Online Power Consumption Estimation for Functions in Cloud Applications

Norbert Schmitt*, Lukas Iffländer†, André Bauer‡ and Samuel Kounev§
Chair of Software Engineering
University of Würzburg
Würzburg, Germany
Email: *norbert.schmitt@uni-wuerzburg.de, †lukas.ifflaender@uni-wuerzburg.de,
‡andre.bauer@uni-wuerzburg.de, §samuel.kounev@uni-wuerzburg.de

***Abstract*—The growth of cloud services leads to more and more data centers that are increasingly larger and consume considerable amounts of power. To increase energy efficiency, informed decisions on workload placement and provisioning are essential. Micro-services and the upcoming serverless platforms with more granular deployment options exacerbate this problem. For this reason, knowing the power consumption of the deployed application becomes crucial, providing the necessary information for autonomous decision making. However, the actual power draw of a server running a specific application under load is not available without specialized measurement equipment or power consumption models. Yet, granularity is often only down to machine level and not application level.**

In this paper, we propose a monitoring and modeling approach to estimate power consumption on an application function level. The model uses performance counters that are allocated to specific functions to assess their impact on the total power consumption. Hence our model applies to a large variety of servers and for micro-service and serverless workloads. Our model uses an additional correction to minimize falsely allocated performance counters and increase accuracy. We validate the proposed approach on real hardware with a dedicated benchmarking application. The evaluation shows that our approach can be used to monitor application power consumption down to the function level with high accuracy for reliable workload provisioning and placement decisions.

***Keywords*-Energy efficiency, serverless, micro-services, code-offloading, DevOps**

I. Introduction

According to a New York Times study from 2012, data centers worldwide consume about 30 billion watts [1]. With the increasing trend to move services to the cloud, the size and number of data centers continues to grow. Especially the Internet of Things (IoT) is rapidly growing with an estimated 20 to 30 billion devices by 2020 [2]. IoT devices produce data that normally requires some processing causing additional load on the cloud. In addition to the popular cloud services, IoT devices capable of code offloading increase the load on data centers. Millions of battery powered devices can push computations to the cloud to preserve battery power. This increased load results in higher power consumption for the sum of all data centers. It is estimated at 140 billion kilowatt-hours annually by 2020. Compared to 91 billion from 2013 [3], this is an increase of 54%. Therefore, the need to make data centers more energy efficient and reduce the power consumption is becoming increasingly important.

To conserve energy, accurate information about the current state of servers and their workloads is necessary. This information allows making well-informed decisions on workload placement and provisioning. Upcoming micro-service architectures and serverless platforms allow for more granular deployment options down to singular functions. This trend is known as Function-as-a-Service (FaaS). As the choice of deployment options increases, having detailed knowledge about the power consumption of the deployed software becomes crucial. The actual power draw of a server running a specific application under load is not available without specialized measurement equipment or power consumption models. The expenditure on measurement equipment is high, even for smaller data centers. Power models, on the other hand, are often only available on the machine level and not on the application or function level, necessary for autonomous and fine-grained deployment decision making. In this paper, we address this issue by proposing power models on the function level designed to be usable without requiring specialized measurement equipment.

Current research in energy efficiency is focused on the energy efficiency of server hardware [4], as well as energy efficiency on the data center level. For example, many techniques have been proposed for application consolidation inside a data center through virtualization [5]. With new emerging paradigms, such as serverless computing, individual functions of an application are deployed rather than a full application stack. With the move towards fine-grained deployments, away from classical multi-tier applications, hierarchical energy saving techniques [6] become less suitable. Code offloading (i.e., moving computations to the cloud to conserve battery power), also leads to deploying only parts of an application [7]. The fine-granular control of the deployment limits the use of existing power models, which normally do not capture the power consumption in such detail. If the software in question is also developed in a DevOps environment, code changes can occur more often. These changes can impact power consumption and consequentially also impact the optimal deployment. Modeling a software system's power consumption is normally either done by training a model with measurements [8] or by

defining a workload profile from user input combined with expert knowledge on the hardware [9].

In this paper, we propose a new online power modeling approach on the application function level based on performance counters. Instead of using the performance counters to determine the total power consumption of a system, we attribute the counted performance events to application functions based on the stack trace of the monitored application. Our approach constructs the stack trace without requiring to instrument the application code manually. This allows to determine the power consumption of specific functions, even if their implementation changes frequently, allowing for autonomous (but also manual) informed decision making about energy efficiency.

The main contributions of this work are:

- A novel technique to attribute energy consumption through globally recorded performance counters to specific functions without changing the business logic that can act as an input to autonomous placement decision making.
- A method for relative comparison of functions, portable across different systems, allowing developers to improve the source code to increase energy efficiency and autonomous deployment of the most efficient version of a function.

The remainder of this paper is structured as follows: First, we describe our approach and the underlying foundations in Section II. Section III shows our testbed setup, which is used to evaluate our approach. In Section V, related work is reviewed. Finally, after an outlook on future work in Section VI, the paper is wrapped up in Section VII.

II. Approach

Our approach determines the power consumption of application functions without requiring to change the application itself. The source code can remain free of instrumentation code that has no relation to the actual business logic. Existing applications can be monitored without change and specialized measurement equipment hardly present in most data centers.

A. Foundations

We must first show that energy efficiency is not only a matter of efficient hardware but is also influenced by the software running on said hardware. We further present the used power model and benchmarking application.

Following the principle that applications indirectly control the hardware they are running on [10], we base our approach on improving the power consumption of the software rather than the hardware. To investigate the validity of this principle and in turn our approach, we measured the energy consumption (power over time) of three different variants of a REST service for 240s. This service performs caching and resizing image files stored on a physical drive. While the resizing needs processing by all variants, the implementation of the data structure for the cache varies. Variants include a random replacement strategy (RR) as a linked list, a least frequently used strategy (LFU) with a red-black tree, and no cache with direct drive access (DDA). The results in Table I show that the DDA uses the most amount of energy per request. It seems that the constant drive access is responsible for the higher energy consumption but even LFU and RR exhibit differences in energy consumption supporting our assumption. With a constant request rate, the RR caching strategy will consume the least amount of energy in this scenario. The lower energy consumption for RR most likely occurs due to the random selection of images. Singh et al. also confirmed the impact of software on efficiency by showing that different software designs, implementations, and configuration parameters for the same logic result in different energy consumption [11], [12].

