Portable Power Modeling with Transfer Learning on JVM-Based Applications

Joseph Raskind
SUNY Binghamton
Binghamton, USA
jraskin3@binghamton.edu

Timur Babakol
SUNY Binghamton
Binghamton, USA
tbabako1@binghamton.edu

Yu David Liu
SUNY Binghamton
Binghamton, USA
davidl@binghamton.edu

ABSTRACT

Developing energy-aware applications is an important approach to promoting sustainable computing. This paper addresses a fundamental and challenging problem faced by developers and deployers: how to *port* a power model built for one machine to another? We present LucRETius, a novel approach toward portable power modeling through transfer learning, where a pre-trained power model on one machine can serve as a *teacher* to help construct a *student* model on another machine rapidly. The key insight that enables transfer learning is that the layer of application runtimes can abstract away the machine-specific details, so that two machines—despite different hardware and software system configurations—are unified with a common set of runtime events that can serve as features for transfer learning. We evaluate LucRETius through bi-directional transfers across 4 machines, and we show the power models built by LucRETius have a median percentage error of 1.01%-1.75% when used for predicting the energy consumption of 36 real-world JVM-based applications. Compared with training from scratch, LucRETius can lead to a speed up of 8.07×.

CCS CONCEPTS

• **Software and its engineering → Runtime environments**; **Virtual machines**; • **Hardware → Power estimation and optimization**; • **Computing methodologies → Transfer learning**.

KEYWORDS

power modeling, language runtimes, Java virtual machines

1 INTRODUCTION

An important approach in green software engineering is to develop energy-aware applications, i.e., applications whose behavior may adapt to the energy consumption of their host machine. Complementing systems-level solutions, energy-aware applications fundamentally expand the optimization space of energy efficiency through enabling application-specific optimization. Developing energy-aware applications has received significant attention for mobile systems [5, 13, 16, 20, 21, 23, 40, 57], and in recent years, there is an urgency and a growing interest in building energy-aware applications on server-class systems to address the grand challenge of sustainability [22, 27, 31, 46, 49].

To enable energy awareness, a basic requirement is that the application must be able to obtain the energy consumption of its host machine. Two dominant approaches exist: *measuring* or *modeling*. A simple measurement approach is to attach physical meters to the machine, but it is unfriendly to scalable deployment: to deploy energy-aware applications on 100 machines, one must install 100 meters. Another popular measurement-based approach is to obtain energy consumption information through architecture interfaces such as RAPL [17, 44]. While easy to use, RAPL-based approaches are subject to severe security concerns. First, they require privileged access (root access in Linux). Second, recent studies show exposing RAPL to applications may be vulnerable to side channels attacks [28, 34]. As a result, today's cloud providers, from Google Cloud to AWS [18], have disabled RAPL to their users.

Power modeling [7, 8, 10, 30, 38, 39, 42, 47, 60, 62] is an attractive alternative to support energy-aware applications. Its premise is that the events happening in the computing stack have a strong correlation with the power consumption of the machine, so strong that the former can predict the latter. Relative to measurement-based approaches, a power model, once built, neither requires physical meter deployment nor exposes security risks. Indeed, there is a long history behind research in power modeling, well known for its applications in power simulator design [7, 38] and energy-aware scheduler design [2, 48].

1.1 Portable Power Modeling

While attractive, power modeling — in the use scenario of developing and deploying energy-aware applications — faces a fundamental challenge: *portability*. Existing efforts universally consider power modeling as *machine-dependent*: a power model constructed through the data collected from a machine μ can only be used to model the power consumption of applications running on μ. To model the power consumption on a different machine μ', the power model must be rebuilt *from scratch*, i.e., re-collecting all data on μ' and constructing a brand-new model. Can we *port* the power model built for μ by retrofitting it to the extent possible to enable the power prediction on μ'?

