WattsKit: Software-Defined Power Monitoring of Distributed Systems

Maxime Colmant, Pascal Felber, Romain Rouvoy, Lionel Seinturier

WATTSKIT: Software-Defined Power Monitoring of Distributed Systems

Maxime Colmant
ADEME / University of Lille / Inria
maxime.colmant@inria.fr

Pascal Felber
University of Neuchâtel
pascal.felber@unine.ch

Romain Rouvoy, Lionel Seinturier
University of Lille / Inria / IUF
first.last@inria.fr

***Abstract*—The design and the deployment of energy-efficient distributed systems is a challenging task, which requires software engineers to consider all the layers of a system, from hardware to software. In particular, monitoring and analyzing the power consumption of a distributed system spanning several—potentially heterogeneous—nodes becomes particularly tedious when aiming at a finer granularity than observing the power consumption of hosting nodes. While the state-of-the-art in software-defined power meters fails to deliver adaptive solutions to offer such service-level perspective and to cope with the diversity of hardware CPU architectures, this paper proposes to automatically learn the power models of the nodes supporting a distributed system, and then to use these inferred power models to better understand how the power consumption of the system's processes is distributed across nodes at runtime.**

Our solution, named WATTSKIT, offers a modular toolkit to build software-defined power meters "à la carte", thus dealing with the diversity of user and hardware requirements. Beyond the demonstrated capability of covering a wide diversity of CPU architectures with high accuracy, we illustrate the benefits of adopting software-defined power meters to analyze the power consumption of complex layered and distributed systems. In particular, we illustrate the capability of our approach to monitor the power consumption of a system composed of Docker SWARM, WEAVE, ELASTICSEARCH, and Apache ZOOKEEPER. Thanks to WATTSKIT, developers and administrators are now able to identify potential power leaks in their software infrastructure.

I. INTRODUCTION

The design and the deployment of energy-efficient distributed systems is a challenging task, which requires software engineers to consider all the layers of a system, from hardware to software. While the state-of-the-art in green computing proposes solutions to increase energy efficiency at all levels, from applications over run-time to hardware [20], it remains difficult to evaluate the power consumption of a distributed software system *i)* spanning several—potentially heterogeneous—nodes and *ii)* composing several distributed algorithms and/or protocols.

In this domain, *Power Distribution Units* (PDUs) are often shared amongst nodes to deliver aggregated power consumption reports, in the range of hours or minutes. However, in order to improve the energy efficiency of distributed systems, one needs to offer novel power monitoring solutions that go beyond the node's granularity—*i.e.*, at a finer granularity by considering software processes—therefore surpassing the actual capabilities of PDUs [24]. To build such fine-grained and distributed power monitoring solutions, or *software-defined power meters*, the CPU—considered as the major power consumer within a node [9, 18]—requires to be accurately modeled in order to capture the activity of a distributed software service. Designing power models that can accurately cover the power-aware features of a CPU (*e.g.*, multi-threading, frequency scaling) is a complex task. We therefore build on the expertise we developed in [5] to automatically learn the power model of the nodes supporting the execution of a distributed software system. The resulting power models are then used to build a software-defined power meter, named WATTSKIT, which can report on the power consumption of complex distributed systems by aggregating and processing in real-time the power measurements collected across multiple hosting nodes. Unlike the *in-depth* approach we reported in [5] with BITWATTS, this paper rather investigates *in-breadth* power monitoring of a software system that spans several physical hosts. In particular, WATTSKIT proposes a complementary solution to BITWATTS that can therefore aggregate power measurements both at scale and with a fine granularity. As a matter of motivation and validation, we illustrate the benefits of WATTSKIT on the monitoring and the analysis of the power consumption of a distributed system stack composed of Docker SWARM, WEAVE, ELASTICSEARCH, and Apache ZOOKEEPER, deployed in a cluster of 6 nodes grouping 3 generations of CPUs. We have implemented WATTSKIT in Scala, as an extension of the POWERAPI actor toolkit, and made it freely available under the AGPL license to encourage the reproducibility of our results and the effective deployment of our solution.[1]

The remainder of this paper is organized as follows. Section II introduces the case study covered by this paper. Section III details our approach to automatically learn the CPU power model of a node. Section IV reports on the implementation and the accuracy of software-defined power meters based on WATTSKIT. Section V revisits the case study with WATTSKIT by offering new perspectives on the power consumption of distributed systems. Section VI discusses the state of the art of software-defined power meters and current limitations of existing approaches. Finally, Section VII concludes this paper and sketches some perspectives for further research.

[1] http://powerapi.org

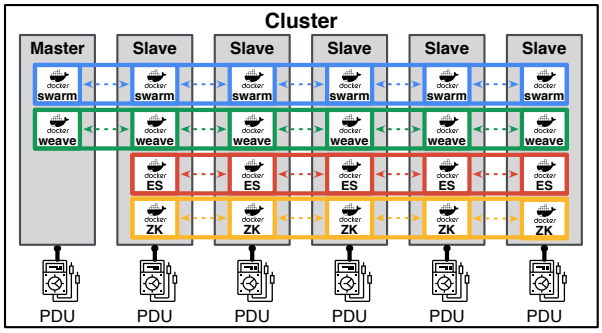

Fig. 1. Overview of the distributed search engine based on ELASTICSEARCH.

II. CASE STUDY

Distributed software systems are generally composed of several protocols and algorithms that are used to implement the various services that are required across the system. In this paper, we study a distributed system stack composed of several distributed services, which are widely deployed nowadays. In particular, we consider an instance of a distributed search engine based on ELASTICSEARCH[2], which is—at the time of writing this paper—the most popular enterprise full-text search engine with an HTTP web interface and schema-free JSON documents.[3]

More specifically, ELASTICSEARCH builds on Apache ZOOKEEPER[4] to implement the discovery and coordination services, which are required to operate in a distributed configuration. The deployment of instances of ELASTICSEARCH onto nodes is achieved by Docker SWARM, which can be considered as the *de facto* standard for building a cluster of Docker hosts.[5] Both ELASTICSEARCH and ZOOKEEPER are therefore packaged as Docker containers and we use WEAVE to manage their network configuration.[6]

Figure 1 summarizes the deployment of this distributed software system on 6 hosts composed of 1 master node (Intel Xeon W3520) and 5 slave nodes (2 Intel Xeon W3520, 1 Intel Core2 Q6600, and 2 Intel Core2 E8400).