Implementation	Energy [Ws]
Direct Drive Accesss (DDA)	3.78
Least Frequently Used (LFU)	3.68
Random Replacement (RR)	3.43

TABLE I
Energy consumption of different cache implementations per request over a 240s measurement period.

The power consumption P_{total} of a system consists of two parts, the static or idle power consumption P_{static} and the dynamic power consumption $P_{dynamic}$, as shown in Eq. (1). The dynamic power is dependent on the systems utilization or CPU utilization [10], [13], [14]. The idle consumption must be measured and removed from the total power to determine the power consumption caused by the running software. We assume that $P_{idle} = P_{static}$ for our approach. The relation of power and temperature is considered by operating our Systems Under Test (SUTs) in a controlled environment inside an air-conditioned data center and warm-up periods before measurements.

$$P_{total} = P_{static} + P_{dynamic} \tag{1}$$

To derive the power consumption without actually measuring it, our approach uses performance counters, a statistics feature available in modern CPUs. Their main use is identifying performance bottlenecks in an application by counting specific events occurring in a CPU. However, performance counters have been shown to be correlated with power consumption, which can be leveraged to model power consumption as shown in existing work [15]–[21]. Typically, regression models (Equation 2) are used in this context. $\mathbf{Y}$ is the *response* variable, that is, $P_{dynamic}$ in our case. $\mathbf{X}$ is the vector of *regressor* variables for which we use the monitored counts of performance events. The regression parameters β must be trained to derive the power model.

$$\mathbf{Y} \approx f(\mathbf{X}, \beta) \tag{2}$$

The selection of performance events is critical to building a viable model and needs expert knowledge about the system. To avoid overfitting the regression model to specific events

relevant to our scenario, we use the identified events in the work of Yasin [22]. In his work, he uses performance events for systematically identifying bottlenecks. As performance and power consumption are related, but not identical, we see this as a useful selection of performance events without fitting the model to a particular application. The following performance counter descriptions are taken from the Intel manual and shortened for brevity [23].

- *CPU_CLK_UNHALTED.THREAD*: "The event counts the number of core cycles while the logical processor is not in a halt state."
- *IDQ_UOPS_NOT_DELIVERED.CORE*: "Counts the number of uops that the Resource Allocation Table (RAT) issues to the Reservation Station (RS)."
- *UOPS_ISSUED_ANY*: "Counts the number of uops not delivered to Resource Allocation Table (RAT)."
- *UOPS_RETIRED.RETIRE_SLOTS*: "Counts the retirement slots used."
- *INT_MISC.RECOVERY_CYCLES*: "Core cycles the allocator was stalled due to recovery from earlier machine clear event for this thread."
- *CYCLE_ACTIVITY.STALLS_MEM_ANY*: "Execution stalls while memory subsystem has an outstanding load."
- *RESOURCE_STALLS.SB*: "Cycles stalled due to no store buffers available."

We monitor performance counters per CPU core. As we do not want to instrument our application directly, we need a method to attribute the performance events to application functions. To keep the source code of the application as it is, we obtain the stack trace with Kieker[1], running concurrently to the application. Kieker claims an overhead below 3% [24]. The application we selected is the image provider service of the TeaStore[2] [25] benchmarking application. TeaStore is a Java application, specifically built for testing power and performance models. The associated image provider uses CPU (resizing images), memory (through caching), and I/O (loading uncached images from storage). It features recursive and non-recursive function calls allowing to validate our model's compatibility with recursion. Function call overhead is attributed to the calling function.

B. Model

To calculate the power consumption of our application, we need to first know for how long a function of our application has exclusive access to the processing resources. For this, we state that if a function is on top of the call stack it has exclusive access. An example stack trace is shown in Figure 1 with five different functions. The time a function x resides on stack height s can then be calculated with Equation 3 by adding up the times between when the function was called $t^s_{x,n}$ and its return $t^s_{x,n+1}$.

$$\sum_{i=1}^{n-1} t^s_{x,n+1} - t^s_{x,n} \tag{3}$$

As functions can call subroutines, we also calculate the time any function was atop the caller up to the maximal stack height s_{max} of the current interval Δt, as shown in Equation 4.

$$\sum_{j=s+1}^{s_{max}} \sum_{i=1}^{n-1} t^j_{i+1} - t^j_i \tag{4}$$

By subtracting the time of the callee from the time of the caller and dividing it by the sampling interval time $\Delta t = 1000$ms (see Equation 5), we calculate the relative time-share $f_x \leq 1$ of how long a function x has exclusive resource access as follows:

$$f_x = \frac{\sum_{i=1}^{n-1} t^s_{x,i+1} - t^s_{x,i} - \sum_{j=s}^{s_{max}} \sum_{i=1}^{n-1} t^j_{i+1} - t^j_i}{\Delta t} \tag{5}$$

The server operating systems (usually being preemptive and able to withdraw resources) weakens the assumption that an application has exclusive access to a machine's resources, which introduces inaccuracies in the measurement of the stack trace and performance counters. The withdrawal of resources is reflected in the stack trace by a prolonged time between the function call and its return. As our approach distributes performance events according to the time a function spends on top of the stack, this subsequently leads to more performance events assigned to a function than it actually causes. Not only can the prolonged time-share affect the performance event counts, but also the callee of a subroutine. For example, function a in Figure 1 is called once from function e and later twice from function b. Taking that function e generates event E that is not present for function b, then function a falsely gets assigned event E. Additionally, the preemptive operating system will also lead to higher recordings of performance events. This effect leads to two inaccuracies in our approach. First, a measurement sample can contain more performance events and secondly, a longer time-share f_x for the interrupted functions adding performance events not associated with a call to the functions.