Why Does Portable Power Modeling Matter for Green Software Engineering? Building portable power models are critical for at least four reasons. (1) Building power models from scratch for each and every machine is a time-consuming task. The one-machine-one-model practice does not scale for the reality of software deployment, hindering *the portability of energy-aware software*. (2) Given the astronomical number of computing devices available today, there is a large positive *impact on cost and sustainability* [22, 24, 27] if one can avoid building each power model from scratch. (3) The rise of cloud computing widens the well-known DevOps divide [37]. On one end, the developer is increasingly conscious about energy efficiency, treating it as an important non-functional property in software development. On the other end, the cloud provider is increasingly conservative in exposing RAPL-like interfaces. Porting power models from the developer side to the operator side may potentially *narrow the DevOps divide*, so that energy-aware software can continue to be supported in the cloud. (4) When the system

configuration of a machine evolves—i.e., new hardware components are installed, the OS is upgraded, or runtime settings are reconfigured—a new machine is created from the standpoint of power modeling: a new power model needs to be reconstructed. It would be a laborious task in *software maintenance* if each and every change in the computing stack requires complete retraining of the power model.

Why is Portable Power Modeling Hard? Despite decades of research in power modeling, portability support in power modeling remains a long-standing open problem. Ultimately, this boils down to a number of fundamental challenges.

Challenge I: Complex Whole-Stack Power Impact Blindly copying the power model built from one machine to predict the power of another does not work, nor does it work if one naively scales the power from one computer to another by a constant factor (see § 2). Indeed, it is well known that [32, 53] there is a complex relationship between applications and their power consumption. It depends on numerous factors from the architecture, the scheduler, the threading model, and the runtime. Given two machines μ to μ', many of these contributing factors may lead to the deviation of power consumption.

Challenge II: System-Dependent Events To port a power model from μ to μ', a basic premise is that the events on μ and those on μ' must "match," i.e., they are either similar if not identical, or there is a correspondence between them. This however is not true for the most widely known approach for power modeling, where a power model is built upon hardware-level events known as *hardware performance counters* (HPCs) [6, 8, 30, 47, 62]. To gain intuition, a simple HPC-based power model may be a function f defined as $f(e_{\texttt{L1-dcache-stores}}, e_{\texttt{L1-dcache-loads}}, \dots) = 3 \times e_{\texttt{L1-dcache-stores}} + 4 \times e_{\texttt{L1-dcache-loads}} + \dots$ where the first two arguments indicate the occurrences of the `L1-dcache-stores` and `L1-dcache-loads` events in the system for a given time interval. The problem is that such an approach is fundamentally *hardware-dependent*: different machines may have drastically different hardware details (see § 2).

1.2 This Paper: LucRETius

We introduce LucRETius [1], a novel approach to porting power models built for one machine to another. The key idea is *transfer learning*: a (power) model *pre-trained* for applications running on computer μ can be *fine-tuned* to a model for μ'. The resulting model has comparable precision as one built from scratch on μ', but transfer learning can be accomplished at a fraction of the time required by the latter. In essence, the step of fine-tuning *learns* the complex difference in power characteristics between two machines, a solution to Challenge I.

To address Challenge II, LucRETius relies on *runtime events* to build power models. Relative to lower layers of the computing stack, runtime events are much more stable from one machine to another: machines μ and μ' may have drastically different sets of HPCs, but two JVMs installed on μ and μ' respectively are likely to have similar sets of Java Virtual Machine (JVM) events — from memory access to just-in-time compilation to garbage collection. In other words, it is the abstraction at the layer of runtimes that provides a common set of features, critical for transfer learning. Prior work [56] showed that precise and low-overhead power models can be built from runtime events. Building on top of it, LucRETius further justifies why runtime-level power modeling matters through a new perspective: *portability*.

Name	CPU	Cores	Sockets	Memory	OS
μ_A	Intel Xeon Silver 4316 v3 2.30 GHz	40	1	65.4 GB	Debian 6.1.0-0
μ_B	Intel Xeon E5-2630 v4 2.20 GHz	40	2	65.8 GB	Debian 5.17.0-3
μ_C	Intel Xeon Gold 6548Y+ 2.50 GHz	64	2	380.4 GB	Rockey Linux 9.4
μ_D	AMD Ryzen 5 1600 3.6 GHz	12	1	32.8 GB	Linux Mint 22.1

Table 1: Example Machines. We provide links to each machine's CPU specs: μ_A, μ_B, μ_C, μ_D.