Given the distributed nature of each of these services (ELASTICSEARCH, ZOOKEEPER, SWARM, WEAVE) and their entanglement due to respective dependencies, monitoring and analyzing the power consumption of individual services is a particularly tedious task. For example, Figure 2 illustrates the measurements reported by a PDU physically connected to each node of the cluster (cf. Figure 1). In this experiment, we deploy and sequentially stress each of these services by running the ZOOKEEPER benchmark[7] and *Yahoo! Cloud Serving Benchmark* (YCSB)[8] [7] while logging the power

[2]https://www.elastic.co
[3]http://db-engines.com/en/system/Elasticsearch
[4]https://zookeeper.apache.org
[5]https://docs.docker.com/swarm/overview
[6]https://www.weave.works
[7]https://github.com/brownsys/zookeeper-benchmark
[8]https://research.yahoo.com/news/yahoo-cloud-serving-benchmark

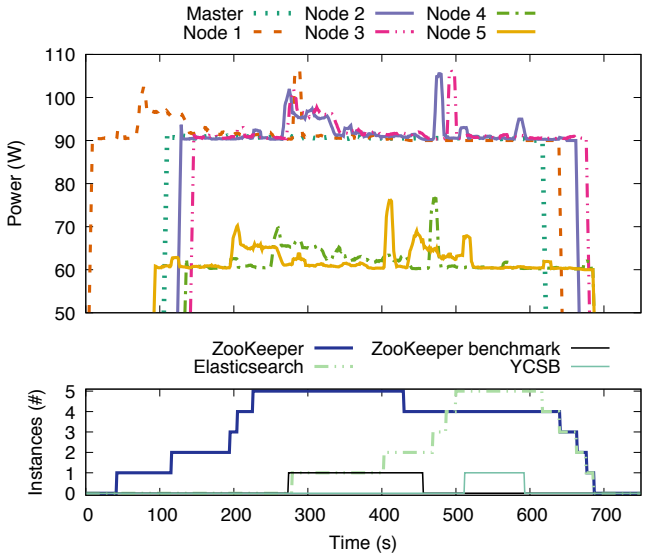

Fig. 2. Power consumption of the distributed search engine based on ELASTICSEARCH.

consumption per node reported by the associated PDU. In particular, we run the *update heavy workload* (Workload A) of YCSB, which has a mix of 50/50 reads and writes. An application example is a session store recording recent actions. We complete the scenario by killing sequentially each node of the cluster to observe the impact of nodes' leaves on the distributed system's behavior.

If one can observe some variations in the power consumption of individual nodes along the execution of the scenario, it still remains difficult to analyze which software services can be accounted for such node-scale variations. Furthermore, the heterogeneity of nodes (Intel Xeon W3520, Intel Core2 Q6600 and E8400), which is the rule in modern production systems, complicates the power analysis due to the diversity in idle powers and CPU power features (hyper-threading, turbo-boost, etc.).

One therefore needs to manually tag the physical nodes to software services and to ideally find the relevant scenarios that isolate the execution of services for obtaining a better insight on their individual power consumption, or for identifying potential energy leaks or optimizing the whole system's configuration.

In this paper, we therefore introduce WATTSKIT as a solution to this key limitation of coarse-grained power measurements and we propose, in particular, to introduce a modular approach to monitor—in real-time—the power consumption of individual software services involved in a distributed system. In the following sections, we first define and assess a service-level power model before revisiting the above case study with our solution.

III. ENABLING SERVICE-LEVEL POWER MONITORING

To deliver service-level power measurements, our approach consists in tracking the power consumption—per node—of the system processes associated to the services of a given

distributed system before aggregating these power measurements at the scale of the cluster. Achieving such process-level power measurements therefore requires a software-defined power meter, as physical power meters are limited to the boundaries of nodes and hardware components. For example, Intel's *Running Average Power Limit* (RAPL) fails to support this process-level granularity [5].[9]

Given the diversity of nodes and services, we describe in this section our technique to automatically learn a service-level power model that can be used to monitor a distributed service deployed across several—potentially heterogeneous—physical nodes. Unlike the state-of-the-art in this domain [27], our power model is service-agnostic, which means that it can be used to track a wide diversity of distributed services. Once defined, this power model can therefore be used in production by WATTSKIT to monitor the power consumption of the individual services that compose the distributed system in real-time.

To build this service-level power model, we adopt a bottom-up approach, therefore estimating the power consumption of the instances of the services running on the hosting nodes, before aggregating them into a service-level power model. Regarding network-intensive workloads, we have previously demonstrated in [18] that the power consumption of network-intensive systems were dominated by the activity of the CPU spent on I/O operations. By carefully modeling such I/O operations, we are therefore able to deliver accurate estimation of both memory-intensive and network-intensive workloads [5]. To better understand our approach, we therefore start by describing the key power-aware features of modern CPUs, before presenting our learning approach to build a service-level power model.

A. Power-Aware Features of Modern Processors

To control their energy consumption, modern CPUs heavily rely on frequency scaling and power saving modes to adjust their performance according to computation requirements. In particular, the multi-core processors designed by Intel integrate the following power-aware features:

Hyper-Threading (HT) is used on latest processor generations (*e.g.*, Core2, Xeon) to separate each physical core into two logical threads. The technology is based on the *simultaneous multi-threading* (SMT) principle, which allows the processor to seamlessly support *thread-level parallelism* (TLP) in hardware and share more effectively the available resources.