The first problem is countered by measuring performance events in the idle state and removing them from the performance counter samples. We assume that the operating system must run independently of the workload. A network heavy application would force the operating system to empty the network interface buffers more frequently, limiting workload independence. The power consumption estimation should reflect whether the application is responsible for higher operating system interventions. Only preemptions that also take place in an idle state and are not attributable to the application should be removed. While this still leaves errors in the stack trace, the application in question does not need instrumentation to pause performance event recording if control of the resources is not available.

[1]Kieker Application Monitoring: http://kieker-monitoring.net/
[2]TeaStore: http://descartes.tools/teastore

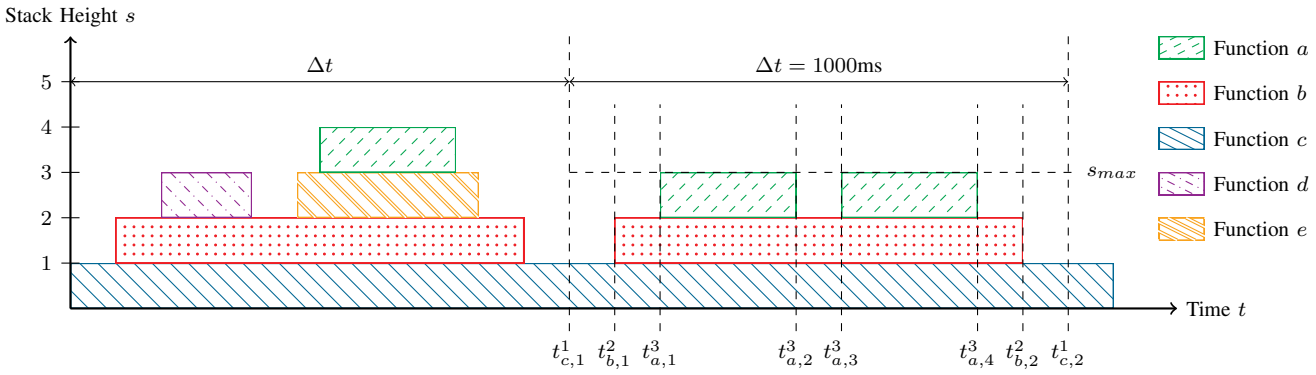


Fig. 1. Example stack trace with five functions, performance counter sampling interval Δt and stack sample times.

Introducing a correction factor to our performance event assignment addresses the second problem. Removing falsely counted performance events does not correct the prolonged time-share and still leads to disproportional assignments of performance events to interrupted functions. Hence, we introduce a correction factor c_t shown in Equation 6.

$$\mathbf{c}_t = \begin{bmatrix} c_{0,a} & c_{1,a} & \dots & c_{n,a} \\ c_{0,b} & c_{1,b} & \dots & c_{n,b} \\ \vdots & \vdots & \ddots & \vdots \\ c_{0,z} & c_{1,z} & \dots & c_{n,z} \end{bmatrix} \tag{6}$$

Each factor $c_{E,x}$ is built through an unweighted moving average over the last n number of performance events of event E assigned to function x in Equation 7. The smoothing over the history reduces the amount of wrongfully assigned performance events. This correction is limited, as functions only called from one specific callee are not correctable since the history never contains samples without the wrong performance events.

$$c_{E,x} = \frac{1}{n} \sum_{i=0}^{n-1} p_{corrected,x,M-i} \tag{7}$$

Combining a function's time on top of the stack f_x (Equation 5) and the monitored performance events into a column and row vector respectively in Equation 8, we finally calculate the corrected amount of performance events for each function $p_{corrected,t}$ at time t in Equation 9. The total time-share for all functions $\mathbf{f}_t^T \leq 1$ must be fulfilled.

$$\begin{aligned} \mathbf{f}_t^T &= \begin{bmatrix} f_a & f_b & \dots & f_z \end{bmatrix} \\ \mathbf{p}_t &= \begin{bmatrix} pc_0 & pc_1 & \dots & pc_n \end{bmatrix} \end{aligned} \tag{8}$$

$$\mathbf{p}_{corrected,t} = \left(\frac{\mathbf{f}_t \cdot \mathbf{p}_t + \mathbf{c}_t}{2} \right) \tag{9}$$

In combination with the regression model, we can now determine the power consumption per function as the moving average over the last $n+1$ assignments.

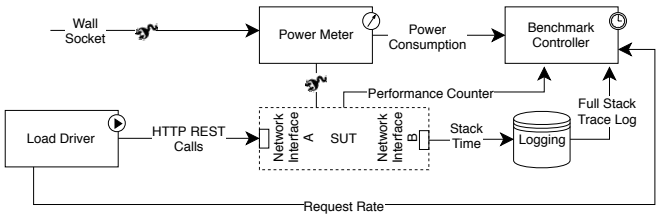


Fig. 2. Evaluation testbed setup.

Hardware power management, like Dynamic Voltage and Frequency Scaling (DVFS), are not directly modelled but indirectly through the use of performance counters. Disabling DVFS would increase accuracy but would reduce applicability in realistic applications. With decreasing clock speed, the number of performance counters decreases as well and the runtime increases. As we assume a linear relation between power (represented by performance counters) and runtime, the energy efficiency is identical.

C. Testbed Setup

To investigate whether our model can distinguish between the application's functions and the overhead generated by the software stack and operating system, we set up the testbed shown in Figure 2. We use Apache JMeter[3] as a load driver to issue HTTP POST requests to the system under test (SUT) over a dedicated network interface. Thus, the employed monitoring tool Kieker does not create overhead inside the backbone network. Kieker sends generated stack traces through a second network interface to a dedicated logging server. A Hioki PW3335 power meter connected to the benchmark controller measures the SUT's wall power. After each measurement, the benchmark controller collects the request rate, performance counters, and stack traces.