While the main goal of LucRETius is to port power models from one machine to another, we further extend our idea to port power models from one configuration of a machine to another of the same machine, useful in the use scenario of power modeling in the presence of system updates. As an example of this idea, we show that LucRETius can be used to predict the variation in power consumption under different garbage collection (GC) settings: a power model pre-trained for workloads running with one GC setting can be fine-tuned to a model for predicting the power consumption with another GC setting.

To the best of our knowledge, LucRETius is the first system to enable portable power modeling across machines and across configuration settings, a long-standing open problem in energy-efficient computing. This paper makes the following contributions:

- an investigation of an overlooked problem in energy-efficient computing and software engineering: the portability of power models across platforms
- a novel approach that relies on transfer learning of language runtime events, with rigorous specification
- a comprehensive evaluation over 36 state-of-the-art Java benchmarks and *bi-directional* transfers across 4 machines, 4 GC settings, and 2 JDK versions, demonstrating that LucRETius can achieve high precision and low training time

LucRETius is an open-source project. Its code as well as all data is available at the anonymous site https://github.com/lucretius-paper/lucretius.

2 MOTIVATIONS AND USE SCENARIOS

Power Modeling Preliminaries. A power model for a computing environment is a function $\pi : \mathcal{P}(\mathbb{EVENTCOUNT}) \rightarrow \mathbb{POWER}$ where the $\mathbb{EVENTCOUNT} \subseteq \mathbb{INT}$ is a set that represents the number of occurrences of an event in the computer system during a time interval, and $\mathbb{POWER} \subseteq \mathbb{REAL}$ represents the set of power consumption.

In physics, power $P \in \mathbb{POWER}$ is the rate of energy consumption E, following the formula $E = P \times t$ where t is the time. Spanning over a long period of time, the power of a computing system may change. As a result, the formula variant widely used in computer systems takes a *sampling* approach, in the form of $E = \sum_{i=1..n} P_i \times t_i$

[1] LucRETius was a Roman poet and philosopher. He is most known for "transferring" the teachings of Epicurus from ancient Greek to Latin. In our context, LucRETius stands for Learning Runtime Energy Transfer.

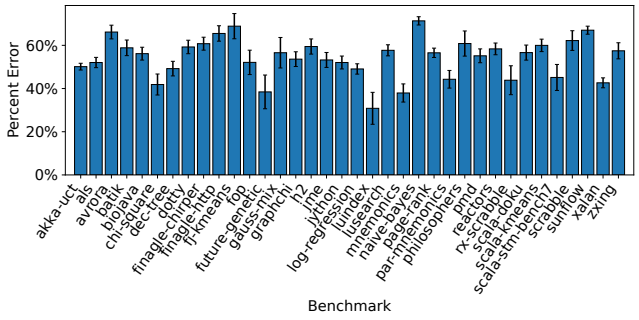


Figure 1: Naively Reusing a Power Model across Machines (The Y-axis shows the percentage error of energy prediction when naively using a power model trained on μ_A to infer the power on μ_C. The X-axis shows the benchmarks whose energy consumption is predicted, with details in § 4. The average error is 54.20%.)

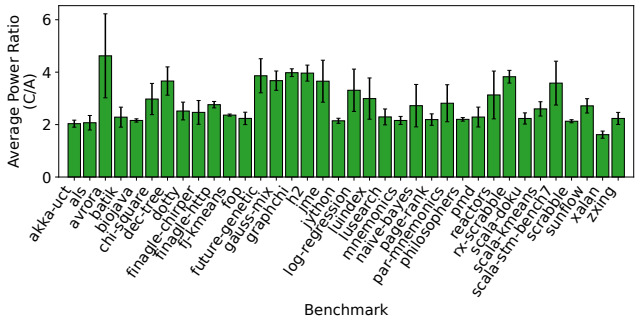


Figure 2: The Workload-Dependent Nature of Power Consumption in the Presence of Machine Change (The Y-axis shows the ratio between the average power consumption of a benchmark on μ_C over that of the same benchmark on μ_A. If the power of the two machines were up to a constant scaling factor, all bars would be of the same height.)

where $t_1, \ldots, t_n$ are the "small enough" time intervals constituting the duration of interest, and $P_1, \ldots, P_n$ are the corresponding power consumption values at each time interval. If $t_1 = \cdots = t_n = \Delta t$, we can further simplify it as $E = \left(\sum_{i=1..n} P_i\right) \times \Delta t$. This formula perhaps explains why power modeling is essential for energy-aware applications: if one may predict the power at each time interval, the energy consumption of an application for some duration can be derived.