SpeedStep (SS) is Intel's implementation of *dynamic voltage/frequency scaling* (DVFS), which allows a processor to adjust its clock speed and run at different frequencies or voltages upon needs. The OS can increase the frequency to quickly execute operations or reduce it to minimize dissipated power when the processor is under-utilized.

C-states (CS) (idle states) 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 units on the processor, and reduce the power consumed. The more units are shut down, the higher are the power savings.

[9]https://01.org/rapl-power-meter

TABLE I
SPECIFICATION OF INTEL PROCESSORS USED IN THE EXPERIMENT.

Vendor	Intel	Intel	Intel
Processor	**Xeon**	**Core2 Quad**	**Core2 Duo**
Model	W3520	Q6600	E8400
Design	8 threads	4 cores	4 cores
Frequency	2.66 GHz	2.4 GHz	3.0 GHz
TDP	130 W	105 W	65 W
HT	✓	✗	✗
SS	✓	✓	✓
CS	✓	✓	✓
TB	✓	✗	✗

TurboBoost (TB) dynamically increases the CPU frequency beyond the maximum bound, which can be greater than the *thermal design power* (TDP), for a short period of time. It therefore allows the processor cores to execute more instructions by running faster for a short period of time, hence saving energy by triggering C-states.

For example, Table I reports on the features made available for each of the nodes we used in the experimental setup of the case study we describe in Section II. These 3 configurations differ by the number of cores and threads available as well as the CPU features (Hyper-Threading, TurboBoost) that can be exploited by the operating system.

The definition of a node-level power model therefore requires to consider the impact of these features, when available, on the power consumption of the node and the subsequent processes executed by this node.

B. Learning of a Service-Level Power Model

To define our service-level power model, we first model the power consumption of a single node n as the sum of its idle consumption $P^{idle}_{node}(n)$ and the consumptions of individual processes $P^{dyn}_{node}(pid)$:

$$P_{node}(n) = P^{idle}_{node}(n) + \sum_{pid \in P(n)} P^{dyn}_{node}(pid)$$

Our objective is to build a lightweight model that imposes a very limited overhead to our monitoring solution.

While the state-of-the-art CPU power models demonstrate that achieving accurate power estimation is possible, they are barely generalizable to processors or applications which were not part of the original study. Therefore, we rather propose a tooled approach capable of learning the specifics of a processor and building the fitting CPU power model. Hence, rather than proposing yet another hand-crafted power model, our approach intends to cover the continuous evolution of CPU architectures and therefore aims at delivering a solution that will be able to deal with current and future generation of CPU architectures.

As reported by [13], the CPU load does not accurately reflect the diversity of CPU activities. In particular, to faithfully capture the power model of a CPU, the types of task executed by the CPU have to be clearly identified. We therefore decided to base our power models on *Hardware Performance Counters* (HPCs) to collect raw, yet accurate, metrics reflecting the types of operation that are truly executed by the CPU. Nevertheless,

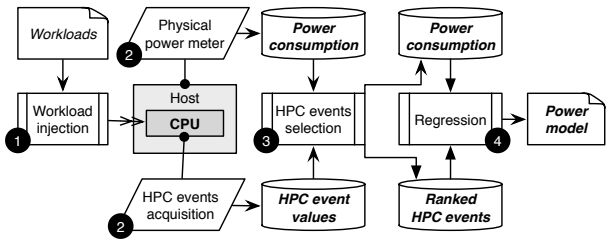

Fig. 3. Architecture-agnostic approach for learning multi-core CPU power models.

the number and the nature of HPC events provided by the CPU strongly vary according to the processor type.

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 two types of HPC 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 one 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 event. When multiplexing is triggered, the events cannot be accurately monitored anymore and an approximation is returned instead.

Learning the CPU power model of a multi-core processor requires the use of workloads that carefully stress the various features it supports. Thereby, it is important to isolate the noise induced by other hardware components to properly capture the power consumption of the CPU under study.

The learning phase we depict in Figure 3 analyses the power consumption and triggered HPC events of the target CPU, in order to identify the key events that impact the power consumption. The combination of these events is then used to learn automatically the CPU power model.

Our goal here is to automatically classify the HPC events in order to identify those which are best characterizing the CPU activity and are correlated with its power consumption. Each step of our architecture-agnostic approach for learning CPU power models is depicted in Figure 3 and described below.

a) Input workload injection.: For exploring the activity of a CPU, we consider a set of representative applications covering its features. In particular, to promote the reproducibility of our results, we favor freely available and widely used benchmark suites, such as PARSEC [3]. However, this choice does not prevent us from including additional benchmark suites or any sample workloads. All workloads are then launched several times in isolation for reducing the noise that can be experienced during the learning phase.

b) Acquisition of raw HPC counters.: Unfortunately, the CPU cannot monitor hundreds of HPC events simultaneously [10]. Thus, we have to split the list of available events into subsets of events to avoid multiplexing that might cause inaccuracies. Consequently, we dynamically infer at runtime the number of events that we can monitor together for getting accurate raw measurements. For a given CPU, the number of workload executions w to be considered is therefore defined as:

$$w = \sum_{p \in PMU} \left\lceil \frac{|E_p|}{|C_p|} \right\rceil \times |W| \times i$$

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 required to guarantee an acceptable duration for the learning phase. Our approach proposes an automated way to focus on the most relevant events. In the first step, each workload is only executed for a few seconds while collecting values from HPC events and from a power meter. We then select relevant HPC events by applying the Pearson correlation coefficient [6, 27].

c) Selection of relevant HPC events.: As next step, we eliminate the HPC events that have a median correlation coefficient ($\tilde{r}$) below a given threshold. In particular, we consider that any coefficient below 0.5 clearly indicates a lack of correlation between the considered event (e) and the associated power consumption (p). With this step, we 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 default runtime. 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 4.