We use three physical servers as SUTs. Table II lists the SUT servers. Each SUT has a different Intel Xeon CPU generation, different core and thread counts (4/8, 8/16, 12/24) and the *medium* and *large* SUT also have double memory

[3]Apache JMeter: https://jmeter.apache.org/

SUT	CPU (Cores/Threads)	Memory
Small *(S)*	E3-1230 v5 @ 3.40GHz (4/8)	16GB
Medium *(M)*	E5-2640 v3 @ 2.60GHz (8/16)	32GB
Large *(L)*	E5-2650 v4 @ 2.20GHz (12/24)	32GB

TABLE II
SERVERS USED AS SYSTEM UNDER TEST (SUT).

capacity. The TeaStore image provider is deployed eight times on each bare-metal SUT in Docker containers, including application stack monitoring. Each Docker container hosts an Apache Tomcat application server running the image provider service. We selected eight instances of the image provider and limited the CPU usage to 1 via Docker, which is the largest deployment runnable on all SUTs without constraints. This also eliminates possible accumulation errors due to multithreading. Using the threadcount as the maximum for each SUT resulted in memory contention on the large SUT and subsequently a minimal ability to process any requests on time.

All SUTs use the same load profile with four different load levels shown in Figure 3. This load curve was recorded for the *small* SUT but the *medium* and *large* SUTs are similar. Variations at the highest level are due to limitations on our testbed present on all SUTs. The load driver repeatedly requests random images with sizes ranging from 800 to 950 pixels in width and height from all application instances.

To build our regression model mentioned in II-A, we stress the small SUT with the four different request rates and repeat this measurement nine times. We use six measurements as training data for our model and the remaining three for model validation. The coefficients are listed in Table III.

III. EVALUATION

For our evaluation, we utilize our testbed described in Section II-C. First, we evaluate if our model is valid and

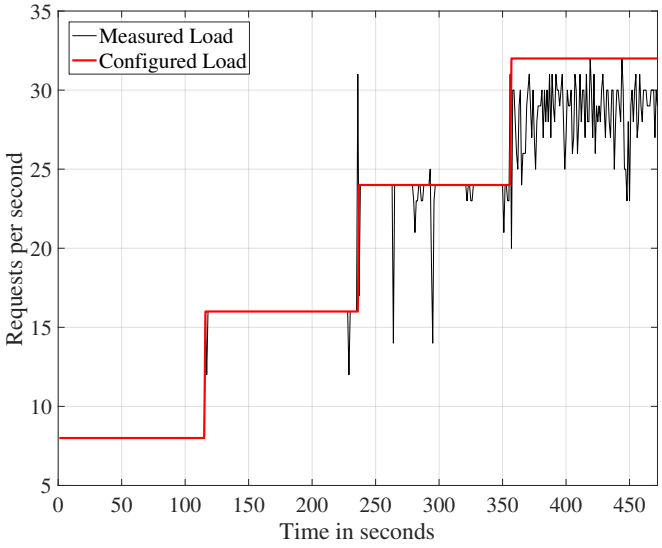


Fig. 3. Configured and measured requests per second.

Performance Event	β
Intercept	1.921
CPU_CLK_UNHALTED.THREAD	$-3.771 \cdot 10^{-9}$
IDQ_UOPS_NOT_DELIVERED.CORE	$-2.437 \cdot 10^{-8}$
UOPS_ISSUED_ANY	$-2.787 \cdot 10^{-9}$
UOPS_RETIRED.RETIRE_SLOTS	$6.270 \cdot 10^{-8}$
INT_MISC.RECOVERY_CYCLES	$7.338 \cdot 10^{-8}$
CYCLE_ACTIVITY.STALLS_MEM_ANY	$9.098 \cdot 10^{-9}$
RESOURCE_STALLS.SB	$6.996 \cdot 10^{-8}$

TABLE III
REGRESSION PARAMETERS β FOR OUR POWER MODEL.

Requests per Second	Absolute Error	Relative Error
8	1.18 W	7.5 %
16	1.15 W	3.5 %
24	1.70 W	3.5 %
32	1.48 W	2.6 %

TABLE IV
ERROR OF THE PREDICTION ABSOLUTE AND RELATIVE TO THE MEASURED VALUES.

confirm that the differentiation of overhead of the software stack and operating system from the actual test application results in plausible values. Second, we show that our model can identify a function's power consumption by filtering out the functions with significantly higher energy consumption. As a last step we check if our approach is transferable to different machines without retraining the regression model. We then treat the model's output as a relative value for comparing implementations across different systems. We assume that performance events caused on the trained system also occur on an untrained system.

A. Application Separation

To determine if our model can distinguish between power consumed on account of the application or otherwise, we perform nine measurements on the *small* SUT. We apply our approach to the recorded time-shares and performance events without the correction factor. The power is calculated for each function and summed. As $\mathbf{f}_t^T \leq 1$, not allocating all performance events towards the monitored application is likely. We consider this as software stack overhead due to the Java Runtime Environment, Docker service, and the operating system.

Figure 4 shows that the power draw from the overhead can make up a considerable amount. Notably, around 170 to 240 seconds for the TeaStore application where the overhead and application assigned power are close to equal. Neither the overhead nor the application can reach the observed wall power for the complete system as expected. If the overhead (P_o) and application power (P_a) are combined to the estimated dynamic power ($P_e = P_o + P_a$), the power consumption is close to the measured wall power of the SUT. Table IV shows the mean and relative errors for the prediction.

