VESTA: Power Modeling with Language Runtime Events

JOSEPH RASKIND, SUNY Binghamton, USA
TIMUR BABAKOL, SUNY Binghamton, USA
KHALED MAHMOUD, SUNY Binghamton, USA
YU DAVID LIU, SUNY Binghamton, USA

Power modeling is an essential building block for computer systems in support of energy optimization, energy profiling, and energy-aware application development. We introduce Vesta, a novel approach to modeling the power consumption of applications with one key insight: *language runtime* events are often correlated with a sustained level of power consumption. When compared with the established approach of power modeling based on hardware performance counters (HPCs), Vesta has the benefit of solely requiring application-scoped information and enabling a higher level of explainability, while achieving comparable or even higher precision. Through experiments performed on 37 real-world applications on the Java Virtual Machine (JVM), we find the power model built by Vesta is capable of predicting energy consumption with a mean absolute percentage error of 1.56%, while the monitoring of language runtime events incurs small performance and energy overhead.

CCS Concepts: • **Software and its engineering → Virtual machines**; **Runtime environments**; • **Hardware → Power estimation and optimization**.

Additional Key Words and Phrases: power modeling, language runtimes, Java virtual machines, BPF

ACM Reference Format:
Joseph Raskind, Timur Babakol, Khaled Mahmoud, and Yu David Liu. 2024. VESTA: Power Modeling with Language Runtime Events. *Proc. ACM Program. Lang.* 8, PLDI, Article 172 (June 2024), 26 pages. https://doi.org/10.1145/3656402

1 INTRODUCTION

As of 2023, data centers constitute approximately 2% of total electricity consumption in both the US [39] and the EU [38]. Tools for tracking the energy and power consumption of the computing stack allow developers to build more energy-conscious systems [4, 15, 37, 53] and contribute in sustainable computing [17, 21, 23, 35]. Broadly speaking, power consumption can be tracked in two ways: *measure it* or *model it*. Measurement-based approaches require meter deployment and physical access to the computing platform. In contrast, modeling-based approaches are easy to deploy and have gained popularity over the years. The most established approach for power modeling relies on monitoring architectural events—e.g., Hardware Performance Counters (HPCs)—and predicting power consumption based on their occurrences [6, 7, 9, 26–29, 36, 51, 52].

A key insight of this paper is that *language runtime events* may impact the power behavior of the application, and the correlation of the two may open up a new avenue for building power models. Take Java applications for example. Intuitively, the diverse behavior of their runtime—e.g., heap management, thread management, just-in-time compilation (JIT), and garbage collection (GC)—may impact how the underlying systems and hardware are used. Compared with HPC-based power

Authors' addresses: Joseph Raskind, jraskin3@binghamton.edu; Timur Babakol, tbabako1@binghamton.edu; Khaled Mahmous, kmahmou1@binghamton.edu; Yu David Liu, davidl@binghamton.edu.

ACM 2475-1421/2024/6-ART172
https://doi.org/10.1145/3656402

Table 1. Power-Tracking Approaches

Approach	Type	Deployment	Explainability	Security Implications
meter-based	measurement	peripherals needed	no	physical access
RAPL-based	measurement	specific to CPU design	physical	access whole-system info
HPC-based	modeling	friendly	physical	access whole-system info
VESTA	modeling	friendly	(more) logical	access per-application info

modeling, a language runtime-level model has two main advantages: reduced security concerns and a higher level of explainability. Runtime-level events are produced in the scope of the application, as opposed to system-wide information such as HPCs. Requiring access to system-wide information (for power modeling) has its own security implications [14, 25, 31]. Furthermore, language runtime events—coming from a higher level of the computing stack—provide a more logical cause-effect understanding of how an application's design and execution impact power consumption.

Concretely, we introduce Vesta [1], a novel power modeling system that bases its predictive abilities on *language runtime events* in the Java Virtual Machine (JVM). Vesta must address several design challenges. Unique to language runtime events is that they are routinely *split-phase*: when a `long` field of an object is accessed, the JVM does not produce *one* event but *two*: an event `GetLongField__entry` that indicates the access has begun, and another event `GetLongField__return` that indicates the access has completed. In contrast, HPC events are generally *ephemeral*: a cache miss event is produced when it is happening *now*. For Vesta, addressing split-phase events for power modeling is the rule not the exception. In addition, the design of Vesta must address the *diversity* of JVM-traceable events—in the hundreds—and rein in on the classic challenges of reducing *overhead* and improving *precision*.

We use Vesta to model the power consumption of 37 real-world applications running on the OpenJDK. Results show that the power model built by Vesta is capable of predicting their energy consumption with a mean absolute percentage error (MAPE) of 1.56% while incurring small overhead. This is consistent with state-of-the-art HPC-based power modeling where the reported error is generally 3-10% [5, 7, 26, 36, 52]. For experimental comparison, we also (re-)implemented the HPC-based power modeling approach and ran it over the same applications, with a MAPE consistent with their reports.

To the best of our knowledge, Vesta is the first system to use language runtime events—JVM events in our case—for predicting power consumption. The contributions of this paper are:

- a methodology that uses *language runtime events* to build power models;
- a design that *systematically* and *automatically* selects JVM events for power modeling from the complete set of User Statically Defined Tracepoint (USDT) probes [40], and addresses the split-phasedness of JVM events;
- a system that predicts energy consumption with high accuracy and low overhead, and a *decision tree*-based model for explaining the impact of JVM events on energy prediction.

2 MOTIVATIONS

In this section, we motivate the design of Vesta by answering two questions: how Vesta differs from existing approaches, and what challenges an approach such as Vesta must address. From now on, we will use term *runtimes* (as in "runtime systems") to refer to language runtimes.

[1] Vesta is a goddess in Roman mythology. According to Ovid, Vesta derives from Latin *vi stando*, or "standing by power." In our context, Vesta stands for Virtual Energy System for Tracking and Analysis.

Table 2. JVM Event Examples (A full list of OpenJDK-traceable events can be found here [40]).

Event Name	Description
`GetLongField__entry`, `GetLongField__return`	return the `long` field value of an object
`GetMethodID__entry`, `GetMethodID__return`	return method ID
`gc__begin`, `gc__end`	start system-wide garbage collection
`GetObjectClass__entry`, `GetObjectClass__return`	return the class of an object
`compiled__method__load`	JIT-compile a method
`Throw__entry`, `Throw__return`	throw an exception
`NewStringUTF__entry`, `NewStringUTF__return`	construct a `String` object from an character array in modified UTF-8 encoding
`safepoint__begin`, `safepoint__end`	reach a "safepoint" for state examination, e.g., garbage collection
`thread__sleep__begin`, `thread__sleep__end`	invoke a thread `Thread.sleep()`
`vmops__begin`, `vmops__end`	call a JVM bookkeeping operation

2.1 Tracking Power across the Systems Stack

From an end-user perspective, power can be tracked either through *measurement* or *modeling*. A summary of these approaches can be found in Table 1.

Power can be measured either through a physical meter, or through consulting power-reporting architecture features [13]. Measurement-based approaches are straightforward to use for the end user, but they come with some limitations. First, they are subject to *deployment availability*: the deployment site of the application must be either equipped with a meter, or built with architectures that support energy readings, such as Intel's RAPL [13] (see §6). Second, measurement approaches offer little *explainability*: the readings do not explain *how* or *why* energy is consumed.

Modeling-based approaches are more friendly for deployment. HPC-based power modeling approaches share one common insight: power consumption is the effect of hardware use, and hence, power can be modeled by tracking how intensely each architecture component is used, as indicated by HPCs such as cache miss rates. These approaches provide insights on *physical* explainability: the weights associated with each HPC in the model can identify hardware components that play more critical roles in power consumption. The most successful use of HPC-based power models is perhaps power simulation [5, 7, 26, 36, 52] in cycle-accurate simulators.

When HPC-based approaches are used for workload power prediction, one drawback is that HPCs are *system-wide* information whose access has security implications [14]. Under the threat model that the underlying OS may not fully trust the application running on top, giving away system-wide HPC information to applications is a violation of Principle of Least Privilege, and as a result, HPC-based power modeling is best suited for kernel-space whole-system power modeling. This requirement may limit their applicable use scenarios (see § 3.5). In addition, HPC-based power modeling assumes a hardware-centric view for power modeling: its power prediction is based on the hardware states (e.g., cache or TLB)—blind to the software eco-system running on top of the hardware—hence offering little insight on application-level explainability.

2.2 Challenges with Runtime-Level Power Modeling

In contrast, Vesta is a runtime-level approach to power modeling. While power modeling at this layer comes with unique benefits (see Table 1), constructing a power model on top of runtime

events is challenging. While some challenges are common for all power modeling approaches (e.g., overhead and precision), our approach calls for distinct solutions.

2.2.1 Challenge I: Diversity and Dimensionality. The first step of building a power model is to determine what factors may have impact on power consumption. For HPC-based approaches, this is straightforward. Given an architecture, the number of HPC events that can be tracked is relatively small. For example, there are 60 HPCs for the Intel Xeon E5-3630 v4, the platform we run our experiments on. Starting from a relatively small set, one can rely significantly on *domain knowledge* while constructing HPC-based power models. For example, to model the power consumption of a TLB, there are only a handful of HPCs related to TLB behavior, and their impact on power consumption can often be analytically derived *a priori* [26]. As a result, a manual selection of HPCs relying on domain knowledge is not only sensible, but also effective.

In contrast, manual selection based on domain knowledge is unlikely to work for runtime events. First, runtime events are much more diverse. This is particularly true for managed languages—the focus of Vesta—where the runtime events not only come from the application logic, but also come from the virtual machine maintenance, such as JIT and GC. For instance, OpenJDK comes with 520 traceable USDT probes, i.e., the candidate events. The events are usually diverse: in Table 2, we show a small subset of events. Handpicking a subset of JVM events does not scale. Second, deriving analytical power models *a priori* is beyond the skills of domain experts. For example, while JVM experts can confirm that GC has high impact on energy consumption [24, 47], it remains too challenging to manually derive a mathematical model *a priori* that connects GC events with power.

2.2.2 Challenge II: Overhead. Given that manual selection of events is impractical, a naive solution would be to *track'em all*: building a power model based on *all* events and let the (automated) statistical analysis handle the rest. However, this approach comes with a prohibitive cost. While overhead is inherent for building power models, the problem is amplified at the JVM level where hundreds of events could potentially be tracked. Figure 1 shows the trend of the average execution time overhead when a random subset of JVM events are monitored together.