The distribution of Pearson coefficients for the 30 best events 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 simulate all selected HPC events (*e.g.*, `x264`, `vips`) from the ones whose power consumptions match only specific events (*e.g.*, `freqmine`, `fluidanimate`). Deriving a CPU power model that is capable of covering all kinds of workloads accurately is consequently a challenging task.

d) Power model inference: We finally apply a regression analysis to derive the CPU power model from the previously selected HPC events. In particular, we use the robust ridge regression [15, 23], which belongs to the family of multivariate linear regressions. Our approach being fully configurable, the aforementioned linear regression can be thus chosen upon needs. Our choice was guided by the need to easily eliminate outliers. It has been validated further by the experiments and results presented in Section V.

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

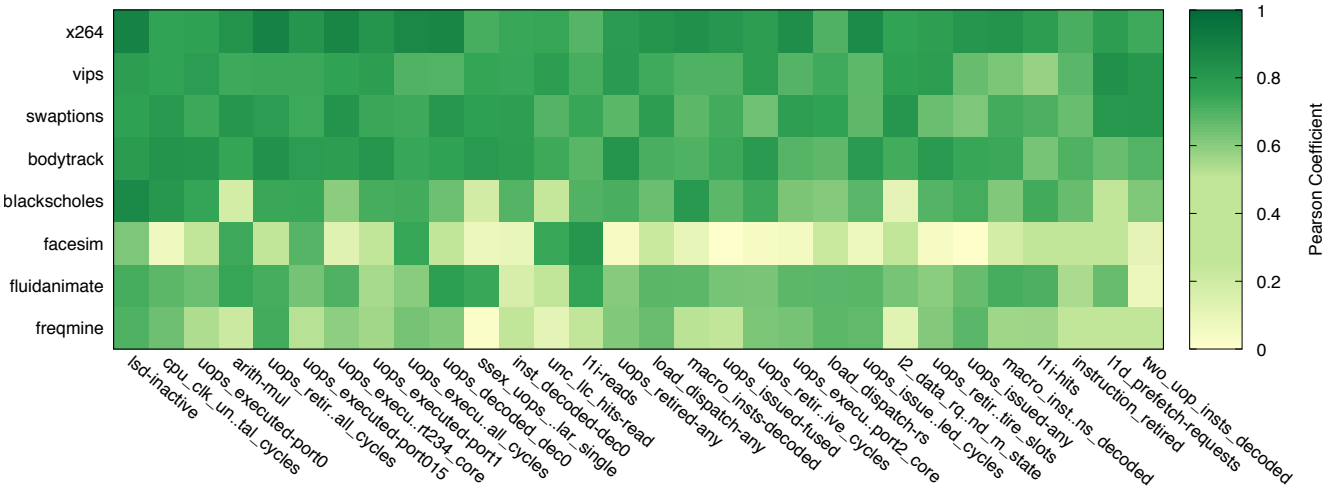

Fig. 4. Pearson coefficients of the Top-30 correlated events for the PARSEC benchmarks on an Intel Xeon W3520.

consider a subset R_n of n benchmarks , 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 in parallel. For each training set R_n, 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, not included in the training set.

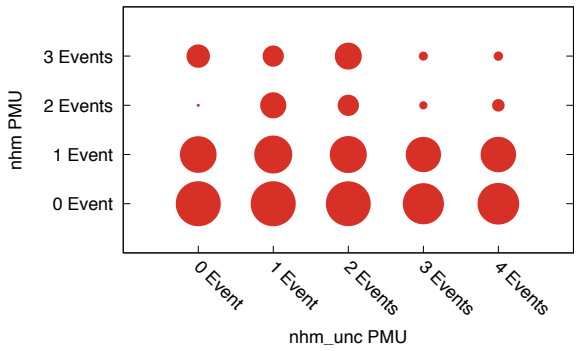

Fig. 5. Average error per combination of events for R_3 on an Intel Xeon W3520 processor.

As an illustration, Figure 5 reports on 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 resulting power model. A larger circle means a larger error. One can clearly see that a CPU power model that combines a high number HPC events may exhibit a larger error than the one that uses a lower number of events.

As an example, in the Intel Xeon W3520 processor, the CPU power model composed of 2 events taken from the PMU `nhm` emerges from this analysis and reports, on average, an error of 1.35%, 1.60 W respectively.

Once the node-level power models are inferred and deployed, we can define the service-level power model as:

$$P_{service}(s) = \sum_{n \in N(s)} \sum_{p \in P_n(s)} P_n^{dyn}(p)$$

where $P_{service}(s)$ aggregates in real-time all the power measurements for each instance p of the service s running on the set of hosting nodes n in the cluster.

The following section describes the integration and the assessment of this service-level power model within WattsKit.

IV. Building Software-Defined Power Meters

The learning phase is an offline process that is realized when a new node is deployed into the cluster. The learning phase is therefore the only phase where a PDU is required to be connected to the node since the resulting power models are intended to be integrated into the instances of software-defined power meters, thus offering a finer monitoring granularity than physical PDUs. In this section, we describe how such power models are integrated within the software-defined power meters built with WattsKit.