While the application power draw has rising steps in con-

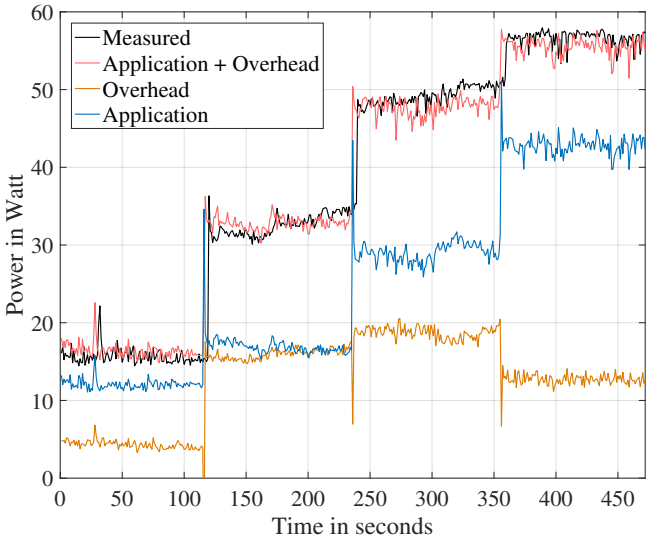


Fig. 4. Mean estimated power consumption of the regression model over nine measurements and actual wall power for the *small* SUT.

junction with the load level, the overhead shows a different behavior. With the increasing load level, the overhead is also expected to grow with the number of calls to the operating system. The overhead climbs with the load level to 24 requests per second. As the load switches to 32 requests per second, the overhead drops sharply. We estimate that the non-linear overhead could be due to falsely assigned performance counters, changes in scheduling, e.g., to avoid Java's garbage collection while close or at full CPU capacity, operating system interventions, simultaneous multithreading or a combination of those. Further research is necessary to find the cause and determine if this is a result of performance events falsely assigned as overhead instead of the application's functions.

There is also a shift of 3 seconds between the model estimation and the actual power measured. We attribute this to certain inertia in the server system power supply as all monitoring probes start within 10ms.

B. Correction Factor

To check if our correction factor $\mathbf{c}_t$ (see Section II-B) can dampen outliers, probably stemming from computations scheduled in between the test application, we calculated our model's power consumption with varying sizes for the moving average history. We let the size of the moving average range from 2 to 60 seconds in steps of 2. Against our expectations, the smoothing over the history only moved performance events from the application to the overhead but not vice versa. Hence, the correction factor in its proposed form is neglectable for further evaluation, and other possibilities should be taken into account.

C. Identifying Function Power Consumption

Our approach can identify the power consumption of a micro-service or serverless cloud application. As the sum of all function's power consumption is accurate, we check each function's contribution to the overall power consumption.

As a first step, we summed up each function's contribution to the proportional power consumption over all load levels. Four functions are consuming over 99% of the total power attributed to the application out of in total 56 functions called during the measurements. The remaining functions only contribute marginally with one order of magnitude below *Function 12*, which is below 1%. The four functions and their relative energy consumption are:

- *Function 08* (67%): Converts the internal image representation of Java into a byte array and encodes it in base64 for the request response.
- *Function 12* (<1%): Creates a new object that can be stored in the cache holding the base64 encoded data.
- *Function 29* (1%): Converts the base64 encoded image to the Java image representation.
- *Function 47* (31%): Resizes images.

As the image provider has a least frequently used replacement strategy, it seems that the substantial impact of these four functions results from the requests selecting random images, causing a low cache hit rate. Hence, images must be converted back to the Java image representation, scaled, stored in the cache, and converted back to base64 for the request response.

We further break down the contribution of each function to the power consumption by load level. Figure 5 shows the power consumption for the four functions. The sharp changes in the load level are not included to let the load driver stabilize. We cut off three seconds before and after each load change. As the power consumption for each function rises with the load level indicating that our approach can correctly attribute a function's impact on the energy usage, variations in the measurement increase with the load level. This behavior matches with the random sizes requested by the load driver querying more images of different sizes, also coinciding with the increasing contribution of *Function 47*, scaling images to the correct size.

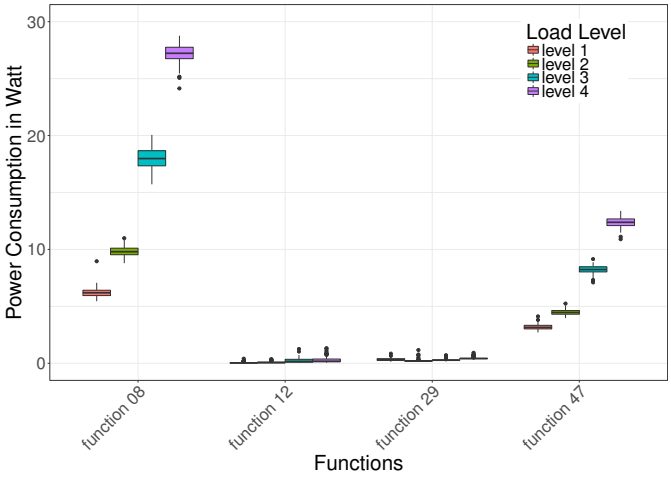


Fig. 5. Function power consumption for four load levels.

		Small/Medium	Small/Large
Normalized	MAE	0.1547 (15%)	0.1495 (15%)
	σ	0.0975	0.1586
	σ^2	0.0095	0.0251
Measured	MAE	5.7683 (10%)	12.4118 (21%)
	σ	2.9736	4.5653
	σ^2	8.8421	20.8416

TABLE V
MEAN ABSOLUTE ERROR, STANDARD DEVIATION AND VARIANCE BETWEEN THE SMALL, MEDIUM AND LARGE SUT.