From now on, we call the machine where the pre-trained power model has previously been built as the *source machine*, and the machine where the power model is to be built as the *destination machine*. The four machines we use for the evaluation of LucRETius are listed in Table 1.

Naively Reusing a Power Model. Naively reusing the power model from one machine to another does not work. Fig. 1 illustrates how poor the results are by this naive approach. Here, we train a power model on a 20-core machine using trace data collected over 36 applications. We then copy that model to predict the energy consumption of the same applications running on a 64-core machine. It can be seen that some predictions are off by over 60%. By physics, it is common sense that the 64-core machine has a much larger power footprint than the 20-core machine, let alone other machine setting differences.

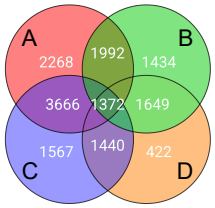
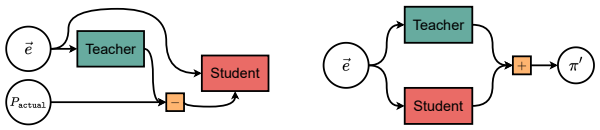


Figure 3: The Machine-Specific Nature of HPCs (The sum of four numbers within a circle labeled x is the total number of available HPCs for machine μ_x, as reported through the `perf` tool. The number of overlapping HPCs is shown at the intersection of circles, as represented as a Venn diagram.)

Naively Scaling a Power Model. Another naive approach is to scale the power model by a constant factor. For example, one might argue that the power consumption of the 64-core machine might be $\frac{64}{20}\times$ that of the 20-core machine. Another may argue that the factor may not be exactly $\frac{64}{20}$, but will still resolve to some constant factor. Fig. 2 illustrates why constant scaling does not work. The reality is that the power consumption of a machine significantly fluctuates from one workload (application) to another. Even for the same application, power consumption may fluctuate significantly over time [3, 4, 29]. In other words, the power consumption of a given machine is not a constant, which renders constant scaling across machines moot. The underlying cause is Dynamic Voltage and Frequency Scaling (DVFS) [11, 26], a standard feature available in nearly all commodity machines we use today.

Machine Dependence of HPCs. In Fig. 3, we show how varied the available HPCs are for three different machines. The root cause of this discrepancy is that HPCs are, by definition, specific to architectures. As a result, if a power model is built on one machine relying on the HPCs available at that machine, it is unclear how to "transfer" such a model to another machine where the available HPCs are radically different.

Use Scenarios. We envision LucRETius is useful in at least three use scenarios. First, LucRETius may speed up power model construction time for *clusters*. It is common for organizations to maintain a cluster of machines, often in groups of tens or hundreds, for research, development, and education purposes. With LucRETius, only one machine needs to have its power model trained from scratch; all others can be trained through LucRETius. As the cluster scales up, it becomes advantageous to cut down on the amount of time spent on training power models. Taking all machines in the



(a) Fine-Tuning (b) Prediction

Figure 4: LucRETius Overview. (In a, the student is trained on the difference between the teacher prediction and the observed power. In b, the predicted power is the sum of the teacher prediction and the student prediction.)

cluster as a whole, our approach may significantly reduce the aggregated training time and the systems/energy resources required for it.

Second, as *cloud computing systems* increasingly restrict the use of RAPL-like hardware features to query energy consumption, LucRETius offers an attractive alternative for developers to continue running their energy-aware applications in the cloud. Here, an end user can pre-train a power model on her own machine (source machine), and the cloud provider can apply LucRETius to build a transfer learning model on behalf of the user where RAPL can be accessed by the cloud provider. Once the model is built, the end user can deploy her applications in the cloud and run them as long as she wishes without the need to access RAPL.

Third, LucRETius offers an economic option for power model training during *system updates*. Such updates can appear anywhere in the computing stack: CPUs may be upgraded, a new OS version may be installed, a GC setting of the JVM may be reconfigured. As expected, the power model may change accordingly. With LucRETius, the power model does not need to be trained from scratch for each update.