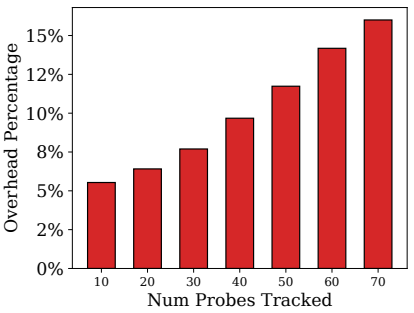


Fig. 1. Wall-Clock Execution Time Overhead (The X-axis shows the number of JVM events monitored at the same time. The events are randomly selected. The Y-axis shows the average execution time across all benchmarks monitored in experimental settings described in § 5.1.)

Overhead is a significant design concern for building power models for two reasons. First, it is undesirable for a monitored application to experience significant performance degradation from the perspective of end-user quality of service. Second, significant overhead is a symptom of perturbation of the original (i.e., unmonitored) application behavior. Indeed, when overhead reaches an extent—2x for example— it becomes a principled concern: the power model no longer characterizes the behavior of the application, but the event tracking logic itself.

2.2.3 Challenge III: Split-Phase Events. As we discussed in § 1, JVM events are often split-phase, in pairs with an entry event (signifying the beginning of an operation) and an exit event (signifying the end of an operation). From now on, we call a JVM event that is not split-phase an *ephemeral* event. In Table 2, all example events except one are split-phase. Indeed, this composition is representative: Split-phase events comprise 94% of all USDT probes that can be traced as JVM events. The dominating presence of split-phase events—as opposed to their non-presence in HPCs—reveals a difference

in software and hardware: whereas most hardware events can be viewed "atomically"—e.g., a cache miss happens at a *snapshot* in time—a software event generally lasts for a *duration* of time.

Split-phase events introduce a unique challenge for runtime-level power modeling. Between `GetLongField__entry` and `GetLongField__return`, the application is in the state of accessing a `long` field. Intuitively, it is the duration when the application stays *within* this state that a stable level of power should be correlated to.

Pairing split-phase events *per se* is not hard: the names of the events are descriptive on pairing. The question is how to *monitor* them. Naively, one may imagine a monitoring algorithm that tracks whether the application is in the state of `GetLongField` or not. Unfortunately, state tracking is *non-binary* for real-world *multi-threaded* applications. Multiple threads could trigger `GetLongField__entry` and `GetLongField__return` concurrently, and it is possible that a program is in a state where five `GetLongField__entry` events have been issued, but no corresponding `GetLongField__return` event is issued. This five-event state should be treated differently from one where only one `GetLongField__entry` is issued. In other words, this quantitative information characterizes how *intensive* memory access is happening, which should be used for power modeling.

When no confusion can arise, we refer to a split-phase event X when there are a pair of events X`__entry` and X`__return` that can be monitored by the JVM.

2.2.4 Challenge IV: Precision. Prior HPC-based power modeling techniques predominantly use linear regression (LR) to predict power consumption [7, 26, 52]. LR is suitable in their approaches because its use is aligned with our intuition on how hardware is used: the power consumption of the entire system is the sum of the consumption from individual hardware components. If a cache consumes 3W and a TLB consumes 4W, the combined power consumption of the cache and the TLB is 7W. In other words, hardware power consumption is *additive*. The impact of runtime events on power consumption is however more complex. The events collectively influence the power state of the underlying system, but does additivity hold?

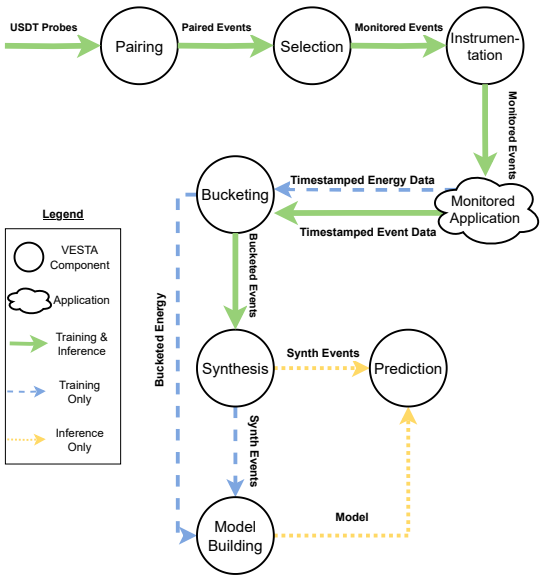


Fig. 2. The Design of VESTA (For training, follow solid green arrows and dashed blue arrows. For inference, follow solid green arrows and dotted yellow arrows.)

3 VESTA DESIGN

3.1 Overview

Fig. 2 shows the overall design of VESTA. As a power modeling tool, VESTA operates in two (standard) modes: training and inference. During training, VESTA can be viewed as a runtime monitor that tracks two pieces of information: the occurrence of runtime events, and the power/energy

consumption of the system. The output of training is a power model, i.e., a function $\mathcal{P}(\mathbb{INT}) \rightharpoonup \mathbb{REAL}$ that takes in the occurrence of runtime events and computes a power consumption value. During inference, Vesta only monitors the occurrence of runtime events, applied to the power model for the predicted power consumption.

Event Domain. Given the complex runtime behavior to capture, the *domain* of runtime events is important for power modeling. Our domain of choice is USDT probes, motivated by several considerations. First, the 520 USDT probes available for JVM monitoring cover a broad spectrum of behavior in managed language runtimes, from object-oriented (OO) semantic features (e.g., heap management, method calls, class loading), non-OO features (e.g., primitive data access, exception handling, JNI), to VM services (e.g., JIT, GC, thread management), to VM metadata management (e.g., VM operations, `safepoint` management). Given that a higher dimensionality is innate with the runtime-based approach (Challenge I), USDT probes provide a comprehensive base set of candidate events for Vesta to sift through. Second, the interface of USDT probe tracing is (largely) language-agnostic. While we currently focus on JVM, the support of USDT probes for other language runtimes is helpful for porting the idea of Vesta in the future. Third, USDT probe tracing has native support on most Linux distributions, facilitating the adoption of Vesta.

Vesta Workflow. During training, Vesta first takes all available USDT probes amenable to the JVM and **pairs** them into split-phase events when possible. This is a simple process where a pair of USDT probes with `__entry` and `__return` suffixes are grouped together; for subsequent steps of the workflow, whenever a split-phase event is selected to be monitored, its pair of USDT probes are both monitored. Vesta **selects** runtime events so that those whose monitoring incurs a large overhead are removed from the consideration of power modeling. The remaining events are monitored through **instrumentation** to the monitored application. For each monitored application, its execution produces a trace of energy data and a trace of event data, both time-stamped. To build a power model, we **bucket** them into fixed-size time intervals, i.e., grouping all events that happen within the same time interval together. In other words, our **model building** is based on a data set where each *time interval* is a unit: we correlate the events that happen in the time interval and the power consumption of the time interval. For now, let us focus on two aspects of Vesta's design: how to reduce overhead and how to handle split-phase events, in the next two subsections.

3.2 Event Selection

Due to Challenge II, it is impossible to track all USDT probes available to the JVM for power modeling without invoking an unacceptable overhead. Additionally, the tools available ready-at-hand to track USDT probes have a distinct upper limit of probes one can track during a given application run. In Vesta, we define a percentage threshold T, and classify all post-pairing events into three categories: *under-threshold*, *over-threshold*, and *rare*. *Under-threshold* events incur an execution time overhead of less than T for *all* benchmarks we build our model with. An event is considered *over-threshold* if *any* benchmark incurs an execution time overhead greater than T. *Rare* events are infrequently encountered, defined as not occurring in any benchmark. Only under-threshold events participate in the building of a power model.

Our universally quantified requirement of thresholding reflects our performance-biased philosophy that "no workload should be left behind": we should *not* choose events that can improve the (average) precision of power modeling *at the sacrifice* of drastic performance degradation of some workloads. We believe that each application in a benchmark suite reflects a unique type of workload, so *all* must concur that the overhead is acceptable before an event is chosen. Our design decision of removing rare events is driven by the fact that monitoring such events is analogous to mitigating the long-tail *at the sacrifice of* overall performance.

3.3 Split-Phase Event Synthesis

As Challenge III requires, VESTA must account for the stateful nature of the events: they are mostly split-phase in the JVM. In addition, we also mentioned that the support for multi-threading concurrent applications dictates that the occurrence in a time interval is not binary.

VESTA introduces *event synthesis* to maintain the occurrences of a split-phase event. Intuitively, we pair up entry and exit type events so that the time intervals in between are viewed as when the (synthesized) event is happening. The events are recorded *globally*, i.e., VESTA counts the occurrence of events in a time interval from *all* threads, not in a *per-thread* manner. This is necessary because modern hardware does not have per-core power domains. In other words, we can only obtain a power reading from all cores residing on the same socket, not individually. For that reason, per-thread event readings would not be useful in power model building.

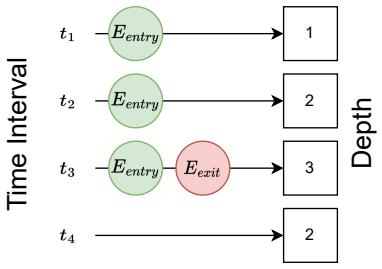


Fig. 3. An Example of Split-Phase Event Synthesis (t_1, t_2, t_3, and, t_4 refer to four distinct, temporally adjacent time intervals, elapsing from t_1 to t_4. For event E, green circles represent its `entry` probes and red circles represent its `exit` probes. The boxes on the right refer to the depth for event `E` at the time interval after synthesis.)

Specifically, recall that we use a split-event X to refer to a pair of USDT probes (i.e., X`__entry` and X`__exit`.) From the first time interval (indexed by 1) of the execution sequence onward, VESTA progressively—i.e., interval by interval—maintains an *accumulated imbalance score* (AIS) for each split-event, defined as:

$$AIS^n = AIS^{n-1} + N^n - X^n \qquad \text{and} \qquad AIS^0 = 0$$

where AIS^k is the AIS for time interval k where $k \geq 0$, N^k is the number of `entry`-suffixed events encountered in interval k, and X^k is the number of `exit`-suffixed events encountered in interval k. Intuitively, *AIS* tracks the number of events started but not yet completed at the end of each time interval. We further define the *depth* of an event at time interval k, denoted as D^k where $k > 0$, as:

$$D^n = AIS^n + X^n$$

D^k captures event *intensity*, i.e., the *maximum* number of events that has started but not completed during the time interval. From an implementation perspective, depth is used by VESTA for building and using our power model (while *AIS* is only a "conceptual" metric to help readers understand the definition of the depth). Depth computation is efficient: it is a linear scan across time intervals.

