figure 9: Relative error distribution of the PARSEC benchmarks on the A15 processor ($P_{idle} = 3.5\,\mathrm{W}$).



the CPU utilization, the memory access count, hard disk, and network I/O rates, and exhibits an error range between 0% and 15%. However, the key limitations of their solution are that *i)* they use a predefined set of metrics, which can clearly differ between architectures, *ii)* they use a heavy subsystem of power planes to get the power consumption of components for their offline power modeling, and, *iii)* their solution produces only power estimation for components, not at process level.

Feng and Ge [49, 50] describe a solution for computing the power profiles of each component of a subsystem. A component power profile corresponds to its power footprint over a given period of time. Moreover, their solution allows them to get additional insights about the software power consumption as they propose the same kind of profile at the code level. However, this solution tends to be very intrusive by connecting to the hardware pins to collect the power consumption of each component, and they do not propose a proper way to estimate the power consumption of these components in production.

We strongly believe that our approach is well suited to explore the space of HPC events made available by the CPU and for profiling with accuracy the power consumption drawn by the CPU.

6. Building Software-Defined Power Meters

The learning and model generation approach we introduced in this paper are combined to build accurate CPU power models. All the described CPU power models are built to represent the overall power consumption of a node. They can also be used to produce accurate process-level power estimations when needed, thanks to the different modes exposed by the hardware counters (cf. Section 6.2 and Section 6.3). From our CPU power models, we can easily extract the idle power consumption of a node and then show its impact. In this section, we define and study various applicative scenarios and, in this process, attempt to answer specific questions regarding the effectiveness of our approach. Throughout this section, we keep using PowerAPI to build different software-defined power meters based on the CPU power models presented in this article.

6.1. Domain-specific CPU Power Models

Can we build CPU power models that better fit specific domains of applications?

In Section 5, we identified applications from the PARSEC benchmark suite as representative workloads for characterizing the power consumption of our processors. In particular, we focused on delivering CPU power models that can estimate the power consumptions of a wide diversity of applications. However, if one knows beforehand that a specific type of workload will run on a node, we can also use our approach to derive domain-specific CPU power models. As an example, we use a set of benchmarks from the well-known *NAS parallel benchmark* (NPB) suite [51] on the ARM Cortex A15, and derive a new power model specifically for this set of applications using our approach described in Section 4.1. NPB was designed to take advantage of highly parallel supercomputers and thus the implemented benchmarks represent CPU-intense workloads.

The resulting CPU power model computed by PowerAPI with the lowest average error is composed of 3 HPC events from the PMU arm_ac15: (e_1 = bus_read_access[25], e_2 = cpu_cycles[26], e_3 = bus_access[27]):

$$P_{idle} = 3.5\,\mathrm{W}$$
$$P^{CPU}_{active} = \frac{-1.72 \cdot e_1}{10^8} + \frac{1.52 \cdot e_2 - 5.08 \cdot e_3}{10^9}$$

In Figure 10, we depict the results and compare them with the original model derived in Section 5. We can see that a domain-specific model can improve the original one (PARSEC model) with an average relative error of 4 % (corresponding to 0.41 W).

In comparison, the PARSEC power model has an average relative error of 20 % (2.34 W), which shows the benefits of building domain-specific CPU power models. We are thus able to derive accurate CPU power models with our approach despite the diversity of benchmarks. To the best of our knowledge, our solution is the first to be open-source, configurable, and directly usable to build CPU power models.

6.2. Real-time Power Monitoring

Can we use the derived CPU power models to estimate the power consumption of any workload in real-time?

[25] bus_read_access is semantically meaningful.
[26] cpu_cycles is semantically meaningful.
[27] bus_access is semantically meaningful.

Figure 10: Absolute error distribution of the NPB benchmarks on the A15 processor by using the PARSEC and NPB power models ($P_{idle} = 3.5\,\mathrm{W}$).