3 DESIGN

In this section, we overview the LucRETius design, followed by a detailed specification of our core algorithm.

3.1 Problem Statement and LucRETius Overview

Given a power model π already built for machine μ, the goal of LucRETius is to derive a power model π' for machine μ'.

Recall in § 2, naively reusing π on μ' or naively scaling π to μ' does not work. Indeed, one may resort to *from-scratch training*, i.e., building a power model for μ' solely based on the data collected on μ' without any consultation or derivation from π, but as we shall see (§ 5), it is costly in terms of training time. Instead, LucRETius chooses to build a *student* model π_0 for μ' that leverages the pre-existing model π for μ. In transfer learning terms, the student[2] model uses the previously learned knowledge of the *teacher* π to solve a "similar" problem. Consider a collection of events (in the form of their counts) $\vec{e}$ where $e \in \mathbb{EVENTCOUNT}$ that occur on μ'. Recall in § 2, we mentioned that there is likely a significant "gap" between $\pi(\vec{e})$ and the actual power consumption on μ' due to the fundamental difference between μ and μ'. LucRETius is trained to learn this "gap", i.e., the difference between $\pi(\vec{e})$ and the actual power consumption on μ' where

$$\pi_0(\vec{e}) = P_{\texttt{actual}} - \pi(\vec{e}) \tag{1}$$

Algorithm 1 Key Structures and Routines

```
1:  typedef Event : int
2:  typedef Power : real
3:  typedef Benchmark : string
4:  typedef Model : List<Event> → Power
5:  typedef DataItem : ⟨Benchmark; List<Event>; Power⟩
6:  typedef Data : List<DataItem>
7:  typedef Error : Map<Benchmark, Real>
8:  function Train(d : Data) : Model
9:  glob benchmarks : List<Benchmark>
10: function Run_Iteration(to_run : List<Benchmark>) : Data
11: function Cross_Validation(teacher: Model, d: Data) :
    (Model, Error)
12:     d_train, d_test ← Split(d)
13:     trainf ← map λ⟨b; e; p⟩.⟨e; p − teacher(e)⟩
14:     student ← Train(trainf(d_train))
15:     inferf ← map λ⟨b; e; p⟩.⟨teacher(e) + student(e); p⟩
16:     for all b ∈ benchmarks do
17:         fil ← filter λ⟨b′; e; p⟩.b′ == b
18:         ⟨predE; realE⟩ ← fold (+) inferf(fil(d_test))
19:         r[b] ← |predE−realE| / realE
20:     return ⟨student; r⟩
```

Algorithm 2 Targeted Training with Initial Models

```
1:  procedure Targeted_Training(t_m : Model, target : Er-
    ror) : (Model, Error)
2:      d ←+ Run_Iteration(benchmarks)
3:      i ← Map<Benchmark, int>
4:      while true do
5:          s_m, r ← Cross_Validation(t_m, d)
6:          to_run ← filter (λb.r[b] > target[b] + β and
7:                           i[b] < MAX_ITER) benchmarks
8:          if to_run ≠ [] then
9:              d ←+ Run_Iteration(to_run)
10:             i[b] ← i[b] + 1
11:         else
12:             return ⟨s_m; r⟩
```

where $P_{\texttt{actual}}$ is the ground-truth power consumption measured from the system. This process is illustrated in Fig. 4a, indeed fine-tuning in transfer learning.

After training, the student model can be composed with the teacher model for prediction in the student's system, as shown in Fig. 4b where

$$\pi'(\vec{e}) = \pi(\vec{e}) + \pi_0(\vec{e}) \tag{2}$$

More concretely, this form of model construction is usually called *transductive learning* [50, 51, 59, 63], where the semantics of the data are identical between the teacher model and student model

[2]For a review of the teacher-student terminology, see § 6.

Algorithm 3 Source-Platform Teacher Training Over Abundant Data

```
1: function Teacher_Overtraining() : (Model, Error)
2:     for i = 0, ..., MAX_ITER do
3:         d ←+ Run_Iteration(benchmarks)
4:     return Cross_Validation(λx.0, d)
```

Algorithm 4 Teacher Training with Minimal Iterations

```
1: function Teacher_Minimal_Training() : (Model, Error)
2:     _, target ← Teacher_Overtraining()
3:     return Targeted_Training(λx.0, target)
```

Algorithm 5 Student Training

```
1: function Student_Training() : (Model, Error)
2:     return Targeted_Training(Teacher_Minimal_Training())
```

but the source of data is not. The key question that remains is how much data is needed for training the student to "correct" or "fine-tune" the teacher model.