We now use an example to demonstrate the bookkeeping of this value. Fig. 3 describes a scenario for event `E`. At t_1, the `entry` probe for `E` is encountered, thus increasing the depth of `E` from 0 to 1. At t_2, another *entry* probe for `E` is recorded and, since no `exit` probe has fired, the depth is incremented once more. At t_3, we encounter yet another `entry` probe, but now accompanied by an `exit` probe; the `exit` probe allows VESTA to reduce the depth as we have "exited" the original `entry`. At t_4, depth reduces to 2 with the encounter of one `exit` probe. The depth of an event is related to the maximum occurrence of an unbalanced `entry` probe during the time interval, this is why the depth is 3 at t_3 despite the presence of an `exit` probe while $AIS_3 = 2$.

3.4 Model Building

We attempted a variety of modeling techniques, ranging from LR, decision trees, and neural networks. As it turns out, both LR and neural networks produced suboptimal results. The model choice of VESTA is `XGBoost`, a decision tree model based on the idea of gradient boosting [12].

3.5 Applicability and Use Scenarios

Language Runtimes. Vesta is implemented over the JVM, with direct beneficiaries being applications written in Java or other JVM-based languages such as Scala. Our benchmarks are Java and Scala applications. The domain of runtime events covers a wide range of behavior of managed runtimes, so we speculate that the high-level wisdom—e.g., what runtime events are important for power modeling—may transcend to other managed language runtimes, such as Javascript, Python, C#, and Go, although the model itself must be rebuilt with the language-specific benchmarks. Thanks to the support of USDT probes for C and C++, interfacing Vesta with unmanaged language runtimes does not alter the high-level design and the workflow we described in Fig. 2.

Power Modeling and Workloads. Research on power modeling is motivated to confirm *feature predictability*, implicitly parameterized by the workloads/applications over which the model is built. In the presence of new workloads, the model in principle needs to be rebuilt. In other words, the real news is not the specific values of model parameters produced by Vesta, but the *confirmation* that a subset of language runtime events *can* predict power. In practice, power models are most successful [7, 26, 36] for predicting the power consumption of *known workloads* but over *unknown traces* (or "known unknowns"). Generally speaking, it is a non-goal to build a power model over `sunflow`, and use it to predict the power consumption of `xalan`.

This latter goal is faced with a largely orthogonal challenge: the coverage and quality of the training data set. In other words, while the trace data from `sunflow` alone cannot build a model to accurately predict `xalan`—which is confirmed by our experiments—one may curate a large set of applications that hopefully capture (empirically) every form of workload, and the model training over their traces can predict the power behavior of `xalan`. Intuitively this form of "unknown unknowns" prediction—predicting the power consumption of *unknown workloads* over *unknown traces*—can be viewed as a special form of our "known unknowns" prediction when the number of diverse training applications reaches infinity. We revisit this potential in § 7. The role of Vesta in this potential future direction is a confirmation of predictability: without Vesta, this latter pursuit would be a blind effort solely by increasing the number of training data.

Intended Use Scenarios. As an end-user tool, Vesta is intended for server-type environments (e.g., cloud providers), useful at least in two scenarios:

- *Servers with large power footprints and long longevity.* Energy accounting for these systems is critical both because of their significant instantaneous power, and of their large (accumulative) energy consumption. To apply Vesta, the model is initially trained on the server, and retrained when system configuration changes or when a new workload emerges. In this latter scenario, only the data for that new workload needs to be collected. Vesta training after data collection is efficient: the time of building a model is under a minute for all experiments described in this paper (§ 5). This work flow is also in sync with our discussion earlier on unknown workloads. As time goes on, when the applications used for training reaches a diverse large set, it *de facto* becomes an "unknown unknowns" power model.
- *Service providers and clients in need of explainable and auditable energy consumption.* For a cloud provider that offers energy-based pricing models, individual clients are charged based on the energy consumption of their payload applications. Without Vesta, the only possible approach would be to have the server provider measure the energy consumption (via RAPL or meters) and communicate such information to the individual cloud client. Vesta however offers a form of audit between the cloud provider and the client: the client can verify that—through the ebbs and flows of language runtime events—the energy consumption claimed by the cloud provider indeed matches her own estimate (or not). Furthermore, explainability

entails a better understanding on what exactly she pays for, in the similar vein as why one prefers itemized utility bills.

Beyond tool building, VESTA plays a fundamental role in revealing the deep connection between JVM events and power consumption. Not only confirming this connection, VESTA shows that the connection is so strong that the latter can be quantitatively derived from the former.

4 VESTA IMPLEMENTATION

USDT Probe Tracing. USDT tracing through instrumentation is supported by BPF Compiler Collection (BCC), a toolkit on Linux. When provided with a list of probes to trace, BCC automatically instruments the application with trace points. The resulting event trace, where the occurrence of each event is timestamped, is kept in a perf buffer, which is in turn read by VESTA. We found the default BCC perf (ring) buffer size of 8 pages to be insufficient (see § 5.7), resulting in many losses in the event logging. We set the size to 2048 pages.

Power/Energy Tracing. During training, VESTA periodically samples the RAPL interface of our Intel-based platform for obtaining the energy consumption of the time interval through a tool called jRAPL [32], which provides a convenient interface for Java-RAPL interaction. Power consumption is calculated by dividing it with the length of the interval. The energy readings consist of energy consumption of (i) all cores of all sockets; (ii) all uncore components (caches, etc); (iii) memory controllers. To generate the power trace, each power sample is also timestamped.

We rely on C's `clock_gettime()` function for retrieving timestamps, with `CLOCK_MONOTONIC` as the argument. This function allows us to retrieve a monotonically increasing timestamp with nanosecond resolution. We decided not to use Java's `nanoTime()` function as its documentation states "no guarantees are made except that the resolution is at least as good as that of `currentTimeMillis()`."[2] For event traces, BCC already reports with nanosecond precision.

Benchmark Selection. All experiments for VESTA were performed using 37 state-of-the-art applications from two benchmark suites: DaCapo [8] and Renaissance [44]. All benchmarks are multi-threaded. Both benchmark suites provide their user the ability to create Java callback plugins which we used to collect runtime energy data and information about each run. We created two sets of callback plugins: one for event selection (where one single event is instrumented) and the other for post-selection data collection (where multiple events are instrumented).

Model Building and Prediction. The alignment of the event trace and the power trace is conducted after the execution is completed. We first bucket event/power data into buckets, i.e., fixed-sized time intervals. When an event does not occur, we use -1 as its depth. The readings during the benchmark harness execution are excluded. The bucket size is 1 second, identical to existing HPC-based approaches [7, 26, 36]. This is also in sync with our use scenarios (§ 3.5): the long-running applications which do not complete in seconds or sub-seconds. We run each benchmark for 256 iterations in one hot JVM run, and discard the first 5 iterations to mitigate the effect of warmup. The rest of the data are used for training and inference, as we describe next.

Table 3. Examples of $k \times 2$ Cross-Validation. (Let A, B, and C be benchmarks and A_i, B_j, and C_k be distinct time intervals where $i \in [1..3], j \in [1..4]$, and $k \in [1..5]$. Three experiment examples on data splits are shown.)

Train	Test
$A_1, B_4, B_2, C_3, C_1, C_2$	$A_2, A_3, B_1, B_3, B_5, C_4,$
$C_2, A_3, A_2, A_1, C_4, C_1$	$C_3, B_4, B_1, B_2, B_5, B_3$
$B_2, B_5, C_1, C_3, A_2, B_1$	$C_4, A_3, C_2, A_1, B_3, B_4$

[2]From the Java System API.

Table 4. Post-Selection Events (Events with * mean they are empheral events, the rest are split-phase events.)

Event Category	Events
Method	`CallObjectMethod`, `CallVoidMethod`, `GetMethodID`
JIT	`compiled__method__load*`, `compiled__method__unload*`, `method__compile`
Type & Metadata Management	`IsInstanceOf`, `GetObjectClass`, `GetEnv`, `vmops`, `safepoint`
Memory Management (GC)	`gc`
Memory Management (Primitive)	`NewString`, `NewStringUTF`, `GetStringLength`
Memory Management (Array)	`GetByteArrayElements`, `GetObjectArrayElement`, `ReleaseIntArrayElements`, `SetByteArrayRegion`
Memory Management (Object)	`GetLongField`, `SetIntField`
Exception Handling	`Throw`
Concurrency	`thread__park`, `thread__sleep`

To perform training and inference, we adopt the approach taken by McCullough et al. [36] in their HPC-based power prediction: we utilize a $k \times 2$ cross-validation. The ordering of the data items (i.e., the per-interval occurrences of runtime events and energy consumption values), taken from all benchmarks, is randomized and then split in half—one half (which may come from intervals of various benchmarks) is used for training, and the other half for testing. Table 3 visualizes this process. Our cross validation is repeated 10 times and the mean, and standard deviation for each benchmark, can be found in Fig 5. Note that a methodology that would split over benchmarks for training and testing is unsound (unless one has thousands of benchmarks), as we described in § 3.5.

Implementation Languages. The runtime monitoring core of Vesta is written in Java, with C (JNI) code for low-level operations such as timestamping and energy sampling. The code base also consists of Python and bash scripts for model building and setting up experiments.

5 VESTA EVALUATION

5.1 Experimental Settings

We evaluate Vesta on a dual socket Intel Xeon E5-3630 v4 2.20GHz CPU server with 20 cores per socket (40 cores in total) and 64GB DDR4 RAM. The machine runs Debian 5.17.11-1, Linux kernel 5.17.0-3-amd64. All experiments were run with OpenJDK 19 with the `ExtendedDTraceProbes` flag set. We used the latest builds of both DaCapo and Renaissance, versions evaluation-git+309e1fa and 0.14.1 respectively. The default power governor in Linux is used, where Dynamic Voltage and Frequency Scaling (DVFS) [10, 22] is enabled.

5.2 Event Selection

Vesta pairs all available USDT probes and then selects the events based on the methodology described in §3.2, where the (relative) threshold is set as $T = 20\%$. Fig. 4 shows the results of this selection.