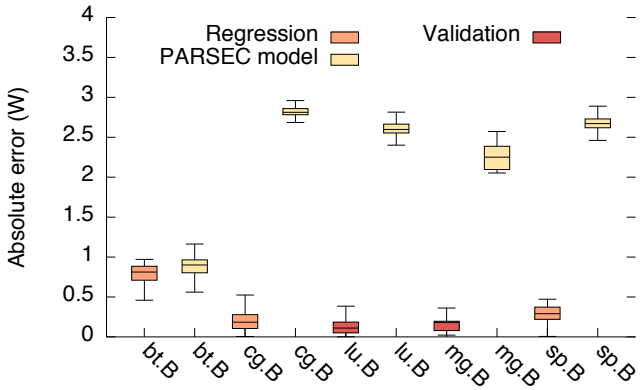
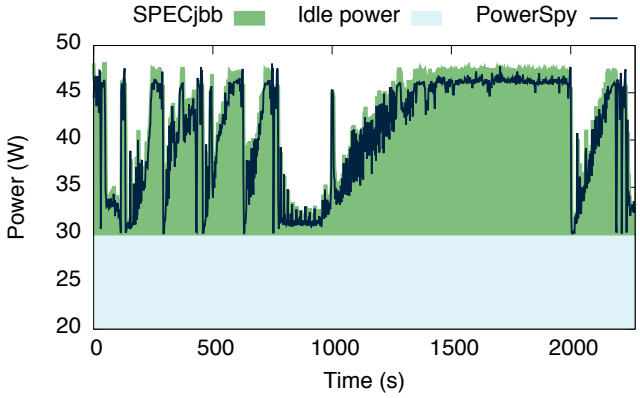


Figure 11: Power estimation delivered by PowerAPI in real-time (4 Hz) for SPECjbb 2013 ($P_{idle} = 30\,\mathrm{W}$).



To further evaluate the applicability of PowerAPI in a real-world and multi-threaded environment, we run the SPECjbb 2013 benchmark [52]. This benchmark implements a supermarket company that handles distributed warehouses, online purchases, as well as high level management operations (data mining). The benchmark is implemented in Java and consists of *controller* components for managing the application and *backends* that perform the actual work. A run takes approximately 45 minutes; it has varying CPU utilization levels and requires at least $2\,GB$ memory per backend to finish properly.

We use the Intel i3 2120 processor for this experiment with the CPU power model introduced in Section 5.2. Thanks to the capabilities of hardware performance counters, the power models which are generated at the host level can be directly used at the application—or process—level. Figure 11 illustrates the per-process power consumption, focused on the SPECjbb process, compared to physical power measurements. We can see that our system is capable of monitoring varying workloads with an average error of 1.6 % (1.70 W). These results clearly confirm that the learned power models can also be used to accurately model the software power consumption, going beyond the node-scale power models.

Regarding the monitoring frequency, PowerAPI is mostly limited by the frequency of hardware and software sensors used to collect runtime metrics. In particular, PowerAPI can report the power consumption of software processes up to 40 Hz when connected to the PowerSpy, and up to 10 Hz when using the `libpfm4` library—*i.e.*, when collecting HPC events. However, by increasing the monitoring frequency, one can observe that the stability of power estimation is affected, which does not help to properly identify the power consumption of processes.

6.3. Process-level Power Monitoring

Can we use the derived CPU power models to estimate the power consumptions of concurrent processes?

PowerAPI is an efficient toolkit that allows building software-defined power meters in order to perform fine-grained power estimation. We now show that our solution is not only able to automatically learn a CPU power model, but also to estimate the power consumption of concurrent processes running on the same CPU. We use the Intel Xeon W3520 processor for this experiment with the power model derived in Section 5.1.

Figure 12 illustrates the ability of the software-defined power meter built with PowerAPI to estimate with accuracy the power consumption of several processes running concurrently. In particular, it depicts the power distribution between the idle power consumption, one benchmark from the PARSEC suite (`freqmine`), and two others from the NPB suite (`bt.C` and `cg.C` configured here to run with 2 MPI processes). Compared to physical measurements, when running at a frequency of 4 Hz (one power estimation per 250 ms), our solution achieves a relative error of 2 % (2.92 W), thus competing with the state-of-the-art solutions [5, 9, 10, 29]. One can also notice the effectiveness of our solution even on the NPB suite, not used here during our learning phase.

Additionally, Figure 12 reports on the power consumption of PowerAPI along its execution. The power consumption of 2 W, on average, demonstrates that our implementation of the CPU power model has a reasonable energy footprint and is weakly impacted by the number of processes being monitored. This footprint acknowledges the design and the implementation of PowerAPI as a scalable system toolkit to build efficient software-defined power meters.