How the teacher model is trained is orthogonal to our interest. In the current implementation of LucRETius, it relies on Vesta [56].

3.2 Algorithm Specification

The key structures of our algorithm can be found in Algorithm 1. The rest of the Algorithm's define the behavior of LucRETius, spanning the teacher and the student.

As explained in § 1, the power Model is a function from a List of Events to Power. We treat the Data collected from the Benchmarks as a list of tuples. Each tuple can be viewed as a piece of DataItem produced at a given time interval, with 3 components: (1) the Benchmark that produces the data; (2) the List<Event> produced at the time interval; (3) the measured Power at the time interval. Furthermore, we place all prediction Errors in a Map from Benchmarks to Reals.

$k \times 2$ Cross-Validation in Transfer Learning. The key routine in Algorithm 1 is Cross_Validation. The core logic here, known as $k\times$ 2 validation, is widely used in machine learning for constructing and validating models. In power modeling, it has also been previously adopted by McCollough et al. [47] in their effort for constructing HPC-based power models. In this approach, we Split our Data into two halves: a training half (*d_train*) and a testing half (*d_test*). Our implementation of Cross_Validation comes with the feature of transfer learning baked into it. This is achieved by two core mapping functions—*trainf* and *inferf*—which correspond to our discussion in Equations 1 and 2, respectively. Cross_Validation returns a Model alongside the prediction Errors.

Target-Based Training. Routine Targeted_Training defines an iterative process for training based on a `target` of Error. If the precision target is met through Cross_Validation for a given benchmark, no further iterations will be run for that benchmark. As seen here, this iterative routine continually calls Cross_Validation until all Benchmarks are predicted within an acceptable range of Error. Note that a *band* β is added to define whether a cross validation has met the target. In other words, we allow the prediction for a given benchmark to miss the target by β percent. The rationale of the band is to ensure some benchmark is not "trying too hard" in meeting the target. The role of this parameter will be fully revealed in § 5.5 when experimental results are shown.

From-Scratch Training with `MAX_ITER` *Iterations.* When no knowledge of the system is present (i.e., no prior power models exist), Algorithm 3 is used to provide a baseline where an overly abundant amount of data is collected. Here, Teacher_Overtraining runs all the benchmarks `MAX_ITER` number of times. Once all of the Data has been collected, Cross_Validation will be invoked with a "placeholder" Model to ensure the Model returned will be trained without any *de facto* teacher. Recall that our Cross_Validation routine has already considered transfer learning.

From-Scratch Training with Minimal Iterations (Teacher Model). The premise of LucRETius is that transfer learning can significantly speed up the time needed for constructing a power model on the destination machine ($t_{\texttt{transfer}}$), relative to "from scratch" training on that machine ($t_{\texttt{scratch}}$), i.e.:

$$speedup = \frac{t_{\texttt{transfer}}}{t_{\texttt{scratch}}}$$

What constitutes $t_{\texttt{scratch}}$ requires careful consideration. One candidate would be the time needed to run Algorithm 3 on the destination machine. However, with that algorithm including an intentionally excessive data collection, the execution can be arbitrarily long. For example, parameter `MAX_ITER` may be either set to 256 or 2560, as long as it is sufficiently large. In other words, if the execution time for Algorithm 3 were used for $t_{\texttt{scratch}}$, the resulting *speedup* would be arbitrary.

To compute this metric fairly, we introduce *minimal-iteration* "from scratch" training, as defined in Algorithm 4. The intuition is that, given we know the precision that Algorithm 3 can achieve on the destination machine, we can now adaptively determine the number of *minimal* iterations needed to meet the same precision target. $t_{\texttt{scratch}}$ is defined as the execution time for Algorithm 4.