A. Overview of WattsKit

WattsKit is built on top of Scala and the actor programming model, thanks to the Akka library. Scala is a language that combines object-oriented and functional programming paradigms with a strong static type system [19]. Scala code runs on top of the JVM and is fully interoperable with Java code. Akka is an open-source toolkit for building scalable, and distributed applications that pushes forward the actor

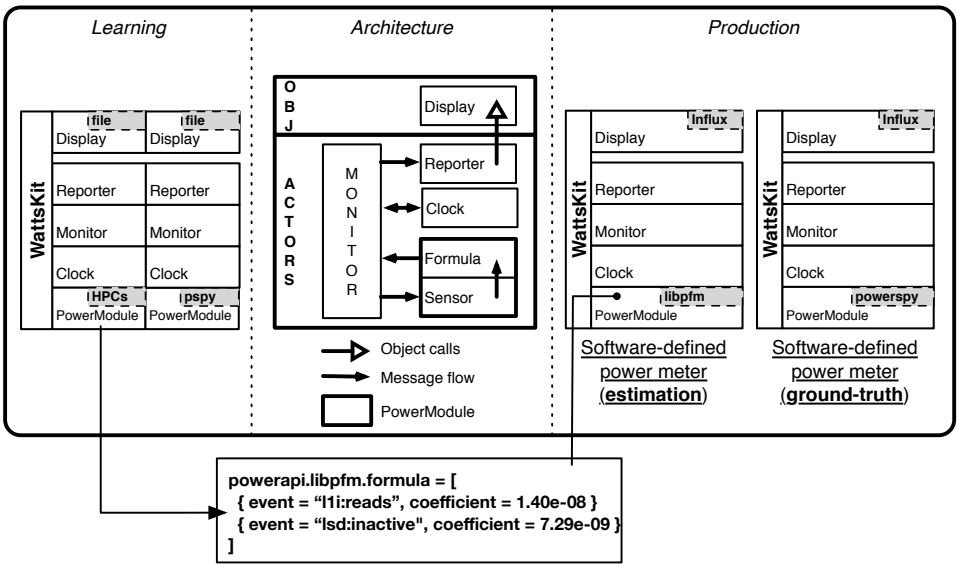

Fig. 6. Software-defined power meters built with WATTSKIT for this paper. The middle part describes our middleware toolkit design. The left side shows the component assembly used in Section III, while the right side presents the assemblies we use here and in Section V.

programming model as the best programming model for concurrency.[10]

The software components of WATTSKIT are implemented as Akka actors, which can process millions of messages per second [17], a key property for supporting real-time power estimation. WATTSKIT is therefore fully asynchronous and scales both on several dimensions—*i.e.*, the number of input sources, the requested monitoring frequencies...

More especially, the WATTSKIT middleware framework identifies 5 types of actor components:

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

Monitor actors reflect 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 actor is also responsible to aggregate the power estimation by applying a function (*e.g.*, SUM, MEAN, MAX) defined when needed.

Sensor actors connect the software-defined power meters to the underlying system in order to collect raw measurements of system activities. Raw measurements can be coarse-grained power consumption reported by third-party power meters and embedded probes (*e.g.*, PowerSpy, RAPL), or 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.

Formula actors use the raw measurements received from the sensor to compute a power estimation. A formula implements a specific power model [13, 25] to convert raw measurements into power estimation. The granularity of the power consumptions reported by the formula (machine, core, process) depends on the granularity of the measurements forwarded by the sensors.

Reporter actors finally gives 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 information (*e.g.*, the timestamp, the monitoring id or the devices) into a suitable format. The built-in report is then provided, by example, via a web interface or a virtual file system (*e.g.*, based on FUSE), or can be uploaded into a database (*e.g.*, INFLUXDB).

As actors have lightweight CPU and/or memory footprints, a Monitor, a Sensor, a Formula, and a Reporter actors are created per monitoring request and target. Each Sensor and Formula actors being tightly coupled, we grouped them as a PowerModule that represents the link between the input data and the power model. All actors are centralized on a common event bus where they can publish messages and subscribe to topics for actively waiting events.

The overall architecture of WATTSKIT is described in Figure 6. Several PowerModule components can be assembled together for grouping power estimation from multiple sources.

One can see that WATTSKIT is fully modular and can be used to assemble power meters upon needs to fulfill all monitoring requirements. We can also note that WATTSKIT is a non-invasive solution and does not require costly investments or specific kernel updates.

Regarding the monitoring frequency, WATTSKIT is mostly limited by the frequency of the hardware and software sensors used to collect runtime metrics. In particular, WATTSKIT can report on the power consumption of software processes up to 40 Hz when connected to the PowerSpy, and up to 10 Hz when using the `libpfm` library. However, increasing the monitoring frequency affects the stability of the power

[10]http://akka.io

```
object SDPowerMeter extends App {
  val sdpm = PM.loadModule(LibpfmCoreProcessModule())
  val influxDisplay = new InfluxDisplay(host, ...)

  // Can be configured to monitor all services.
  val zookeeperPower = sdpm.monitor("zookeeper")
                           .every(250.milliseconds)
                           .to(influxDisplay)

  zookeeperPower.waitFor(1.hour)

  zookeeperPower.cancel()
  sdpm.shutdown()
}
```

Snippet 1. Example of software-defined power meter built with our API to monitor the ZOOKEEPER service power consumption.

consumption observed, thus not helping to properly identify the power consumption of the services [5].

Two instances of software-defined power meters are also depicted in this figure. In the left side, one can find an instance of WATTSKIT especially configured to learn the CPU power models. This instance is composed of 2 PowerModule components, one for retrieving raw accurate CPU metrics via `libpfm`, and another, for retrieving the power measurements from the bluetooth power meter. The data are then forwarded to several files to be later processed by our learning approaches. The resulting power model is then written inside a configuration file that can be used later by a new instance of WATTSKIT to estimate the power consumption. In the other side, another instance of WATTSKIT is configured to use the aforementioned power model for producing service power estimation.

WATTSKIT can also be used as a connector to external probes for retrieving power measurements (*e.g.*, PowerSpy, or G5K OmegaWatt), as shown in Figure 6.

WATTSKIT adopts the Docker[11] technology to package our software-defined power meters inside lightweight images that contains everything needed to deploy them. These images can be directly downloaded[12] and used without any dependency to install (except Docker itself).

Different ways are proposed for creating software-defined power meters. Firstly, end users can assemble and build their own power meters by using our customizable and documented API. Secondly, a CLI has been made available and can be used for testing or doing basic power monitoring that use default components.

An example of software-defined power meter to monitor the power consumption of the ZOOKEEPER service is described in Snippet 1.