6.4. Adaptive CPU Power Models

Can we adjust the CPU power model depending on an execution profile?

Figure 12: Process-level power estimation delivered by PowerAPI in real-time (4 Hz) ($P_{idle} = 92\,\text{W}$).

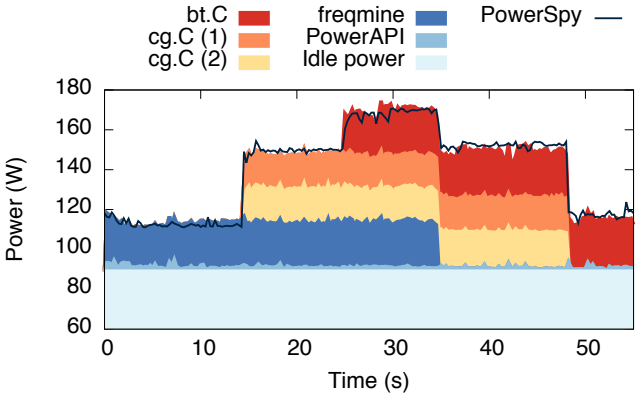
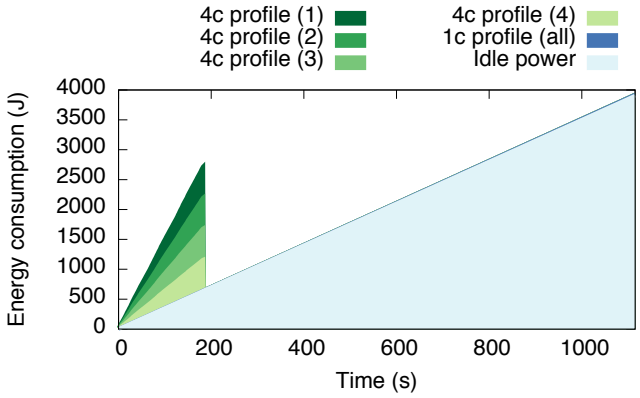


Figure 13: Energy consumption of the host by using the 4-plus-1 power profiles on the A15 processor and `cg.B` ($P_{idle} = 3.5\,\text{W}$).



In this experiment, we use the 4-plus-1-core processor available on the Tegra K1 card developed by NVIDIA. Based on the approach described above, we build different CPU power models in order to model the different modes—*i.e.*, when the 4 cores are enabled, or when the low power core is used. The first CPU power model is described in Section 5 and represents the processor power consumption when the 4 cores are in action. We are now interested in modeling the low power core. To trustfully represent the underlying optimizations, we therefore build a separate CPU power model for being able to distinguish the different profiles. For the low power core, the power model with the lowest absolute error is composed of 3 HPC events from the PMU arm_ac15 (e_1 = cid_write_retired[28], e_2 = ttbr_write_retired[29], e_3 = inst_spec_exec_load[30]):

$$P_{idle} = 3.5\,\text{W}$$
$$P^{CPU}_{active} = \frac{7.82 \cdot e_1 + 4.38 \cdot e_2}{10^4} + \frac{3.67 \cdot e_3}{10^{10}}$$

This model is very different from the one presented in Section 5 as the number and the types of events differ. To better understand the difference between both profiles, we plot the energy consumption of each profile for the benchmark `cg.B` (taken from NPB) spawned on 4 processes in Figure 13. The energy consumption is either shared between 4 cores (4c profiles) to optimize the performance, or 1 core (1c profile) when the low power mode is enabled. One can observe that the 4-cores profile completes 6 times faster by exploiting the parallelism of the underlying architecture, resulting in much lower energy consumption ($2.7\,\text{KJ} < 4\,\text{KJ}$). The 1-core profile exhibits a low power consumption, but is penalized by the idle power accumulating over time.

[28] cid_write_retired counts the writes to "CONTEXTIDR".
[29] ttbr_write_retired counts the writes to "TTBR".
[30] inst_spec_exec_load counts the load instructions speculatively executed.

In comparison to existing solutions, these power profiles were derived automatically without a deep expertise of the underlying architectures and they can be used directly in our middleware toolkit, PowerAPI. Our solution is then able to detect which mode is enabled and to adapt the CPU power model accordingly at runtime for estimating the power consumption of software assets. Our approach therefore captures all the features enabled on the processor to build adjusted CPU power models.