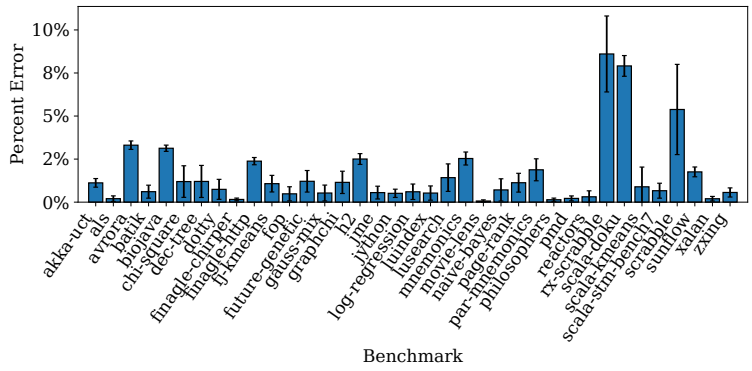


Fig. 5. Vesta Precision (The Y-axis shows the MAPE of energy prediction normalized against the actual consumption. Average ϵ = 1.56%.)

Table 4 enumerates all events that remain after the selection process. As seen here, the 24 events ultimately selected for power model building remain diverse in scope.

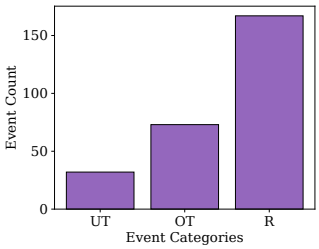


Fig. 4. Event Demographics (UT is *under-threshold*, OT is *over-threshold*, and R is *rare*.)

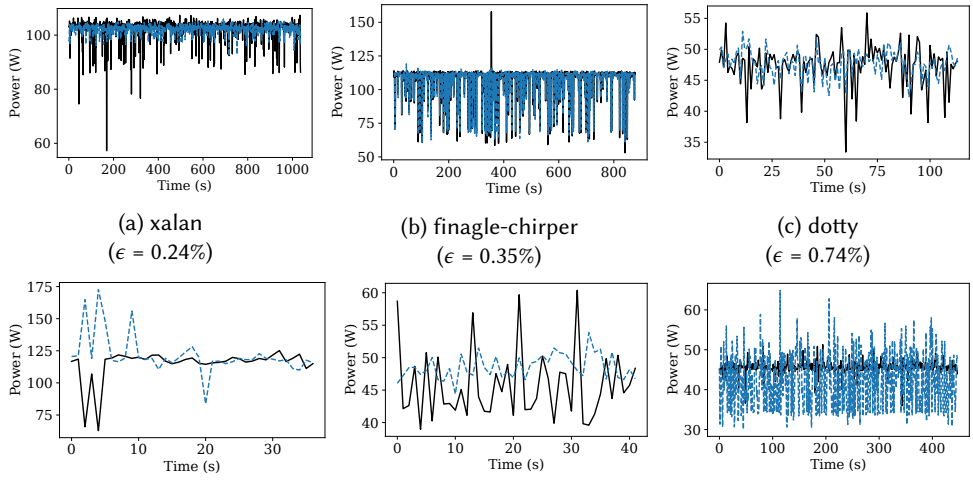
5.3 Prediction

Fig. 5 shows the precision of Vesta, which relies on the `XGBoost`-based decision tree model. With the power model produced by Vesta, we are able to predict the energy consumption of all benchmarks with a MAPE of 1.56%. The worst prediction among all 37 benchmarks remains under 10%, showing that Vesta is capable of describing energy consumption under diverse workloads.

Fig. 6 shows Vesta managed to predict 6 separate benchmarks. First, let us look at the highly accurate top row on display (Figures 6a, 6b, 6c): where *xalan* transforms XML documents into HTML, *finagle-chirper* simulates a microblogging service using Twitter Finagle, and *dotty* runs the Dotty compiler on a set of source code files. Despite the vast difference in workloads represented by these benchmarks, Vesta was able to predict the power usage of each with a MAPE of under 2%. We believe that some of the monitored events in these experiments are closely linked to the fluctuation in power consumption, and Vesta is able to accurately reproduce the same power fluctuations in its prediction.

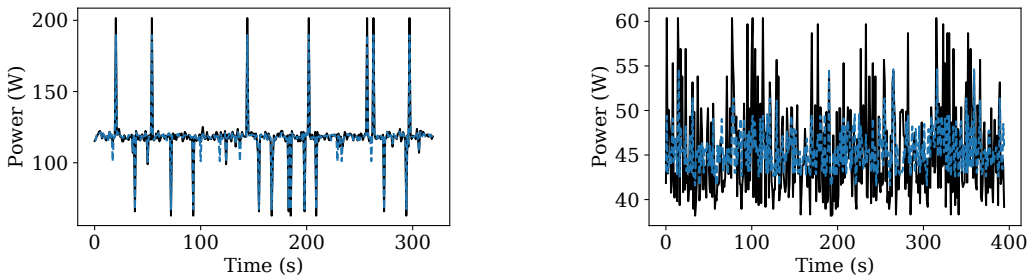
On the other end of the precision spectrum, Figures 6e, 6d, and 6f shows the three "worst performers": *rx-scrabble*, *scrabble*, and *scala-doku*. All from Renaissance, *rx-scrabble* and *scrabble* solve Scrabble puzzles using Rx streams and JDK streams, respectively, and *scala-doku* solves Soduku puzzles using Scala collections. These three benchmarks are the only ones among the 37 that result in prediction errors over 5%. We can break them into two separate groups: *insufficient data points*, and *ineffective events*. Benchmarks *rx-scrabble* and *scrabble* fall into the former camp whereas *scala-doku* falls into the latter.

Both *rx-scrabble* and *scrabble* are among the top-3 shortest benchmarks of the entire 37 benchmarks: each runs around a third of a second per iteration. Relative to other benchmarks, the execution time imbalance implies that the workloads represented by *rx-scrabble*, *scrabble* are underrepresented by our model, trivializing the power behavior exhibited by the "lightning fast" duo *scrabble* and *rx-scrabble*. In order to test our hypothesis we artificially expanded the data



(a) xalan (ϵ = 0.24%)
(b) finagle-chirper (ϵ = 0.35%)
(c) dotty (ϵ = 0.74%)
(d) scrabble (ϵ = 5.84%)
(e) rx-scrabble (ϵ = 6.15%)
(f) scala-doku (ϵ = 7.73%)

Fig. 6. Predicted Power vs Measured Power for Representative Benchmarks (The X-axis represents the elapsed time. The Y-axis shows power consumption being predicted in Watts. The 3 benchmarks in the first row are the best-performing predictions and the 3 benchmarks in second row are worst-peforming predictions. The dashed blue line is the predicted power, and the solid black line is the measured power.)



(a) scrabble – alternate prediction (ϵ = 0.31%)
(b) rx-scrabble – alternate prediction (ϵ = 0.13%)

Fig. 7. Predicted Power vs Measured Power with Increased Data Points (Alternate *scrabble* and *rx-scrabble* predictions that yield a MAPE of 0.31% and 0.13%, respectively when we artificially duplicate the data in Fig. 6 10 times.)

points available to both *scrabble* and *rx-scrabble* by copying our recorded data 10 times over and rerunning Vesta. Fig. 7 shows that adding extra data greatly increased prediction precision. As we envision a production-strength system inspired by Vesta is likely to be trained over more and longer benchmarks, we think the problem exhibited by *scrabble* and *rx-scrabble* is a superficial one.

Unlike *rx-scrabble* and *scrabble*, *scala-doku* is faced with a entirely different challenge: a lack of *effective* events for prediction. For example, even though *scala-doku* has a similar event count and execution time as *par-mnemonics*, *scala-doku* has a MAPE nearly 5 times greater. We believe that

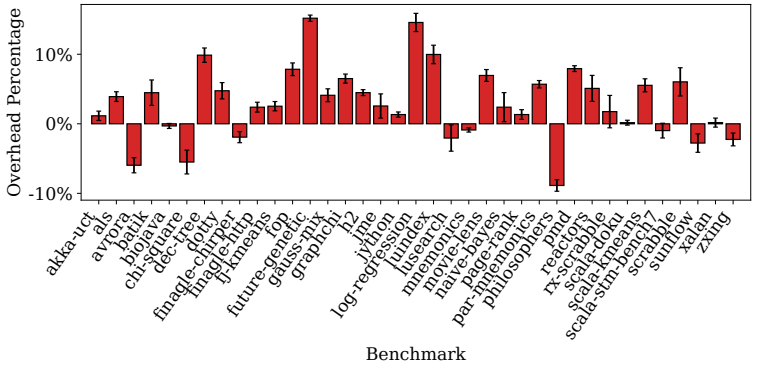


Fig. 8. VESTA Reference Cycle Overhead (The Y-axis shows the number of reference cycles normalized against that of unmonitored runs. Average ϵ = 2.90%.)

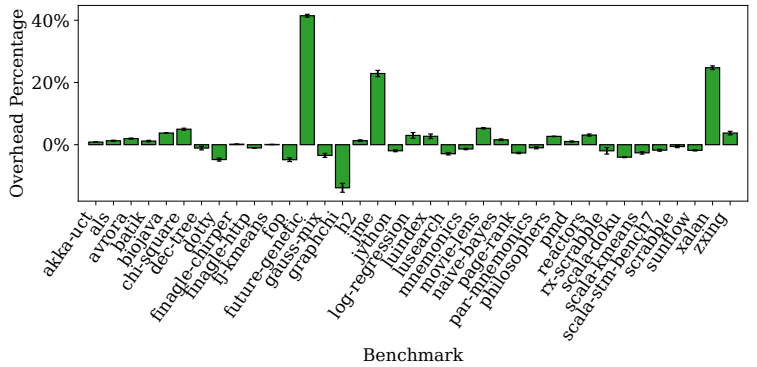


Fig. 9. VESTA Energy Overhead (The Y-axis shows the energy consumption normalized against that of unmonitored runs. Average ϵ = 2.05%.)

the degraded precision results from the fact that the chosen 24 events for power modeling happens to poorly characterize *scala-doku*'s power behavior. It is likely that *scala-doku* is missing a key event that has been filtered out through the selection process. This outlier points to a limitation of our selection approach: our universal quantification of thresholding at its essence is performance-biased (recall § 3.2). In other words, to make sure all 37 applications exhibit reasonable performance, the precision of *scala-doku* has to suffer. With this in mind, it is worth pointing out that an error of 7.73%—the worst of all 37 benchmarks—may still be acceptable for power modeling. For instance, most HPC-based approaches in existing literature (see § 6) report outliers with higher errors.