D. Portability

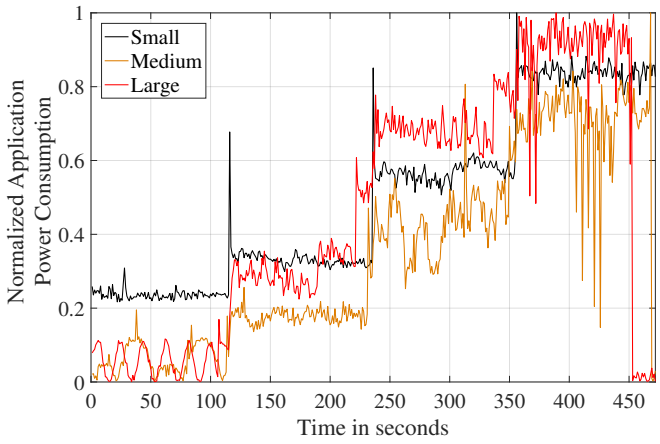
As the software is indirectly controlling the hardware, we assume that performance events occurring on one system are also observable on others. We normalize the power estimation of the model to see if our approach is usable as a reference even without retraining the model. We then compare the normalized estimations of the trained model of the small SUT to the untrained medium and large SUTs. We do not normalize over the measured load. Not every request must necessarily call all functions.

Figure 6a shows the results for a single measurement of each SUT. The small SUT works well as expected while the medium and large SUT exhibit large variations. Especially at the lowest and highest load. The same behavior is present for both SUTs in Figure 6b, clearly visible at the large outliers at the highest load level. The experienced behavior is observed in the measurement as well as the relative application power and therefore can not be completely attributed to the model being untrained on the medium and large SUT's data. The model also overestimates the power consumption for the large SUT. The measured power consumption in Figure 6b shows an ordering from large to small SUT for identical request rates, with the large instance mostly (88% of the time) below the medium instance.

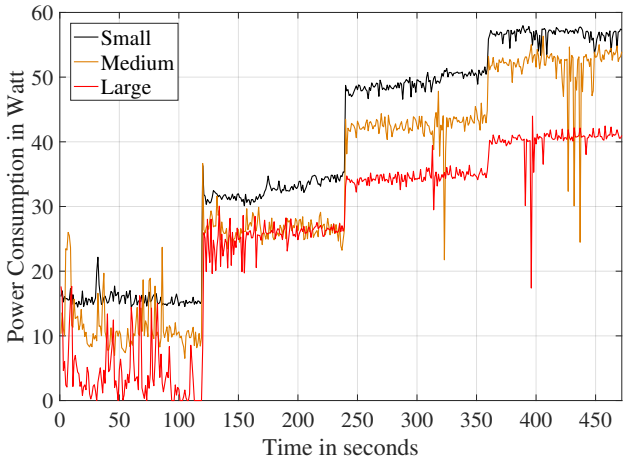
Calculating the absolute error between the trained *small* server and the *medium* and *large* servers, we can see in Table V that the power consumption deviates strongly from the mean value. While the relative application power seems unsuitable for comparison, we take a look if the proportional power consumption remains portable.

In accordance with Section III-C, we examine the proportional power consumption of the application functions in Table VI. Proportional accounting to functions works well, yet small deviations are present and were expected. The functions vary between 3% to 2% for *Function 08* and *Function 29*, respectively. The remaining functions account for 2% of the total power consumption of the medium SUT. We estimate that small changes in the CPU's architecture are responsible for the minor changes shown in Table VI.

We showed that our approach can distinguish between the power draw of our test application and the overhead of the operating system and software stack. The correction factor as a moving average did not yield the expected improvements



(a) Normalized application power.



(b) Measured power.

Fig. 6. Normalized and measured application power consumption of all three SUTs.

	Functions				
SUT	08	12	29	47	others
Small	67%	$< 1\%$	1%	31%	$< 1\%$
Medium	64%	$< 1\%$	3%	31%	2%
Large	66%	$< 1\%$	2%	31%	$< 1\%$

TABLE VI
PROPORTIONAL POWER CONSUMPTION OF APPLICATION FUNCTIONS FOR ALL SUTS.

and is rejected. Further, we showed that the allocation of performance events to specific functions works well and we can identify functions with a high impact on power consumption. It also achieves reasonable results across multiple SUTs.

IV. DISCUSSION AND LIMITATIONS

In our work, we use the performance counters pre-selected by Yasin [22]. Yasin selected his chosen performance counters to detect performance bottlenecks as opposed to predicting a function's power consumption. This choice might impact the resulting quality of our approach. A further exploration

of the performance counter configuration space might yield a set of performance counters even better suitable for our use case. Nevertheless, the presented results already exhibit high accuracy and adequate portability. Therefore, the used counters are sufficient to confirm the applicability of our approach. Since the performance counters are a proxy metric, they represent the true values only to a certain degree. The same is true for Running Average Power Limit (RAPL) counters. Intel states that the RAPL counters are a software model and not on-chip measurement Devices [26], [27]. To not rely on an opaque software model not under our control, we opted for a self-trained performance counter model. Additionally, the most influential power consumer in a server is the CPU, apart from special-purpose systems. While AMD and Intel both have RAPL counters, others might not. To keep our approach portable and transparent we do not use RAPL counters.

Serverless and virtualized environments are often not exposing performance counters. This issue can be resovled by recording the performance events on the host system and a dynamic mapping from CPU core to VM or container. The available counters or power consumption could then be made available to a tenant.

As we try to keep the overhead introduced by the performance counter sampling minimal, we only collect what is necessary and use a low sampling rate of 1s, corresponding to the sampling rate of the power meter. A higher frequency, e.g., on a function basis may increase the accuracy, but would incur a high overhead. As the evaluation shows, 1s is enough to reach a sufficient accuracy. A possible technical limitation of our evaluation is the timer resolution of the monitoring solution Kieker. While Kieker claims a resolution of one nanosecond, inaccuracies are possible. However, over the course of our measurements, these inaccuracies should compensate each other.

In this paper, we assume that power consumption is linearly correlated to the runtime. However, idle phases like IO waits might lead to a non-linear correlation. Micro-service workloads are often a mixture of CPU, memory and IO based computations but linear models have proven to have high accuracy. However, for IO heavy workloads, other models could prove more accurate.