6.5. System Impact on CPU Power Models

Does the CPU power model depend on the underlying operating system or performance profiles?

We already showed that the CPU optimizations can have a non-negligible impact on the power consumption. Additionally, several hardware optimizations are controlled by the operating system, such as frequency scaling or hyper-threading.

We now compare the power consumption of 2 popular open source operating systems: Ubuntu and CentOS. Ubuntu is known as user-friendly, with a huge community of users and the philosophy of supporting a wide variety of systems, from servers to mobile devices. CentOS is derived from the open-source version of *Red Hat Enterprise Linux* (RHEL) and hence targets productivity systems that require a stable and dependable OS. Some very useful tools are available on this system for optimizing the hardware and software.

A version of Ubuntu 14.04 with a Linux kernel 3.13 and a version of CentOS 7 with a Linux kernel 3.10 were installed on the Intel Xeon W3520. We use the benchmark `bt` from the NPB suite.

For our experiment, we compare 3 CPU power models. One was already presented in Section 5.1 and we use the default settings without specific behavior (`U.def`).

The second model represents the default CPU settings of CentOS (`C.def`). The CPU power

model is composed of 4 HPC events from the PMU nhm (e_1 = uops_retired:active_cycles[31], e_2 = uops_issued:any[32], e_3 = ssex_uops_retired:scalar_single[33], e_4 = uops_retired:retire_slots[34]):

$$P_{idle} = 92\,\text{W}$$
$$P^{CPU}_{active} = \frac{2.02 \cdot e_1}{10^8} + \frac{7.76 \cdot e_2 + 4.43 \cdot e_3 + 2.70 \cdot e_4}{10^9}$$

The third model covers the performance optimizations provided by CentOS (`C.perf`). We use the `tuned-adm` tool for improving performance in specific use cases and for interacting with the power saving mechanisms. This command comes with different tuning server profiles depending on the use of the underlying system and hardware. We use the latency-performance profile, which allows the operating system to lower the latency of the system and thus to increase performance of a virtual guest; it is thus well suited to reduce swapping when CentOS is installed in a VM.

The CPU power model computed for CentOS with the latency-performance profile is composed of 4 HPC events from the PMU nhm (e_1 = l1d_prefetch:triggers[35], e_2 = uops_decoded_dec0[36], e_3 = fp_comp_ops_exe:sse_fp_scalar[37], e_4 = l1i:reads[38]):

$$P_{idle} = 125\,\text{W}$$
$$P^{CPU}_{active} = \frac{8.86 \cdot e_1}{10^8} + \frac{7.93 \cdot e_2 + 6.33 \cdot e_3 + 5.38 \cdot e_4}{10^9}$$

One can observe that the idle consumption (P_{idle}) of this power model is higher than the one isolated in the previous power model. This results directly from the choice of selecting the latency-performance profile to keep all the cores active. The operating system implements this policy by disabling the C-states CPU feature, which impacts the idle consumption of the host. While the operating system cannot benefit from C-states to optimize the energy consumption, the induced overhead is therefore associated to the idle consumption of the host as it cannot be fairly distributed among processes. The average power consumption values reported by the CPU power models are depicted in Figure 14.

[31] uops_retired:active_cycles counts the number of cycles with μops retired.

[32] uops_issued:any counts the number of cycles when μops are issued by the *Register Alias Table* (RAT) to *Reservation Station* (RS).

[33] ssex_uops_retired:scalar_single counts *Streaming SIMD Extensions* (SSE) scalar single-precision FP μops retired.

[34] uops_retired:retire_slots counts number of retirement slots used when μops are retired.

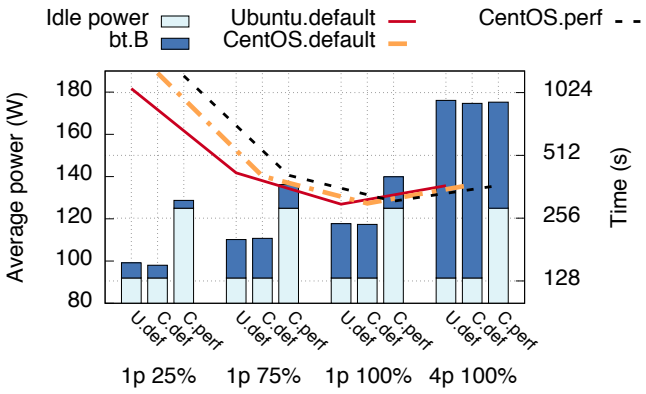
[35] l1d_prefetch:triggers counts number of prefetch requests triggered by the *Finite State Machine* (FSM) and pushed into the prefetch FIFO.