5.4 Overhead

Figs 8 and 9 show that, on average, VESTA runs with a performance overhead (in reference cycles) and energy overhead, averaging at 2.90% and 2.05% respectively. Compared with Fig 1, these figures highlight the importance of event selection: it is only through the algorithm defined in § 3.2, we can achieve relatively low overhead.

Beyond the general trend, several observations can be made. First, our execution overhead reports all-thread reference cycles, i.e., the overall number of reference cycles from all threads in

the benchmark, not wall-clock time. Relative to the latter, the reference cycle overhead indicates "extra work" due to Vesta event monitoring, regardless of whether such work is on the critical path of a multi-threaded application (all our benchmarks are). As a result, the end-to-end wall-clock overhead of Vesta is generally less (ϵ = 0.90%). Second, a small number of benchmarks report negative performance overheads. We believe this results from the interaction between Vesta monitoring and DVFS. The monitoring activities (of 24 probes) by Vesta may have intensified the CPU activities, driving their host cores to a higher power state, i.e., operating at a higher CPU frequency. As a result, a program may run faster. This phenomenon was reported in energy profiler design before (e.g., [2]). Note that reference cycles—as opposed to CPU cycles—already take clock speed into account. Third, the trend for reference cycle overhead and energy overhead do not always correspond. According to physics, energy is the multiplication of power and time. The metric of reference cycle count—despite the fact that it may not always directly translate to end-to-end wall-clock time—is a time-based metric after all. For example, if we imagine two programs with an identical execution time but one consumes twice the power of the other, then the aforementioned program will also consume twice the energy.

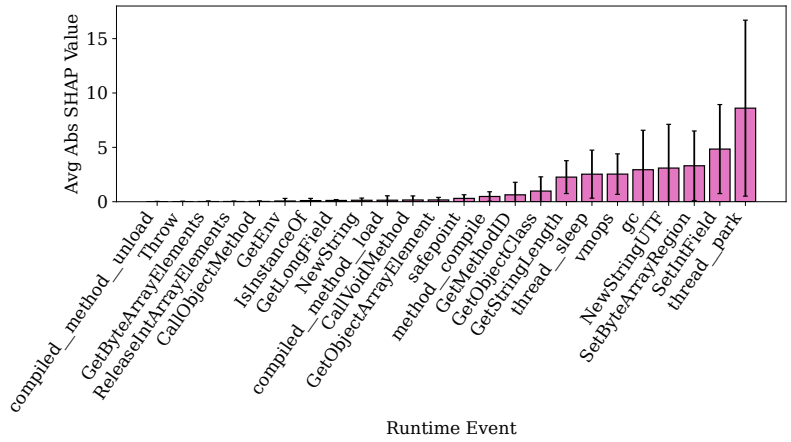
5.5 Event Importance and Explainability

To gain more insight on the behavior of events, we examine the *feature importance* of our model. Our metric of choice is SHAP (SHapley Additive exPlanations) values [33, 34], a high-level metric that has rapidly gained popularity in the field of ML explanability. SHAP is based on cooperative game theory by Shapley [46]. A positive/negative SHAP value for a feature means that the presence of the feature influenced an increase/decrease in the outcome. The higher the absolute SHAP value is, the more influence the given feature has over the outcome of a prediction. In the supplementary material, we also include results based on lower-level metrics such as gain and frequency for decision trees, with similar overall trends as SHAP.

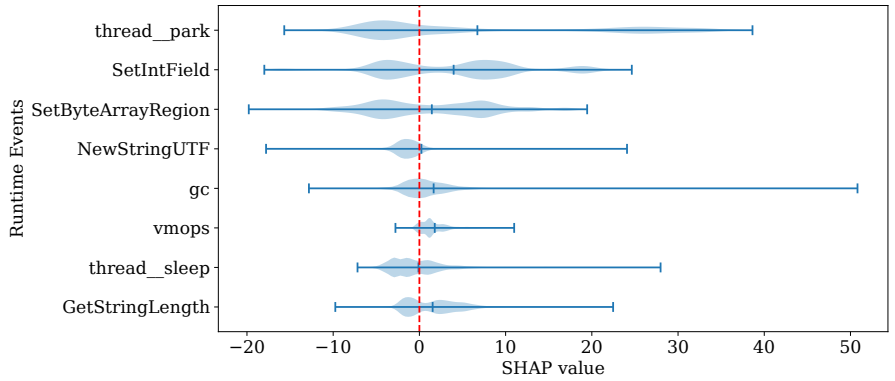
Fig. 10a shows the (ranked) average absolute SHAP values for each feature. Fig. 10b provides a deeper look at the top eight most important features by showing how the SHAP values are distributed. Before we delve into the details, observe that individual SHAP values per time interval may vary greatly, so the violin graph has a long tail. As a result, the mean (the middle vertical line) often does not coincide at where the most data points are. This should be expected, because during any time interval, many events may co-occur, and even the most important event may only have limited and varying influence on power consumption. The large variance here indeed demonstrates the challenge that Vesta has overcome: despite the highly dynamic nature of the executions where even the most important events have varying influence across time intervals, Vesta is able to make accurate power consumption. Specifically, we make several observations.

First, thread management plays an important role in power consumption. `thread_park` is clearly the most important feature, reflected by the high average absolute SHAP value. Similarly, `thread_sleep` is also a highly ranked event. This outcome is not surprising: thread management has a large impact on system utilization. The impact of thread scheduling on energy consumption is well known in energy-efficient computing [50], including prior empirical studies at the JVM runtime level [43].

Second, memory access is influential on power, with `SetIntField` and `SetByteArrayRegion` being the second and third important events for power modeling respectively. This is aligned with the finding in HPC-based power modeling where cache misses are among the most indicative HPCs for power consumption. To gain a more in-depth understanding, we zero in on the behavior of `SetByteArrayRegion`. Fig. 11a shows the scatter plot on how individual feature observations and their correposnding SHAP values. Fig 11b further correlates the feature observations with cache misses, as tracked by the underlying HPCs. Interestingly, Fig. 11a shows there is a



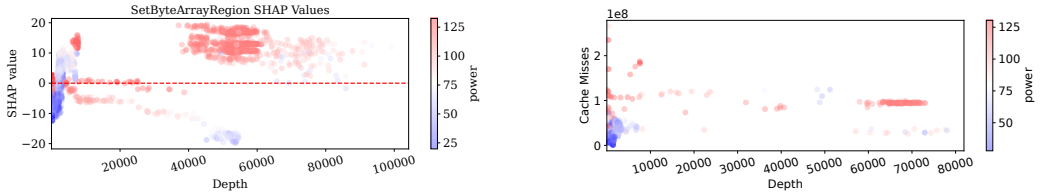
(a) Average Absolute SHAP Values



(b) Events and SHAP Values

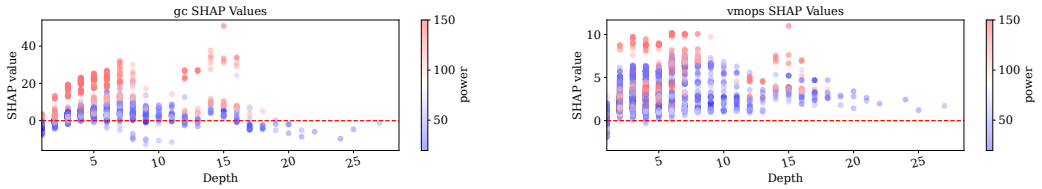
Fig. 10. Vesta Feature Importance. (For the first subfigure, the X-axis refers to the runtime events, and the Y-axis refers to the average absolute SHAP value. The second subfigure is a violin graph where the X-axis refers to the per-interval SHAP value of a given feature, and the Y-axis refers to the number of intervals with that SHAP in violin graph. The leftmost, middle, and rightmost vertical lines in each horizontal bar for each event are the minimal, mean, and maximal SHAP values respectively. Only time intervals that the event occurs, i.e., depth>0, are shown.)

bipartite power behavior. For lower depths, `SetByteArrayRegion` decreases power, whereas for higher depths, it increases power. We think the bipartite behavior is aligned with our intuition. `SetByteArrayRegion` promotes sequential access to the memory, reducing cache misses, and subsequently, power. When the number of `SetByteArrayRegion` increases significantly, different requests of `SetByteArrayRegion` may compete for cache lines, increasing cache misses. Our experiment on the co-occurring cache misses in Fig 11b appears to confirm this intuition. Lower/higher cache misses seem to correspond with lower/higher depths of `SetByteArrayRegion`. Finally, observe that memory "setter" (with prefix `Set-`) events are more important than memory "getter" (with `Get-` prefix) events on power consumption. Whereas both memory reads and writes can lead



(a) SHAP (b) Co-Occurring Cache Misses

Fig. 11. `SetByteArrayRegion` Behavior (Each dot in the figure is a time interval. The X-axis shows the `SetByteArrayRegion` depth. The Y axis of the first subfigure shows the SHAP value, and that of the second subfigure shows the number of cache misses in the same time interval. The colour refers to the observed power in watts.)



(a) gc (b) `vmops`

Fig. 12. VM Internal Events

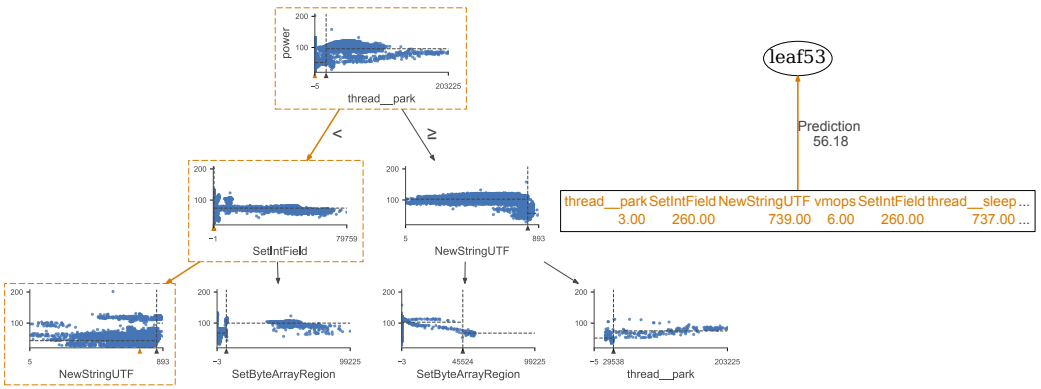


Fig. 13. A Decision Vesta Tree Example (The prediction path is highlighted in orange. Only the first two tree levels are visualized, except node "leaf53" which is the leaf that is ultimately reached if one continues to follow the orange path down the tree. The box associated with each node contains the scatter plot of the named feature for the given test input.)

to cache misses, memory writes may further render a cache incoherent, and subsequently putting cache coherence protocols [49] to work and occasionally triggering write-backs, increasing power.