The applied error correction approach has shown no improvement to the uncorrected results. While, as presented above, our results are suitable for our use case, more complex error correction or smoothing algorithms might be beneficial. Although our experiments stressed a benchmark application with varying load levels and were conducted on three different hardware scenario, the results may not be generalizable to other types of applications or hardware scenarios.

V. Related Work

We categorize related work into three types: (1) work that deals with the proportionality (or the lack of proportionality) between the performed amount of work and consumed power, (2) work that deals with modeling and optimizing code using performance counters, their possible inaccuracies, and (3) work dealing with power consumption on mobile devices.

A. Energy Proportionality

Barosso and Hölzle prose that Energy Proportionality should become a primary design goal [28], that is, servers should be designed to consume energy proportional to the amount of work performed. While the share of a server's total power consumption attributed to CPU processing is declining due to power-saving features, this is hardly the fact for the other components. The authors demand significant improvements for the memory and disk subsystems. Furthermore, they urge developers to run benchmarks not only at peak load but also at lower load levels.

In [11] Singh et al. show the impact of different buffer sizes used when reading/writing and compressing/decompressing files of 20GB using the Java API. The buffer sizes vary from 1KB to 1GB in steps of power of 2. They conclude that, like performance engineers optimizing parameters, the correct setting can save energy and increase energy proportionality.

These works introduce new concepts but deal with proportionality either on a data center level or server level while our work is focused on the function level.

B. Performance Counters

Multiple works [15]–[20], [29], [30] use performance counters to create models for various applications like power consumption and thread scheduling.

However, performance counters are known to be inaccurate in certain situations. The authors of [31] name multiple problems for the use of performance counters: complex configuration, missing programmability, the requirement of root privileges, and the lack of discrimination between the triggering threads. The authors then perform a comparative study of the accuracy of three commonly used measurement infrastructures for performance counters on three different processors. While this is correct for older architectures, CPU vendors provide higher accuracy for most performance counter values by now.

Therefore, they are suitable as a tool to optimize power consumption. In [32] Wu and Taylor describe the methodology of using performance counters to reduce the power consumption of high-performance computers and supercomputers. They use 37 counters to identify energy saving potential inside large workloads. They select a small subset of 37 counters and focus on them. The function is then optimized through further analysis and expert knowledge.

While these works lay an excellent foundation to build upon, none of them deals with optimizing the energy proportionality of an application.

C. Power Consumption on Mobile Devices

In [33] Li et al. present an approach to measure power consumption on a source line level. To perform this measurement, the authors combine hardware-based power measurements with program analysis and statistical modeling. While

executing an application, the approach measures the energy and derives the executed parts of the application using path profiling. The per line consumption is then evaluated using static and regression analysis as presented in the authors' previous work [34]. However, this approach requires a significant amount of resources to perform the profiling, reducing accuracy compared to an application running at 100%.

The work presented in [35] studies a self-modeling paradigm for a mobile system. Thus, it allows the system to generate its energy model without external assistance. This generation uses the smart battery interface. While this approach presents a respectable precision, the results have been achieved only on mobile hardware and especially relatively old hardware that does not yet employ modern power-saving technologies.

Pathak et al. introduce *eprof*, a fine-grained energy profiler for smartphone apps [36]. When applied to several apps, eprof exposes energy drainers like third-party advertisement or pinpoints the location of wakelock bugs in the code. Next, *bundles* are introduced to help developers to optimize the energy drain of their app by presenting the app's I/O energy consumption.

The work in [37] introduces two tools. *PowerBooter* is a tool to construct a power model without using a power meter. *PowerTutor* is a tool that using online analysis shows developers the implications of their design choices on power consumption. Unlike [33], this approach is not based on the code line level but rather on the component level. Thus, the model is built using the consumption of the CPU, LCD, GPS, Wi-Fi, 3G, and audio components.

For Java-based software systems [38] presents the first iteration of a framework for estimating power consumption. This approach focuses on the interaction among distributed components. Thus, it allows developers to estimate their systems' power consumption at design-time.

These works have in common that they do not directly use CPU performance counters but either use the sensors of a mobile device or Java's bytecode. While this might work for the specific device or language, it is complicated to derive lessons for the general utilization. Also, these works give little consideration to energy proportionality.

VI. Future Work

Based on this work, possible contributions in the future become possible. To face the limitations of this work, a first step is to validate our approach on broader hardware set with different workloads. A next step is to introduce a sample based stack monitoring compared to the event based Kieker solution, minimizing the interference and overhead and thus allowing us to monitor large-scale cloud systems. Further, our model should be compared to existing and well-studied power models. For the DevOps community, providing valuable insight into the power consumption of application functions can be beneficial to conserve energy as well as to support autonomous placement and deployment decisions.

To further reduce the monitoring overhead, we envision a fast stack simulation that returns the current stack layout, together with a detailed overhead evaluation. However, such a solution needs to monitor additional input parameters instead. Compared to the current event based stack monitoring, sample-based or the simulation could achieve a lesser overhead.

As the correction factor for our approach did not work as expected, different alternatives include a Kalman filter and time-series prediction. A prediction, including a confidence value, could improve our work and determine falsely allocated performance events. Static code analysis can also help reduce errors. For example, by analyzing functions, finding disk access through known functions in the language's API, performance events could be shifted towards the causing function away from functions without disk access.

VII. Conclusion

Reducing the power consumption of services is crucial for modern providers to be profitable. The move from infrastructure or platform granular deployment options to function granularity adds additional complexity. Developers and providers aim to optimize the performance per power ratio of the deployed applications and their underlying functions. Therefore, it is necessary for them to know about the composition of a function's power requirement.