[36] uops_decoded_dec0 counts μops decoded by decoder 0.

[37] fp_comp_ops_exe:sse_fp_scalar counts number of SSE FP scalar μops executed.

[38] l1i:reads counts all instruction fetches.

Figure 14: Average power consumption of the Xeon processor on Ubuntu, CentOS ($P_{idle} = 92\,\text{W}$) and CentOS with performance profile enabled ($P_{idle} = 125\,\text{W}$).



In particular, we compare the duration and the power consumption of each profile while changing the utilization ratio of a core and increasing the number of allocated cores. With default settings, the choice between Ubuntu and CentOS does not impact the power consumption and none of them pulls out of the game in terms of execution duration. However, more interesting reports are delivered when the latency-performance profile is enabled. Indeed, when one process stresses a full core, the power difference between the default settings and this profile can be greater than 20 W. This difference is due to the idle power that represents a non-negligible part of the power drawn by this profile. Actually, the latency-performance profile turns all cores of the processor in the `C0` state, which means that the cores are always turned on for minimizing the latency to wake up.

Moreover, one can see that the activation of the performance profile does not decrease the execution duration of the benchmark. Hence, we can clearly target Ubuntu or CentOS with default settings to get the best compromise between performance and power consumption.

These experiments show that the optimizations made by the OS can be a source of power loss if used inappropriately. PowerAPI can thus be used as a power profiler to study the efficiency of the optimizations made available at the hardware or OS levels. PowerAPI being modular, the formula actor can be easily extended to handle such hybrid power models at runtime.

7. Conclusion

Since the publication of the first analytical power models [53], the research community has been intensively investigating the design of CPU power models by considering different architectures, power-aware features, workloads, and modeling techniques. Nevertheless, the state-of-the-art in this area demonstrates that the designed CPU power

models are mostly handcrafted and based on assumptions that prevent their reuse in other execution contexts and their deployment at scale.

In this article, we therefore adopt a different approach to this problem by proposing a middleware toolkit that helps researchers to automatically learn the power model of a CPU without requiring a deep expertise of the considered CPU architecture. Our solution implements the aforementioned learning approach and exploits freely available benchmark suites to discover the hardware performance counters that accurately reflect the power consumption of the CPU under a wide diversity of input workloads. The selected hardware performance counters are then exploited by a combination of regression analysis techniques to identify the most accurate power model that fits the targeted CPU. In particular, we illustrate that our solution supports software-defined power meters that can monitor the power consumption of any application with less than 1.5% of error, on average. But more importantly, the architecture of PowerAPI allows experts *i)* to change the learning technique upon needs and *ii)* to compare the accuracy of their results by adopting an open testbed.

To encourage this approach, our implementation of the toolkit, PowerAPI, is published as open source software[39] under AGPLv3 license to foster the wide adoption and deployment of CPU power models. Beyond these deployment issues, we also aim at extending this open testbed to consider other power-consuming components, such as GPU [54] and disk, in order to incrementally learn their power model and thus provide wider cartography of the power consumption of a software system. In the future, we believe that PowerAPI can be a cornerstone to new energy-aware scheduling [4, 15–17, 19], to energy-proportional computing [20–23], to new kind of optimizations [24], and to a better understanding of the power consumption drawn by software [25–27].

Beyond the open source availability of PowerAPI, the hardware and software settings, installation steps, and the experimental protocols are available online for the reader interested in reproducing the results reported in this article.[40]

Acknowledgments

This work has received funding from the EU's Horizon 2020 research and innovation programme under grant agreements 780681 (LEGaTO), as well as CPER Data project. CPER Data is co-financed by European Union with the financial support of European Regional Development Fund (ERDF), French State and the French Region of Hauts-de-France.

[39] Available from: `http://powerapi.org`.

[40] Available from: `http://bit.ly/The-Next-700-CPU-Power-Models`

Bibliography

[1] A.-C. Orgerie, M. Dias de Assuncão, L. Lefèvre, A Survey on Techniques for Improving the Energy Efficiency of Large-scale Distributed Systems, ACM Comput. Surv.