The speed-up metric we defined above is indeed a *lower bound* of speedup, because in practice, a user who conducts "from scratch" training does not know the minimal number of iterations *a priori*: observe that Algorithm 4 is defined based on the assumption that Algorithm 3 has already computed the expected precision when the collected data is sufficiently large. From an end-user perspective, the (conservative) user is more likely to run more iterations than the minimal ones computed by Algorithm 4 just to make sure precision does not further improve. As a result, when we say LucRETius can lead to, say 5× speedup, it is likely to be more than 5 times faster than "from scratch" training by the conservative user.

Student Model. Algorithm 5 creates a student Model by applying the minimal Model (the teacher) alongside its Error into Targeted_Training. In other words, now that we know the teacher model, we can then construct a student model that attempts to meet the Error target of the teacher.

4 IMPLEMENTATION

RAPL at Training Phase. It should be noted that for any training-based approach, including LucRETius, one must obtain the ground truth, which has to come from some form of measurement. In our implementation, such ground truth does come from RAPL. Note however, only training requires privileged access. Once a power model is built, inferring power consumption does not require RAPL access. This works well with the intended use scenarios of LucRETius. For Intel machines (μ_A, μ_B, μ_C), we resort to jRAPL [44] to obtain raw RAPL-reported energy samples. AMD has recently provided the interface to its RAPL support, and we use Powercap [1] to obtain energy samples for μ_D.

Runtime Event Trace. During training and inference, LucRETius obtains runtime events as USDT probes, using the BPF Compiler Collection (BCC) [12]. The concrete list of events can be found in § 5, such as the X-axis of Fig. 8 where we use standard USDT probe names for cross-referencing. For an event named *EVENT* in our paper, there is often a pair of USDT probes named *EVENT*_begin and *EVENT*_end. LucRETius borrows the idea from Vesta in treating such paired events. Intuitively, *EVENT* happens in every time interval between *EVENT*_begin and *EVENT*_end. Vesta also has a more in-depth design on how to account for concurrency, which LucRETius also shares.

The power trace and event trace are collected at a uniform time interval of 1 second, i.e., $\Delta t = 1s$ in § 2.

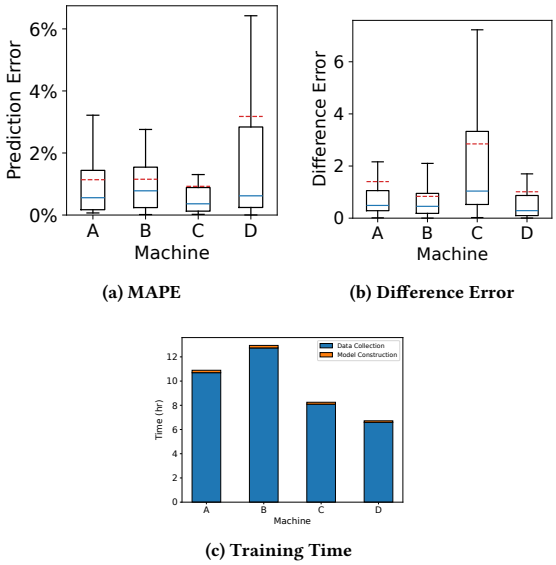
Benchmarks for Data Collection. The data used for training LucRETius was collected from the traces of 36 real-world Java/Scala applications from two benchmark suites: DaCapo [9] and Renaissance [54]. The versions we use are evaluation-git+309e1fa and 0.14.1, respectively. All benchmarks from the two suites are included, except `movie-lens` from Renaissance that deterministically crashes on μ_A. All benchmarks are multi-threaded. Both benchmark suites automatically scale to the number of cores. As a result, when a benchmark runs on system μ_A with 20 cores, or system μ_C with 64 cores, the two executions do not have an identical number of threads as the benchmark suites may adjust them by introspecting the system configurations.

Algorithm Settings. Recall that we first run benchmarks for an overly conservative number of iterations (Algorithm 3). For our experiments, `MAX_ITER` = 256. Further recall that the transfer learning sets a band β (Algorithm 2). Unless explicitly specified, the default band for our experiments is 2%. In our run of determining the minimal iterations of from-scratch training, $\beta = 0$.

For each benchmark, iterations are executed in one hot JVM run. We always discard the first 5 iterations.