Third, we were at first surprised that `NewStringUTF` is among the most important events, especially considering the `NewString` event has a relatively low SHAP value. Upon further inspection, we found that the `NewStringUTF` event is specific for UTF-8 encoding, whereas Java is a UTF-16

language. In other words, `NewStringUTF` is used in Java-C++ interactions through Java Native Interface (JNI). Not surprisingly, a significant portion of OpenJDK—especially those for low-level device I/O operations—was written in C++.

Fourth, VM internal events have a non-negligible impact on power consumption. According to Fig. 10, `gc` and `vmops` are ranked 5^{th} and the 6^{th} in SHAP values. A closer look at these two events are shown in Fig. 12. Here, garbage collection tends to increase power consumption, but there are diminishing returns. This can be seen in Fig. 12a where, as depth increases, the SHAP value continues to increase until we go past a depth of fifteen. This means that after an intensity level of the garbage collector is reached, the underlying system is likely already in a high power state, and therefore further increases in GC are unlikley to change power. Similarly, `vmops` also has a tendency in increasing power consumption, but its influence is relatively limited: observe that in Fig. 12b, the majority of SHAP values are below 10, whereas for Fig. 12a, the majority of SHAP values for `gc` are below 40.

A full-fledged account on explaining the power consumption based on JVM events can be provided by the decision tree produced by VESTA, with an example shown in Fig. 13. Important events such as `thread_park` , `NewStringUTF`, `SetIntField`, and `SetByteArrayRegion` all appear close to the root of the decision tree, signifying their importance in decision making. Here, the bottom right is the observation where the depth of each event is defined as a vector. By traveling down the prediction path (illustrated in orange in the figure), the decision tree helps us understand how VESTA eventually reaches the power prediction.

5.6 Alternative ML Models

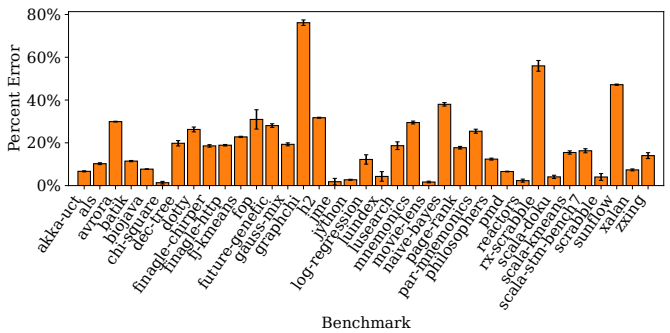


Fig. 14. Linear Regression-Based Power Modeling (The X-axis shows the benchmark. The Y-axis shows the MAPE. The average ϵ = 18.85%.)

VESTA is designed with decision trees (XGBoost) as its core ML model. We arrived at this choice after experimenting with alternative models, all the while trying to address the goal of Challenge IV. We now report the results based on linear regression (LR) and deep neutral networks (DNNs).

Fig. 14 shows LR would introduce high errors (average 18.85%) for the benchmarks we consider. This result highlights the fundamental distinction between HPCs and runtime events. In contrast with HPC approaches, it would not make sense to consider the power impact of runtime events as additive (see Challenge IV). For example, it is indeed true that the `gc` event and the `vmops` event may contribute to power consumption, but their combined impact on power is likely to be more complex than, say, the combined power is 7W if a cache event contributes to 3W and a TLB event contributes to 4W. Fig. 14 confirms the non-additivity of the power impact of runtime events.

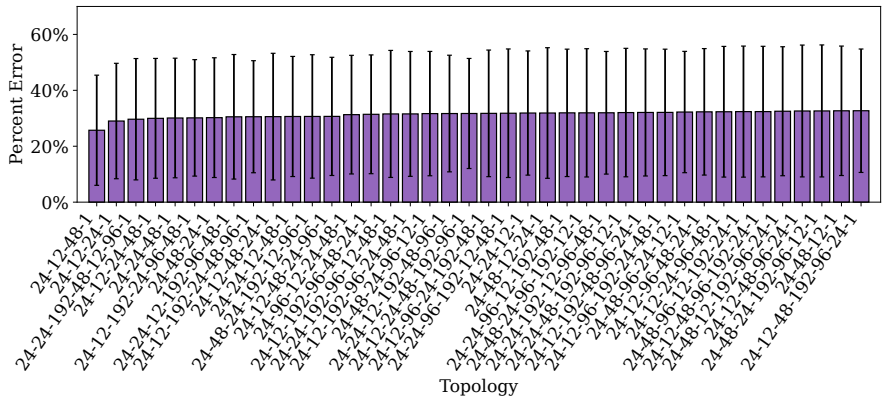


Fig. 15. Neural Network-Based Power Modeling (We present the top-40 DNN configurations with the best prediction accuracy, ordered from the left (the best) to right (the worst). The X-axis shows the topology configuration, where $n_1 - n_2 - \ldots n_k$ means the number of neurons for layer i is n_i where $1 \leq i \leq k$. The Y-axis shows the MAPE across all predictions for all benchmarks. The average of the 40 results ϵ = 31.34%.)

Considering the popularity of DNNs to solve non-linear modeling problems, we also experimented with DNN-based power modeling. We constructed 100 different DNNs all with unique topologies; we present the top-40 performing DNNs in Figure 15. It is discouraging that the prediction errors are not only high, but also appear to be insensitive to the topology configuration. We must be careful to stress that we manually selected the 100 topology configurations, and the results may not be the best possible after *exhaustive* hyperparameter tuning. With the errors of these DNN results stubbornly high and with `XGBoost` already producing competitive results, we are less incentivized to exhaustively explore the DNN space.

The real take-away message here is that Vesta—built with `XGBoost`—does not achieve good performance trivially by plugging runtime events into any model toolkit available. It is important to realize that all reported results in this section are produced over the same set of benchmarks (Dacapo and Renaissance)—indeed, the same data traces—and the same validation methodology ($k \times 2$ cross-validation). In other words, the good performance of Vesta results from its inherent design choices (§ 3), it is not a coincidence of the experimental process.

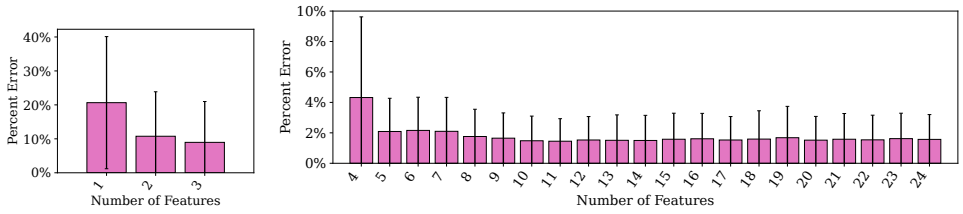
5.7 Alternative Numbers of Events

The (default) Vesta model is built with all 24 events (§ 5.2). Given that these events have different importance (Fig. 10), we now conduct a design space exploration by reducing the number of events (features) for model building. The results are shown in Fig. 16.

Overall, around 10 events have noticeable impact on the accuracy of the model, with the top 3-5 events having significant impacts. This points to a possible customization of Vesta in real-world development, where only a subset of events is used for model building. A word of caution is that the accuracy shown in Fig. 16 is the *average* of all 37 benchmarks, and individual benchmarks may have varying accuracy impacts. In other words, it would be premature to conclude that only the top 10 events shown in Fig. 10 matter categorically.

5.8 Alternative Configurations across the Computing Stack

As another design space exploration, we also conducted an analysis on the accuracy of Vesta with various alternative settings across the computing stack. Concretely, (1) we reconfigure the



(a) Model w/ Top 1-3 Features

(b) Model w/ Top 4-24 Features

Fig. 16. Vesta Prediction Accuracy with Different Numbers of Features (The X-axis shows the number of features — i.e., the number of types of events — used for model building, where a bar labeled with k means building the model only with k number of events with the highest k absolute SHAP values reported in Fig. 10. For example, the first bar in Fig. 16a means building the model only with the `thread__park` event. Due to scale, we divide the results in 2 subfigures. The Y-axis shows the MAPE. The whisker shows the standard deviation computed across the MAPE of 37 benchmarks.)

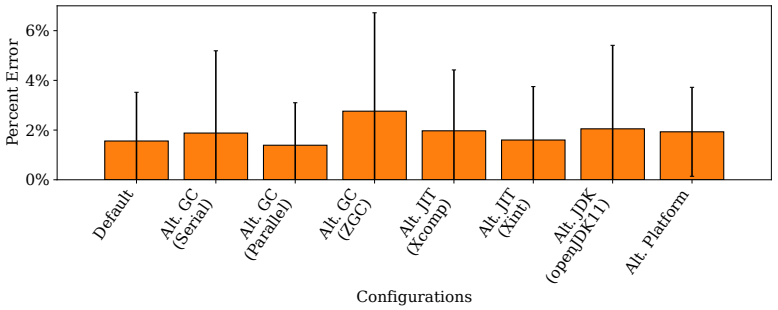


Fig. 17. Vesta Accuracy with Different Configurations. (The X-axis shows different configurations, and Y-axis shows the MAPE of energy consumption based on Vesta prediction. `Default` is the default Vesta configuration where all results are presented hitherto. Labels with prefix `Alt.GC` are 3 namesake GC options. Laberls with prefix `Alt.JIT` are two compilation options. The label with prefix `Alt.OpenJDK` refers to an alternative OpenJDK. `Alt.Platform` refers to a machine with an Intel Xeon Silver 4300 v3 2.30 GHz CPU with 40 cores, Ice Lake micro-architecture, 64GB DDR4 RAM, running Debian 6.1.55. For each alternative configuration, the rest of settings beyond the explicitly stated alternative are identical to the default setting in § 5.1. The whisker shows the standard deviation computed across the MAPE of 37 benchmarks.)