[2] G. Tang, W. Jiang, Z. Xu, F. Liu, K. Wu, Zero-Cost, Fine-Grained Power Monitoring of Datacenters Using Non-Intrusive Power Disaggregation, in: Proceedings of the 16th Annual Middleware Conference, 2015.

[3] D. Economou, S. Rivoire, C. Kozyrakis, Full-system power analysis and modeling for server environments, in: In Workshop on Modeling Benchmarking and Simulation, 2006.

[4] F. Bellosa, The Benefits of Event: Driven Energy Accounting in Power-sensitive Systems, in: Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System, 2000.

[5] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, A. Sobe, Process-level Power Estimation in VM-based Systems, in: Proceedings of the 10th European Conference on Computer Systems, 2015.

[6] A. Kansal, F. Zhao, J. Liu, N. Kothari, A. A. Bhattacharya, Virtual Machine Power Metering and Provisioning, in: Proceedings of the 1st ACM Symposium on Cloud Computing, 2010.

[7] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, R. K. Gupta, Evaluating the Effectiveness of Model-based Power Characterization, in: Proceedings of the USENIX Annual Technical Conference, 2011.

[8] D. Versick, I. Wassmann, D. Tavangarian, Power Consumption Estimation of CPU and Peripheral Components in Virtual Machines, SIGAPP Appl. Comput. Rev.

[9] W. L. Bircher, M. Valluri, J. Law, L. K. John, Runtime identification of microprocessor energy saving opportunities, in: Proceedings of the International Symposium on Low Power Electronics and Design, 2005.

[10] C. Isci, M. Martonosi, Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data, in: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003.

[11] Y. Zhai, X. Zhang, S. Eranian, L. Tang, J. Mars, HaPPy: Hyperthread-aware Power Profiling Dynamically, in: Proceedings of the USENIX Annual Technical Conference, 2014.

[12] T. Li, L. K. John, Run-time Modeling and Estimation of Operating System Power Consumption, SIGMETRICS Perform. Eval. Rev.

[13] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, E. Ayguade, Decomposable and Responsive Power Models for Multicore Processors Using Performance Counters, in: Proceedings of the 24th ACM International Conference on Supercomputing, 2010.

[14] P. J. Landin, The next 700 programming languages, Commun. ACM.

[15] M. Bambagini, J. Lelli, G. Buttazzo, G. Lipari, On the energy-aware partitioning of real-time tasks on homogeneous multiprocessor systems, in: Energy Aware Computing Systems and Applications, 2013.

[16] N. Moghaddami Khalilzad, J. Lelli, G. Lipari, T. Nolte, Towards Energy-aware Multiprocessor Hierarchical Scheduling of Real-time Systems, in: 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2013.

[17] M. Rasmussen, sched: Energy cost model for energy-aware scheduling (2015).

[18] A. Havet, V. Schiavoni, P. Felber, M. Colmant, R. Rouvoy, C. Fetzer, GenPack: A Generational Scheduler for Cloud Data Centers, in: Proceedings of the 5th IEEE International Conference on Cloud Engineering (IC2E), 2017.

[19] M. Kurpicz, A. Sobe, P. Felber, Using Power Measurements As a Basis for Workload Placement in Heterogeneous Multicloud Environments, in: Proceedings of the 2nd International Workshop on CrossCloud Systems, 2014.

[20] L. Barroso, U. Holzle, The Case for Energy-Proportional Computing, Computer.