B. Assessment of the Node-Level Power Model

To assess our node-level power model, we use WATTSKIT and we compare the power estimation resulting from the learned models with raw power measurements from a physical power meter, PowerSpy, which acts as a ground truth (cf. Figure 6).[13] We run the well-known PARSEC v2.1 benchmark suite [3], which includes a wide diversity of workloads. 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 modern architectures.

[11]https://www.docker.com

[12]Freely available from: https://hub.docker.com/u/spirals

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

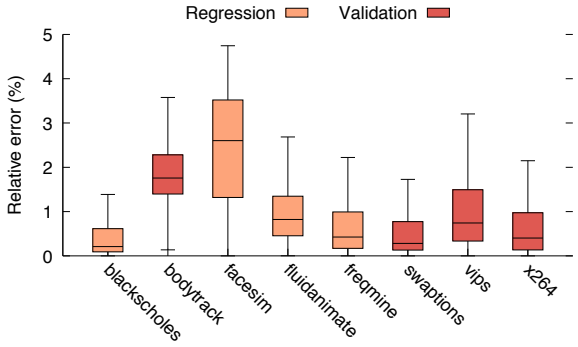

Fig. 7. Relative error distribution of the PARSEC benchmarks on the Intel Xeon W3520 processor ($P_{idle} = 92$ W).

In particular, we report the power consumption of all the benchmarks available on one of the Intel Xeon W3520 nodes used in our experimental setup. The resulting CPU power model computed with the training subset of benchmarks, R_3, comprises 2 HPC events from the PMU nhm ($e1$ = lli:reads, $e2$ = lsd:inactive):

$$P_{idle} = 92 \text{ W} \; ; \; P_{CPU} = \frac{1.40 \cdot e1}{10^8} + \frac{7.29 \cdot e2}{10^9}$$

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.

Our approach isolates the idle power consumption of the processor whose relationship to TDP is defined in [22] as $P \simeq P_{idle} + 0.7 \times \text{TDP}$. Figure 7 reports on an average relative error of 1.35% (1.60 W), which improves the existing CPU power models on such configuration [5].

V. MONITORING THE POWER CONSUMPTION OF DISTRIBUTED SERVICES

In this section, we revisit the case study we introduced in Section II with WATTSKIT to offer a new perspective on the power consumption of the distributed services deployed as part of this system, thus overcoming the limitations we previously observed.

A. Deploying WATTSKIT in a Distributed Environment

As part of this validation, we deploy WATTSKIT as a Docker container, which runs along the other services we previously deployed as containers (cf. Figure 8). This configuration only differs from Figure 1 by unplugging the physical PDUs, which

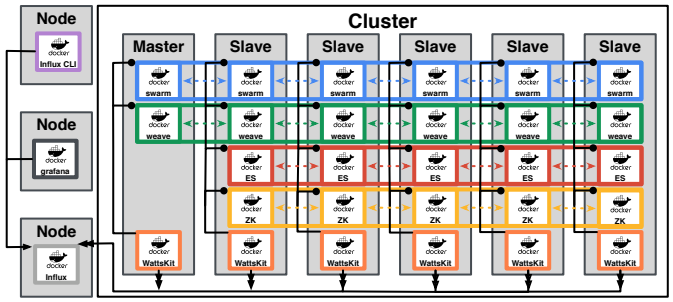

Fig. 8. Overview of the experimental deployment of WATTSKIT.

are replaced by the WATTSKIT containers running on each node and an instance of the INFLUXDB time-series database[14] running on a third-party node.[15] WATTSKIT is configured to automatically monitor all the containers deployed within a node with a sampling frequency of 1 Hz. While WATTSKIT can use SWARM and ZOOKEEPER to coordinate the deployment and the execution of software-defined power meters on the nodes, we decided to disable these features to avoid any side-effect on the power consumption analysis of these distributed services.

All the power measurements recorded by the instance of INLUXDB can be easily queried from any client application, like INFLUXDB CLI or GRAFANA,[16] to monitor, explore, analyze, and aggregate the power consumption of the distributed services in real-time. In the following sections, we execute the same benchmarks as in Section II and we introduce new perspectives on the distribution of power consumption per service and across nodes.

B. Monitoring the Service-Level Power Consumption

We start by delivering, in Figure 9, a new view focusing on the service-level power consumptions, independently of the hosting nodes. This view reports on the overall power consumption of the distributed system, masking the idle power consumption of nodes as well as other systems running within the cluster. Within this distributed system, one can observe the limited impact of SWARM and WEAVE on the power consumption of the cluster along the execution, while ELASTICSEARCH and ZOOKEEPER can be considered as particularly power-consuming services. Beyond the peaks of activity due to the execution of the ZOOKEEPER and YCSB benchmarks, one can also observe that each of these services exhibits some residual power consumption along the scenario to maintain their distributed state. More generally, ZOOKEEPER imposes a larger energy footprint than any other distributed services, consuming 49.27% of the distributed system, due to the consensus algorithm it implements [12]. Additionally, when sequentially killing the nodes, one can observe the energy impact of running the leader election process (at $t = 630$ sec. and $t = 670$ sec.).

[14]https://influxdata.com

[15]The backend services of WATTSKIT can be deployed within the cluster or on any remote node.