JVM with three alternative GCs (`Serial`, `Parallel`, `ZGC`), whereas the default results presented in earlier sections come from the default GC, `G1`. (2) We run benchmarks with two alternative compilation options (`XComp` and `XInt`), the two ends of spectrum of JIT design, i.e., all-method JIT and no JIT (i.e., interpretation) respectively. (3) we run with an alternative OpenJDK version, v11. (4) we experiemnted with a different machine, with details shown in Fig. 17.

As shown in the figure, Vesta is able to retain comparable accuracy in all alternative settings. Note that the standard deviation shown in the figure is computed across the 37 benchmarks. What it reflects is the diversity of benchmarks: just like our default setting, Vesta can offer better prediction for the vast majority of benchmarks, but there are outliers. For the worst-performing configuration `ZGC` (with error 2.76%), we further present the per-benchmark accuracy result in Fig. 18. With `ZGC`, the worst-performing benchmarks are `rx-scrabble`, `scala-doku`, and `scrabble`. These three

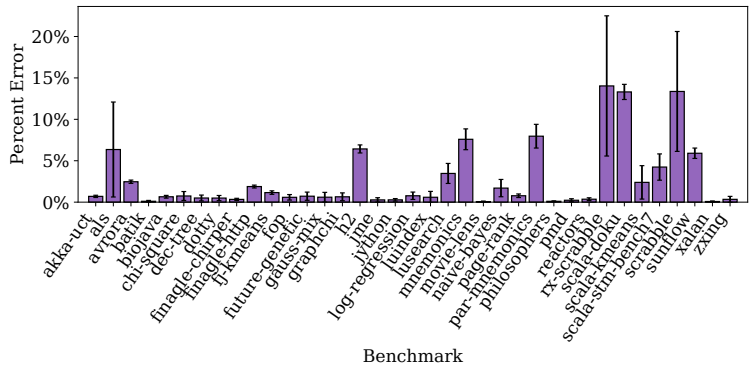


Fig. 18. Vesta Precision using ZGC (The Y-axis shows the MAPE of energy prediction normalized against the actual consumption. Average ϵ = 2.76%.)

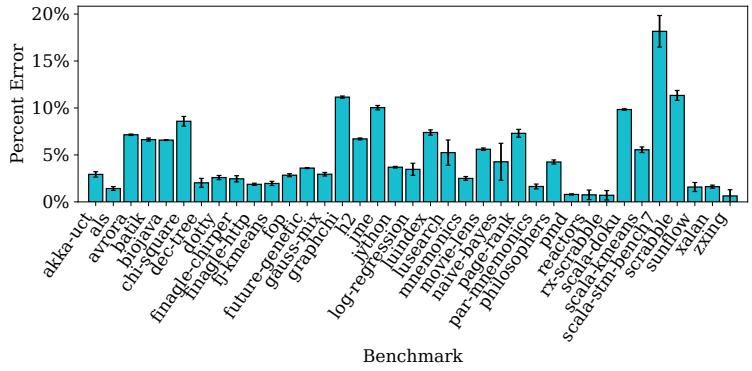


Fig. 19. A Reproduction Study of HPC-Based Power Modeling: Accuracy (The Y-axis shows the MAPE of energy prediction normalized against the actual consumption. Average ϵ = 4.81%)

benchmarks happen to be the same worst-performing benchmarks in our default setting (§ 5.3). Unfortunately for ZGC, the prediction errors for these three benchmarks are larger than the default setting, resulting the average MAPE across benchmarks to increase (as opposed to 1.56% for the default model).

5.9 Other Efforts in Design Space Exploration

The supplementary material includes data on two additional sets of experiments. First, we alternate the bucket size. In summary, the result shows a smaller bucket size (such as 50ms or 500ms) will reduce precision, but still within or around a MAPE of 10%. The more problematic is that smaller bucket sizes will significantly increase the standard deviation of the results. Our choice of 1s bucket size is also consistent with prior work on HPC power modeling [7, 26, 36]. Second, recall that we chose to set the buffer size in our implementation (§ 4) to 2048 pages. The experiments will describe how we decided on this setting.

5.10 A Comparison with HPC-based Models

Existing HPC-based power modeling systems [5, 7, 26, 36, 52] report accuracy (in MAPE) ranging 2-10%. These results provide us an empirical understanding on what the "ballpark" expectations of an effective power model should be. The prediction error of Vesta (1.56%) is on the smaller end of this ballpark. To further gain confidence, we conduct a reproduction study for HPC-based power modeling in our experimental setting. The thorny issue is that none of the prior work contains an exhaustive list of the HPCs used, and due to architectural differences between theirs and ours, a one-on-one mapping is also difficult to establish for those they discuss. Our best effort for approximating known HPC-based power models is to combine the modeling methodology of McCullough et al. [36] with the HPC correlation data of Zamani and Afsahi [52]. In the same methodology as the former, we greedily selected HPCs with the highest power correlation, except that we used the HPC correlation provided by the latter. Following their methodology, we used LR and built a power model that consists of 12 `perf` HPCs, whose names can be found in the supplementary material.

The energy prediction results of this power model are in Fig. 19. The precision of 4.81% confirms the MAPE range specified in the HPC-based systems. The execution time overhead and energy overhead are both <1%, with details reported in the supplementary material.

For readers interested in alternative choices of HPCs, we have included this script in our repository with a brief explanation on customization.

6 RELATED WORK

HPC-based power modeling has a long history. Isci and Martonosi [26] is an early work that shows the feasibility of estimating power at execution time through piece-wise linear combinations of HPC counts. Zamani and Afsahi [52] uses an ARMA (Autoregressive–moving-average) model to estimate power consumption and develop a methodology for ranking the usefulness of HPCs. Bircher and John [7] focuses on how HPCs could be used to predict power consumption of hardware subsystems outside of the microprocessor, such as DRAM and I/O devices. McCullough et al. [36] recreates a number of linear models and demonstrates their relative effectiveness for online modeling. They also studied non-linear models such as Support Vector Regression and Polynomial with Lasso Regression, where results do not show significant improvement. Bertran et al. [5] extends power prediction models with the ability to detect power phases. The relationship between Vesta and HPC-based approaches has been discussed in §3; a performance comparison can be seen in §5.10.

HPC-based power modeling is widely used in cycle-accurate power simulation. For CPU power simulation, examples include Wattch [9], gem5 [6], and McPat [29]. There are also cycle-accurate power models built for GPUs [11, 27, 28]. Power modeling for power simulation does not need to be concerned with overhead: the simulator runs substantially longer than the program it simulates.

There is a small body of prior work that rely on OS events for power modeling. Li and John [30] shows how OS routine invocations can be used to predict overall OS power consumption. Their focus is on modeling the power/energy consumption of the OS principals—e.g., interrupts, inter-process communications, and file system operations—not applications. Pathak et al. [42] developed a power model for Android-based smartphones. Their system models the components of a smartphone, such as WiFi, NIC, SDCard, LCD, camera, GPS, as well as the CPU. For smartphones, their approach is appropriate because the non-CPU components dominate the power consumption, where system calls may be strongly correlated to the use of these non-CPU components. It is however unclear whether their approach can generalize to our setting, a CPU/memory-centric server-class environment.

The discussion of Pathak et al. brings up a fair critique of Vesta: server-class environments also have their additional power-hungry components—such as GPU and NIC—not modeled by Vesta in its current form. As our benchmarks do not significantly interact with these hardware components, they likely incur (near-constant) idle power. In other words, even if we were to attach meters to GPUs and NICs and add their power consumption to our data for model building, the resulting model would be identical modulo a constant.

RAPL [13] allows end users to obtain energy readings through its dedicated registers on some CPUs, such as recent models by Intel and AMD. For some architectures, the energy consumption stored in RAPL registers is also *modeled* through hardware performance counters. By categorizing RAPL as a *measurement* approach in § 2, we emphasize the end-user view. RAPL reports energy data separately for core, uncore and DRAM components, offering a modicum of explainability about these 3 physical components. Accessing RAPL registers requires root access. Recent studies also show [25, 31] that side channels may be formed through the shared RAPL registers, posing security vulnerabilities. We used RAPL during training only, but this is not essential to our design: it can be replaced by any measurement approach.

The goal of energy/power accounting systems is to distribute a global energy/power consumption into software/hardware components, both at the OS level [20, 53] and the application level [1–3]. The latter is also related to energy profiling [16, 18, 41, 48], producing a profile that consists of energy consumption at the granularity of architecture, thread, or software logical components. Power modeling and power accounting are different but complementary approaches.

We borrow the phrase "split-phase" from nesC [19], a sensor network language. The phrase was used in their language to refer to how a traditional synchronous event is split into two asynchronous events: its start and its completion.

7 CONCLUSION

Vesta is a novel power prediction approach where JVM events are used for power modeling. This approach has the benefit of not requiring access to low-level whole-system information, offering logical explainability of application energy behavior, and providing high precision. Vesta is implemented as a lightweight monitor, and the power model it builds is highly precise with small performance and energy overhead.

Now that Vesta has established the power predictability of JVM events, there are a number of opportunities. First, it is interesting to investigate the feasibility of *curating* a set of Java applications that are sufficiently large and representative, so that the prediction of "unknown unknown" workloads (§ 3.5) becomes empirically effective. The answer to this question may also have implications on (power-representative) benchmark suite design. With predictability established, improving prediction through larger training data is a recurring motif in machine learning. Fortunately, there are a large number of Java applications available. Second, our preliminary studies on alternative configurations (§ 5.8) may be significantly expanded, deserving to be an empirical study of its own. As both GC and JIT are active research topics, their possible variations, together with those of the underlying OS/architecture, far exceed what we have experimented. Last but not least, we wish to apply the idea behind Vesta to non-JVM runtimes. Our decision of tracking USDT probes makes porting our implementation to other USDT-supporting languages relatively simple: BPF/BCC already supports USDT tracing for other languages, including unmanaged runtimes such as C and C++. Except for the data collection stage, the rest of Vesta remains the same. Implementability however does not equate effectiveness. Due to the fundamental difference between language runtimes, especially that between managed languages and unmanaged languages, it remains to be seen whether an accurate power model can be built with events from other runtimes.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful suggestions and comments. We are also grateful for the help from Aleksandar Prokopec during early stages of our development, especially on Renaissance and OpenJDK. This project is sponsored by the US NSF under CNS-1910532 and CNS-2215016.

DATA AVAILABILITY STATEMENT

VESTA is an open-source project. The source code of our system, the comparative system with HPC-based power modeling, together with all raw data can be found at an anonymous website: https://github.com/vesta-power-model/vesta. The supplementary material can be found online [45].