In this work, we presented a technique to determining the power consumption of application functions without changing the application itself. We allocated monitored performance events to specific functions. Thereby, it is possible to assess a functions impact on the total power consumption. We evaluated our approach on three different SUTs. To assess the accuracy of our model we compared the calculated values to the measured values. We observed an error of 7.5% for the system under light load and of 2.6% for the system under heavy load. Our approach also correctly yielded the functions responsible for most of the power consumption, making autonomous placement and deployment decisions possible. This also gives insights to developers to improve their code in terms of energy efficiency, as well as energy proportionality, while operators can select machines providing optimal energy efficiency for the most consuming functions.

We also validated the portability for our approach. While the accuracy for the prediction decreases proportional to the commonality to the base system, the estimated share of a functions power consumption remains accurate. Furthermore, we analyzed related works, which provide a foundation to build upon, but do not target the granularity of function level power consumption without dedicated measurement equipment. Finally, we presented planned future extensions to our work.

In conclusion, our approach gives developers and service providers the possibility to discover which functions consume the most power. This ability allows focussing optimization efforts on the relevant functions. Additionally, our application features satisfactory portability and, therefore, applies to heterogeneous systems.

References

[1] J. Glanz, "Power, Pollution and the Internet," *New York Times*, September 23rd 2012.

[2] A. Nordrum, "The internet of fewer things [news]," *IEEE Spectrum*, vol. 53, no. 10, pp. 12–13, 2016.

[3] J. Whitney and P. Delforge, "Data center efficiency assessment," August 2014, https://www.nrdc.org/sites/default/files/data-center-efficiency-assessment-IP.pdf.

[4] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *IEEE Micro*, vol. 32, no. 2, pp. 20–27, March 2012.

[5] Y. Jin, Y. Wen, and Q. Chen, "Energy Efficiency and Server Virtualization in Data Centers: An Empirical Investigation," in *2012 IEEE Conference on Computer Communications Workshops*, Mar. 2012, pp. 133–138.

[6] H. Lim, A. Kansal, and J. Liu, "Power budgeting for virtualized data centers," in *2011 USENIX Annual Technical Conference (USENIX ATC11)*, vol. 59, 2011.

[7] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya, "Mobile code offloading: from concept to practice and beyond," *IEEE Communications Magazine*, vol. 53, no. 3, pp. 80–88, 2015.

[8] C. Seo, S. Malek, and N. Medvidovic, "Component-level energy consumption estimation for distributed java-based software systems," in *International Symposium on Component-Based Software Engineering*. Springer, 2008, pp. 97–113.

[9] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 92–101.

[10] C. Calero and M. Piattini, *Green in software engineering*. Springer, 2016.

[11] J. Singh, K. Naik, and V. Mahinthan, "Impact of developer choices on energy consumption of software on servers," *Procedia Computer Science*, vol. 62, pp. 385 – 394, 2015, proceedings of the 2015 International Conference on Soft Computing and Software Engineering (SCSE'15).

[12] E. Capra, C. Francalanci, and S. A. Slaughter, "Measuring application software energy efficiency," *IT Professional*, vol. 14, no. 2, pp. 54–61, 2012.

[13] R. B. ad Anton Beloglazov and J. H. Abawajy, "Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges," *CoRR*, vol. abs/1006.0308, 2010.

[14] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ACM SIGARCH computer architecture news*, vol. 35, no. 2. ACM, 2007, pp. 13–23.

[15] C. Isci and 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*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003.

[16] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters," *SIGARCH Comput. Archit. News*, vol. 37, no. 2, pp. 46–55, Jul. 2009.

[17] X. Chen, C. Xu, R. P. Dick, and Z. M. Mao, "Performance and power modeling in a multi-programmed multi-core environment," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 813–818.

[18] M. Y. Lim, A. Porterfield, and R. Fowler, "Softpower: Fine-grain power estimations using performance counters," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 308–311.

[19] W. L. Bircher and L. K. John, "Complete system power estimation using processor performance events," *IEEE Transactions on Computers*, vol. 61, no. 4, pp. 563–577, April 2012.

[20] R. Rodrigues, A. Annamalai, I. Koren, and S. Kundu, "A study on the use of performance counters to estimate power in microprocessors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 60, no. 12, pp. 882–886, Dec 2013.

[21] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent gpu architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 673–686.

[22] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 35–44.

[23] *Intel 64 and IA-32 Architectures Software Developers Manual*, Intel Corporation, May 2018.

[24] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous monitoring of software services: Design and application of the kieker framework," Kiel University, Forschungsbericht, November 2009.

[25] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research," in *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '18, September 2018.

[26] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '10. New York, NY, USA: ACM, 2010, pp. 189–194.

[27] S. Pandruvada, "Running average power limit – rapl," Online (https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl), June 2014, accessed: 29.01.2019.

[28] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, no. 12, pp. 33–37, 2007.

[29] C. Lively, V. Taylor, X. Wu, H.-C. Chang, C.-Y. Su, K. Cameron, S. Moore, and D. Terpstra, "E-amom: an energy-aware modeling and optimization methodology for scientific applications," *Computer Science - Research and Development*, vol. 29, no. 3, pp. 197–210, Aug 2014.

[30] G. L. Tsafack Chetsa, L. Lefèvre, J.-M. Pierson, P. Stolf, and G. Da Costa, "Exploiting performance counters to predict and improve energy performance of HPC systems," *Future Generation Computer Systems*, vol. vol. 36, pp. pp. 287–298, Jul. 2014.

[31] D. Zaparanuks, M. Jovic, and M. Hauswirth, "Accuracy of performance counter measurements," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009, pp. 23–32.

[32] X. Wu and V. Taylor, "Utilizing hardware performance counters to model and optimize the energy and performance of large scale scientific applications on power-aware supercomputers," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1180–1189.

[33] D. Li, S. Hao, W. G. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 78–89.

[34] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 92–101.

[35] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 335–348.

[36] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 29–42.

[37] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2010, pp. 105–114.

[38] C. Seo, S. Malek, and N. Medvidovic, "An energy consumption framework for distributed java-based systems," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 421–424.