[16]http://grafana.org

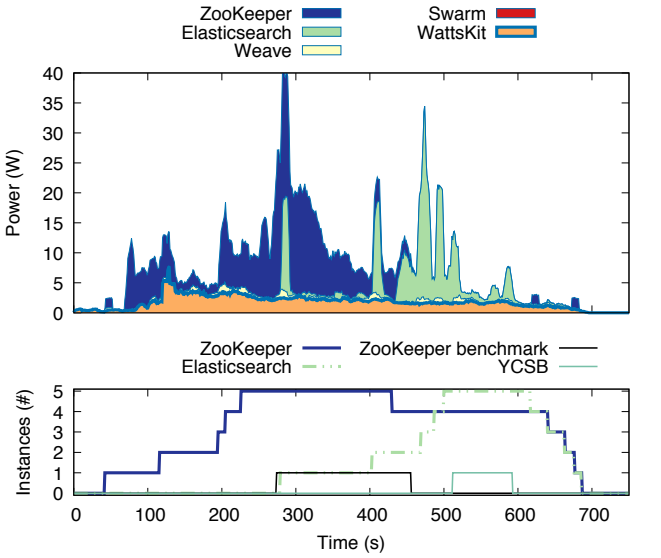

Fig. 9. Monitoring the distribution of the power consumption of a distributed system in a cluster.

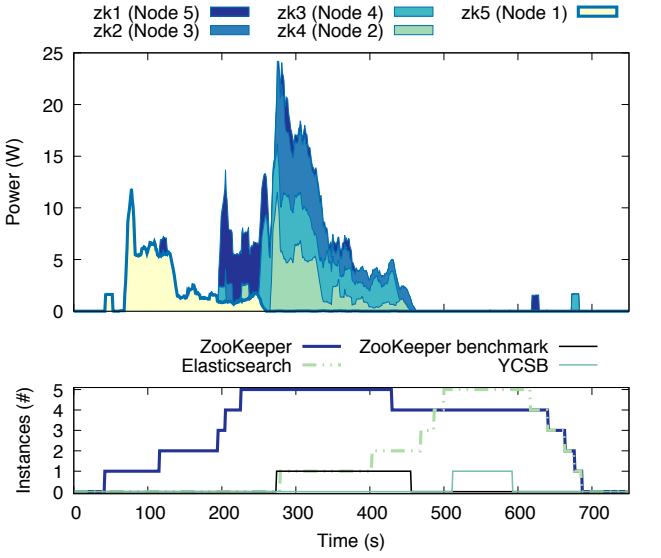

Fig. 10. Analyzing the distribution of the power consumption of ZOOKEEPER across nodes.

WATTSKIT can therefore be considered as a relevant and cheap alternative to physical PDUs by providing a toolkit solution to build accurate and fine-grained power meters for distributed services.

C. Analyzing the Power Consumption per Service

By taking a closer look to individual services composing the distributed system, we can also use WATTSKIT to zoom inside one of these services, thus resulting to finer views. In particular, we describe in Figure 10 the distribution of the power consumption of ZOOKEEPER across the nodes used for its deployment. This perspective on ZOOKEEPER illustrates

that the power consumption of such a service is distributed but not equally balanced across the nodes. It also illustrates that the nodes 5 (zk1) and 4 (zk3) are running the two leader elections we identified towards the end of the scenario.

This granularity of power consumption understanding was particularly difficult to achieve using the coarse-grained power measurements reported in Figure 2 and WATTSKIT clearly advances the state-of-the-art with respect to that. In particular, we believe that WATTSKIT can help software engineers to better understand the energy footprint of their services once deployed in production, thus investigating potential optimizations. WATTSKIT can also benefit to system administrators by investigating the impact of the configuration parameters exposed by the individual services on the power consumption of the distributed system.

VI. RELATED WORK

The design of power models has been regularly considered by the research community over the last decade [1, 2, 4, 5, 8, 11, 13, 14, 15, 16, 25, 26, 27, 28]. In particular, as most architectures do not provide fine-grained power measurement capabilities, McCullough *et al.* [16] argued that power models were the first step towards enabling dynamic power management for power proportionality at all levels of a system. Nowadays, the closest approach to hardware-based power monitoring is the *running average power limit* (RAPL) feature, introduced in the Intel "Sandy Bridge" architecture to report on the power consumption of the entire CPU package. However, this feature is not available on all processor architectures and is not always as accurate as one could expect [5].

Power modeling therefore tends to build on selected raw metrics to apply some machine learning techniques—for example based on sampling [2]—to correlate the metrics with hardware power measurements using various regression models [2, 4, 8, 11, 16, 27, 28]. Three key steps are commonly used to train a power model: the workload(s) to run during learning phase, the minimal set of input parameters, and the form of regression to use [2, 8, 27, 28].

The workloads used while training have to be carefully selected because they represent the targeted behavior to model. Many benchmarks have been identified, but they are, most of the time, specifically designed for an architecture [2, 11], manually selected [4, 5, 6, 8, 14, 26, 27, 28], or even private [28]. A prior work [21] also proposes a technique to generate a synthetic workload by defining its characteristics as inputs. Our approach overcomes this limitation by using standard utilities, which are available on most of UNIX systems, to inject the input workload we use to learn our power models.

The selection of key features used in the power model is also widely addressed by the literature. Existing power models are mostly tailored to a specific processor architecture and manually tuned (including a limited set of power-aware features) [2, 4, 11, 14, 15, 26, 28]. All selected key features are thus based on an *a priori* knowledge of the underlying architecture and the details of the selected HPC events are generally not sufficiently documented [28]. Similar techniques to ours [8, 27] explore the available HPC events, correlate the metrics with the power consumption, and then infer power models. However, these categories of power models are handcrafted [8], or are specific to each input workload [27]. While the multi-core CPU power model proposed in this paper is only assessed on Intel processors, the solution that we describe does not rely on any Intel-specific extensions.

In this paper, we clearly differ from the state-of-the-art by providing an open source, modular, and completely configurable implementation of a software-defined power meters: WATTSKIT. This new generation of software-defined power meters leverages the monitoring of distributed services by providing new perspectives on their power consumption. To the best of our knowledge, WATTSKIT is the first power meter to offer this granularity of monitoring. Furthermore, our implementation of WATTSKIT is the first to automatically learn the power model of a node (independently of its architecture and its features) and then use it for real-time power estimations of software processes. Unlike existing approaches published in the literature, the approach we describe is *i)* architecture agnostic, *ii)* processor aware, and *iii)* easily configurable.