[21] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, R. H. Katz, NapSAC: Design and Implementation of a Power-proportional Web Cluster, in: Proceedings of the First ACM SIGCOMM Workshop on Green Networking, 2010.
[22] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, T. F. Wenisch, Power Management of Online Data-intensive Services, SIGARCH Comput. Archit. News.
[23] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, E. Bugnion, Energy Proportionality and Workload Consolidation for Latency-critical Applications, in: Proceedings of the Sixth ACM Symposium on Cloud Computing, 2015.
[24] E. Schulte, J. Dorn, S. Harding, S. Forrest, W. Weimer, Post-compiler Software Optimization for Reducing Energy, in: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, 2014.
[25] A. Noureddine, R. Rouvoy, L. Seinturier, Unit Testing of Energy Consumption of Software Libraries, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, 2014.
[26] A. Noureddine, R. Rouvoy, L. Seinturier, Monitoring energy hotspots in software - Energy profiling of software code, Autom. Softw. Eng.
[27] C. Sterling, Energy Consumption tool in Visual Studio 2013 (2013).
[28] J. Arjona Aroca, A. Chatzipapas, A. Fernández Anta, V. Mancuso, A Measurement-based Analysis of the Energy Consumption of Data Center Servers, in: Proceedings of the 5th International Conference on Future Energy Systems, 2014.
[29] G. Contreras, M. Martonosi, Power Prediction for Intel XScale® Processors Using Performance Monitoring Unit Events, in: Proceedings of the International Symposium on Low Power Electronics and Design, 2005.
[30] M. F. Dolz, J. Kunkel, K. Chasapis, S. Catalán, An analytical methodology to derive power models based on hardware and software metrics, Computer Science - Research and Development.
[31] S. Rivoire, P. Ranganathan, C. Kozyrakis, A Comparison of High-level Full-system Power Models, in: Proceedings of the Conference on Power Aware Computing and Systems, 2008.
[32] H. Yang, Q. Zhao, Z. Luan, D. Qian, iMeter: An integrated VM power model based on performance profiling, Future Generation Computer Systems.
[33] R. Zamani, A. Afsahi, A Study of Hardware Performance Monitoring Counter Selection in Power Modeling of Computing Systems, in: Proceedings of the 2012 International Green Computing Conference, 2012.
[34] W. Bircher, L. John, Complete System Power Estimation: A Trickle-Down Approach Based on Performance Events, in: Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software, ISPASS '07, 2007.
[35] Intel, Intel 64 and IA-32 Architectures Software Developer's Manual (2015).
[36] X. Fan, W.-D. Weber, L. A. Barroso, Power Provisioning for a Warehouse-sized Computer, in: Proceedings of the 34th Annual International Symposium on Computer Architecture, 2007.
[37] T. Heath, B. Diniz, E. V. Carrera, W. Meira, Jr., R. Bianchini, Energy Conservation in Heterogeneous Server Clusters, in: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2005.
[38] V. Vapnik, S. E. Golowich, A. Smola, Support Vector Method for Function Approximation, Regression Estimation, and Signal Processing, in: Advances in Neural Information Processing Systems, 1997.
[39] T. Do, S. Rawshdeh, W. Shi, pTop: A process-level power profiling tool, in: Proceedings of the 2nd Workshop on Power Aware Computing and Systems, 2009.
[40] J. Flinn, M. Satyanarayanan, PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications, in: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications, 1999.
[41] Intel, PowerTOP (2015).
URL `https://01.org/powertop`
[42] S. Wang, H. Chen, W. Shi, SPAN: A software power analyzer for multicore computer systems, Sustainable Computing: Informatics and Systems.
[43] S. Polfliet, F. Ryckbosch, L. Eeckhout, Automated Full-System Power Characterization, Micro, IEEE.
[44] P. Nordwall, 50 million messages per second - on a single machine (2012).
URL `http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single`
[45] C. Bienia, K. Li, PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors, in: Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation, 2009.
[46] M. Y. Lim, A. Porterfield, R. Fowler, SoftPower: Fine-grain Power Estimations Using Performance Counters, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, 2010.
[47] P. J. Rousseeuw, A. M. Leroy, Robust Regression and Outlier Detection, 1987.
[48] S. Rivoire, M. A. Shah, P. Ranganathan, C. Kozyrakis, JouleSort: a balanced energy-efficiency benchmark, in: Proceedings of the ACM SIGMOD international conference on Management of data, 2007.
[49] X. Feng, R. Ge, K. Cameron, Power and Energy Profiling of Scientific Applications on Distributed Systems, in: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, 2005.
[50] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, K. Cameron, PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications, IEEE Transactions on Parallel and Distributed Systems.
[51] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al., The NAS parallel benchmarks, International Journal of High Performance Computing Applications.
[52] SPEC, SPECjbb2013 Design Document (2013).
[53] A. Kansal, F. Zhao, Fine-grained Energy Profiling for Power-aware Application Design, SIGMETRICS Perform. Eval. Rev.
[54] W. Jia, K. Shaw, M. Martonosi, Stargazer: Automated regression-based GPU design space exploration, in: Performance Analysis of Systems and Software, ISPASS '12, 2012.