REFERENCES

[1] Timur Babakol, Anthony Canino, and Yu David Liu. 2022. Eflect: Porting Energy-Aware Applications to Shared Environments. In *International Conference on Software Engineering (ICSE'22)* (Pittsburgh, Pennsylvania). Association for Computing Machinery, New York, NY, USA, 823–834. https://doi.org/10.1145/3510003.3510145

[2] Timur Babakol, Anthony Canino, Khaled Mahmoud, Rachit Saxena, and Yu David Liu. 2020. Calm energy accounting for multithreaded Java applications. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 976–988. https://doi.org/10.1145/3368089.3409703

[3] Timur Babakol and Yu David Liu. 2024. Tensor-Aware Energy Accounting. In *International Conference on Software Engineering (ICSE'24)*. ACM. https://doi.org/10.1145/3597503.3639156

[4] Woongki Baek and Trishul M. Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI'10* (Toronto, Ontario, Canada). 198–209. https://doi.org/10.1145/1809028.1806620

[5] R. Bertran, M. Gonzelez, X. Martorell, N. Navarro, and E. Ayguade. 2013. A systematic methodology to generate decomposable and responsive power models for cmps. *IEEE Trans. Comput.* 62, 7 (2013), 1289–1302. https://doi.org/10.1109/tc.2012.97

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[7] William Lloyd Bircher and Lizy K. John. 2012. Complete System Power Estimation using processor performance events. *IEEE Trans. Comput.* 61, 4 (2012), 563–577. https://doi.org/10.1109/tc.2011.47

[8] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[9] D. Brooks, V. Tiwari, and M. Martonosi. 2000. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of 27th International Symposium on Computer Architecture (ISCA'00)*. 83–94. https://doi.org/10.1145/342001.339657

[10] Thomas D. Burd and Robert W. Brodersen. 2000. Design issues for dynamic voltage scaling. In *ISLPED'00*. 9–14. https://doi.org/10.1145/344166.344181

[11] Jianmin Chen, Bin Li, Ying Zhang, Lu Peng, and Jih-kwon Peir. 2011. Statistical GPU power analysis using tree-based methods. In *2011 International Green Computing Conference and Workshops*. 1–6. https://doi.org/10.1109/IGCC.2011.6008582

[12] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[13] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*. https://doi.org/10.1145/1840845.1840883

[14] Perf Events and Tool Security. [n. d.]. online document at https://www.kernel.org/doc/html/latest/admin-guide/perf-security.html. ([n. d.]).

[15] Jason Flinn and M. Satyanarayanan. 1999. Energy-Aware Adaptation for Mobile Applications. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (Charleston, South Carolina, USA) *(SOSP '99)*. Association for Computing Machinery, New York, NY, USA, 48–63. https://doi.org/10.1145/319151.319155

[16] J. Flinn and M. Satyanarayanan. 1999. PowerScope: a tool for profiling the energy usage of mobile applications. In *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*. 2–10. https://doi.org/10.1109/MCSA.1999.749272

[17] Anshul Gandhi, Kanad Ghose, Kartik Gopalan, S Hussain, Dongyoon Lee, Y Liu, Zhenhua Liu, Patrick McDaniel, Shuai Mu, and Erez Zadok. 2022. Metrics for sustainability in data centers. In *Proceedings of the 1st Workshop on Sustainable Computer Systems Design and Implementation (HotCarbon'22)*. https://doi.org/10.1145/3630614.3630622

[18] X. Gao, D. Liu, D. Liu, H. Wang, and A. Stavrou. 2017. E-Android: A New Energy Profiling Tool for Smartphones. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 492–502. https://doi.org/10.1109/ICDCS.2017.218

[19] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The NesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/781131.781133

[20] Liwei Guo, Tiantu Xu, Mengwei Xu, Xuanzhe Liu, and Felix Xiaozhu Lin. 2018. Power Sandbox: Power Awareness Redefined. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 37, 15 pages. https://doi.org/10.1145/3190508.3190533

[21] Lorenz Hilty and Bernard Aebischer. 2015. *ICT for Sustainability: An Emerging Research Field*. Vol. 310. 3–36. https://doi.org/10.1007/978-3-319-09228-7_1

[22] M. Horowitz, T. Indermaur, and R. Gonzalez. 1994. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*. 8–11. https://doi.org/10.1109/LPE.1994.573184

[23] S. Hussain, P. McDaniel, A. Gandhi, K. Ghose, K. Gopalan, D. Lee, Y. Liu, Z. Liu, S. Mu, and E. Zadok. 2024. Verifiable Sustainability in Data Centers. *IEEE Security amp; Privacy* 01 (mar 2024), 2–15. https://doi.org/10.1109/MSEC.2024.3372488

[24] Ahmed Hussein, Mathias Payer, Antony Hosking, and Christopher A. Vick. 2015. Impact of GC Design on Power and Performance for Android. In *Proceedings of the 8th ACM International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '15)*. Association for Computing Machinery, New York, NY, USA, Article 13, 12 pages. https://doi.org/10.1145/2757667.2757674

[25] Intel. [n. d.]. Intel RAPL Interface Advisory, online at https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00389.html.

[26] C. Isci and M. Martonosi. 2003. Runtime power monitoring in high-end processors: methodology and empirical data. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36*. 93–104. https://doi.org/10.1109/MICRO.2003.1253186

[27] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 487–498. https://doi.org/10.1145/2485922.2485964

[28] Jonathan Lew, Deval A Shah, Suchita Pati, Shaylin Cattell, Mengchi Zhang, Amruth Sandhupatla, Christopher Ng, Negar Goli, Matthew D Sinclair, Timothy G Rogers, et al. 2019. Analyzing machine learning workloads using a detailed GPU simulator. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 151–152. https://doi.org/10.1109/ISPASS.2019.00028

[29] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 469–480. https://doi.org/10.1145/1669112.1669172

[30] Tao Li and Lizy Kurian John. 2003. Run-Time Modeling and Estimation of Operating System Power Consumption. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (San Diego, CA, USA) *(SIGMETRICS '03)*. Association for Computing Machinery, New York, NY, USA, 160–171. https://doi.org/10.1145/781027.781048

[31] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*. 355–371. https://doi.org/10.1109/SP40001.2021.00063

[32] Kenan Liu, Gustavo Pinto, and Yu David Liu. 2015. Data-Oriented Characterization of Application-Level Energy Optimization. In *Fundamental Approaches to Software Engineering*, Alexander Egyed and Ina Schaefer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 316–331. https://doi.org/10.1007/978-3-662-46675-9_21

[33] Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2020. From local explanations to global understanding with explainable AI for trees. *Nature Machine Intelligence* 2, 1 (2020), 56–67. https://doi.org/10.1038/s42256-019-0138-9

[34] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 4765–4774. https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html

[35] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. 2020. Recalibrating global data center energy-use estimates. *Science* 367, 6481 (2020), 984–986. https://doi.org/10.1126/science.aba3758

[36] John C. McCullough and Yuvraj Agarwal. 2011. Evaluating the Effectiveness of Model-Based Power Characterization. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. USENIX Association, Portland, OR. https://www.usenix.org/conference/usenixatc11/evaluating-effectiveness-model-based-power-characterization-0

[37] Andreas Merkel, Jan Stoess, and Frank Bellosa. 2010. Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 153–166. https://doi.org/10.1145/1755913.1755930

[38] F. Montevecchi, T. Stickler, R. Hintemann, and S. Hinterholzer. 2020. *Energy-efficient cloud computing technologies and policies for an eco-friendly cloud market, https://digital-strategy.ec.europa.eu/en/library/energy-efficient-cloud-computing-technologies-and-policies-eco-friendly-cloud-market*.

[39] Office of Energy Efficiency & Renewable Energy. 2023. Data Centers and Servers. https://www.energy.gov/eere/buildings/data-centers-and-servers.

[40] Oracle. 2023. DTrace Probes in HotSpot VM. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/dtrace.html.

[41] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. 2012. 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* (Bern, Switzerland) *(EuroSys '12)*. 29–42.

[42] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. 2011. Fine-Grained Power Modeling for Smartphones Using System Call Tracing. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 153–168. https://doi.org/10.1145/1966445.1966460

[43] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Understanding Energy Behaviors of Thread Management Constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 345–360. https://doi.org/10.1145/2660193.2660235

[44] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[45] Joseph Raskind, Timur Babakol, Khaled Mahmoud, and Yu David Liu. [n. d.]. *Vesta Supplementary Material (https://www.cs.binghamton.edu/~davidl/papers/PLDI24Sup.pdf)*. Technical Report.

[46] L. S. Shapley. 1953. A value for n-person games. *Contributions to the Theory of Games (AM-28), Volume II* (1953), 307–318. https://doi.org/10.1515/9781400881970-018

[47] Marina Shimchenko, Mihail Popov, and Tobias Wrigstad. 2022. Analysing and Predicting Energy Consumption of Garbage Collectors in OpenJDK. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*. 3–15. https://doi.org/10.1145/3546918.3546925

[48] A. Sinha and A. P. Chandrakasan. 2001. JouleTrack-a Web based tool for software energy profiling. In *Proceedings of the 38th Design Automation Conference (DAC'01)*. 220–225. https://doi.org/10.1145/378239.378467

[49] P. Sweazey and A. J. Smith. 1986. A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture* (Tokyo, Japan) *(ISCA '86)*. IEEE Computer Society Press, Washington, DC, USA, 414–423. https://doi.org/10.1145/17356.17404

[50] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. 1994. Scheduling for Reduced CPU Energy. In *OSDI'94*. USENIX Association, Monterey, CA. https://doi.org/10.5555/1267638.1267640

[51] Xingfu Wu and Valerie Taylor. 2016. 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)*. 1180–1189. https://doi.org/10.1109/IPDPSW.2016.78

[52] Reza Zamani and Ahmad Afsahi. 2012. A study of hardware performance monitoring counter selection in power modeling of computing systems. In *2012 International Green Computing Conference (IGCC)*. 1–10. https://doi.org/10.

1109/IGCC.2012.6322289

[53] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. 2003. Currentcy: A Unifying Abstraction for Expressing Energy. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*. USENIX Association, San Antonio, TX. https://www.usenix.org/conference/2003-usenix-annual-technical-conference/currentcy-unifying-abstraction-expressing-energy

Received 2023-11-16; accepted 2024-03-31