For the purpose of this paper, we based our evaluation on standard benchmarks (*e.g.*, PARSEC, YCSB, ZOOKEEPER benchmark), which are freely available, to assess the validity of our service-level and node-level power models.

VII. CONCLUSION

In this paper, we presented a software-defined power meter, named WATTSKIT, for monitoring the power consumption of distributed systems. Such software meters provide an accurate alternative to dedicated hardware systems or embedded power counters by estimating power consumption at the granularity of services running across several nodes. With WATTSKIT, we cross the boundaries of physical hosts and we provide an estimation of the power consumption of applications spanning several physical (or virtual) machines. To minimize the estimation error in VMs, WATTSKIT needs to deliver accurate power estimation for a wider diversity of services. We therefore developed a service-level power model that conciliates the heterogeneity and the complexity of modern processors, including *multi-cores*, *hyper-threading*, *dynamic voltage/frequency scaling*, and *dynamic overclocking* features that impact power consumption. This power model runs in WATTSKIT without hardware support or system alterations to deliver accurate power estimation (with an average error of 1.35% on an Intel Xeon W3520). This power model is exploited within an instance of software-defined power meter, which can be deployed across all the nodes of a cluster to monitor the power consumption of distributed systems in real-time. It is noteworthy that the proposed solution can be scaled to multiple services and nodes, depending on the complexity of the environment. We evaluated the applicability of WATTSKIT on three processor architectures, and we showed that it performs well for different kinds of distributed protocols and algorithms we considered. Beyond the results we already obtained with WATTSKIT, we plan to consider multi-tenant scenarios in

order to better discriminate the power consumption shares of distributed services used by different stakeholders (including users and client applications). Since the trend is to run software not only locally, but also in data centers and clouds, we expect WATTSKIT to represent a valuable contribution for researchers, developers, and engineers by offering key insights on distributed protocols and algorithms optimizations as well as alternative configurations. The code is freely available as open source.[17]

ACKNOWLEDGEMENTS

This work was partially supported by grants from CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015–2020 and from the ANR SATAS project (ANR-15-CE40-0017).

REFERENCES

[1] Frank 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.

[2] Ramon Bertran et al. "Decomposable and Responsive Power Models for Multicore Processors Using Performance Counters". In: *Proceedings of the 24th ACM International Conference on Supercomputing*. 2010.

[3] Christian Bienia and Kai Li. "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors". In: *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*. 2009.

[4] W Lloyd Bircher et al. "Runtime identification of microprocessor energy saving opportunities". In: *Proceedings of the International Symposium on Low Power Electronics and Design*. 2005.

[5] Maxime Colmant et al. "Process-level Power Estimation in VM-based Systems". In: *Proceedings of the 10th European Conference on Computer Systems*. 2015.

[6] Gilberto Contreras and Margaret 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.

[7] Brian F Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154.

[8] Manuel F. Dolz et al. "An analytical methodology to derive power models based on hardware and software metrics". In: *Computer Science - Research and Development* (2015).

[9] Dimitris Economou, Suzanne Rivoire, and Christos Kozyrakis. "Full-system power analysis and modeling for server environments". In: *In Workshop on Modeling Benchmarking and Simulation*. 2006.

[10] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 2015.

[17] http://wattskit.powerapi.org

[11] Canturk Isci and Margaret 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.

[12] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. "Zab: High-performance broadcast for primary-backup systems". In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2011, pp. 245–256.

[13] Aman Kansal et al. "Virtual Machine Power Metering and Provisioning". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. 2010.

[14] Tao Li and Lizy Kurian John. "Run-time Modeling and Estimation of Operating System Power Consumption". In: *SIGMETRICS Perform. Eval. Rev.* (2003).

[15] Min Yeol Lim, Allan Porterfield, and Robert Fowler. "SoftPower: Fine-grain Power Estimations Using Performance Counters". In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. 2010.

[16] John C. McCullough et al. "Evaluating the Effectiveness of Model-based Power Characterization". In: *Proceedings of the USENIX Annual Technical Conference*. 2011.

[17] Patrik 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 (visited on 08/01/2016).

[18] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "Monitoring energy hotspots in software - Energy profiling of software code". In: *Autom. Softw. Eng.* (2015).

[19] Martin Odersky et al. *An Overview of the Scala Programming Language*. Tech. rep. EPFL Lausanne, Switzerland, 2004.

[20] Anne-Cécile Orgerie, Marcos Dias de Assuncão, and Laurent Lefèvre. "A Survey on Techniques for Improving the Energy Efficiency of Large-scale Distributed Systems". In: *ACM Comput. Surv.* (2014).

[21] S. Polfliet, F. Ryckbosch, and L. Eeckhout. "Automated Full-System Power Characterization". In: *Micro, IEEE* (2011).

[22] Suzanne Rivoire et al. "JouleSort: a balanced energy-efficiency benchmark". In: *Proceedings of the ACM SIGMOD international conference on Management of data*. 2007.

[23] P. J. Rousseeuw and A. M. Leroy. *Robust Regression and Outlier Detection*. 1987.

[24] Guoming Tang et al. "Zero-Cost, Fine-Grained Power Monitoring of Datacenters Using Non-Intrusive Power Disaggregation". In: *Proceedings of the 16th Annual Middleware Conference*. 2015.

[25] Daniel Versick, Ingolf Wassmann, and Djamshid Tavangarian. "Power Consumption Estimation of CPU and Peripheral Components in Virtual Machines". In: *SIGAPP Appl. Comput. Rev.* (2013).

[26] Hailong Yang et al. "iMeter: An integrated VM power model based on performance profiling". In: *Future Generation Computer Systems* (2014).

[27] Reza Zamani and Ahmad 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.

[28] Yan Zhai et al. "HaPPy: Hyperthread-aware Power Profiling Dynamically". In: *Proceedings of the USENIX Annual Technical Conference*. 2014.