JVM settings. Unless otherwise specified, all three systems use OpenJDK 19. `G1GC` is the default garbage collector. Method inlining is enabled, with maximum inline size being 35 bytes. Tiered compilation is disabled. The code cache size is set to 240 MB.

Model Construction. By default, we use XGBoost [14] to train the decision tree-based student model. In § 5.5, we also report our experiences while attempting alternative machine learning models (linear regression and neural networks) as ablation studies. Our teacher model of choice is Vesta [56].



(a) MAPE **(b) Difference Error**

(c) Training Time

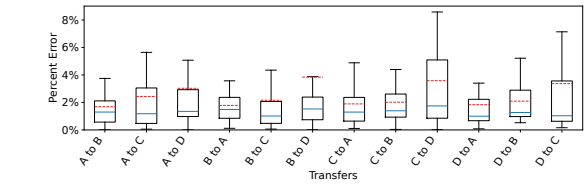
Figure 5: Baselines: From-Scratch Training with Minimal Iterations (In a, the percentage error is represented as a box plot, where the dashed red line indicates the mean average percent error of the total power prediction across all benchmarks. In b, the average difference between the power prediction per interval is represented as a box plot, where the dashed red line is the average difference in watts. In c, the stacked bar represents the overall training time, with the blue/orange bars for data collection and model construction respectively. Labels A, B, C refer to four machines μ_A, μ_B, μ_C, μ_D in Fig. 1.)

5 EVALUATION

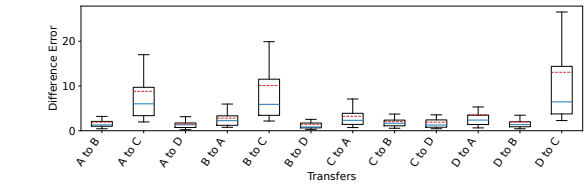
Our evaluation aims at answering the following questions: (**RQ1**) how precise is the power model built by LucRETius for a student machine?(§ 5.2) (**RQ2**) What is the speedup time of LucRETius compared to from-scratch training? (§ 5.2) (**RQ3**) Which language events are most sensitive to the change of machines in terms of power consumption? How important is each event in the model's creation? (§ 5.3) (**RQ4**) How does LucRETius handle changes in system configurations on the same machine, such as the garbage collector and JVM versions? (§ 5.4) (**RQ5**) How do the efficiency and accuracy change under different internal configurations of LucRETius? (§ 5.5). Before we start answering these questions, we first describe the characteristics of our baselines in § 5.1.

5.1 Baseline Characteristics

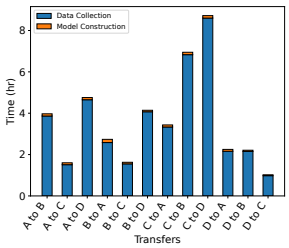
Fig. 5 shows the characteristics of from-scratch training on the four machines we experimented on. As we discussed in § 3, they are *de facto* Vesta runs, except that they are run with a minimal number of iterations that meet the same precision according to Algorithm 4. These results serve two purposes: (a) the model built using from-scratch training serves as the teacher model in our transfer learning. For example, when LucRETius builds a (student) power model for system μ_B, transfer learning from μ_A to μ_B will require a teacher



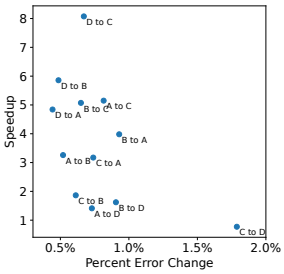
(a) MAPE



(b) Difference Error

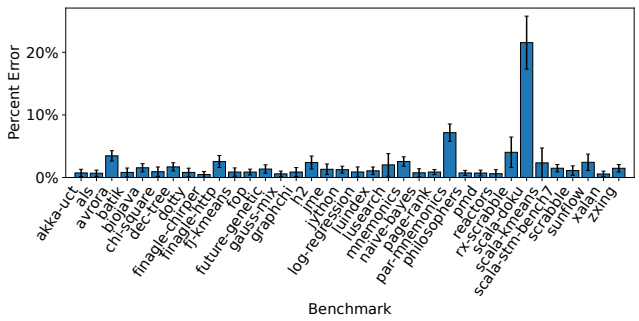


(c) Training Time

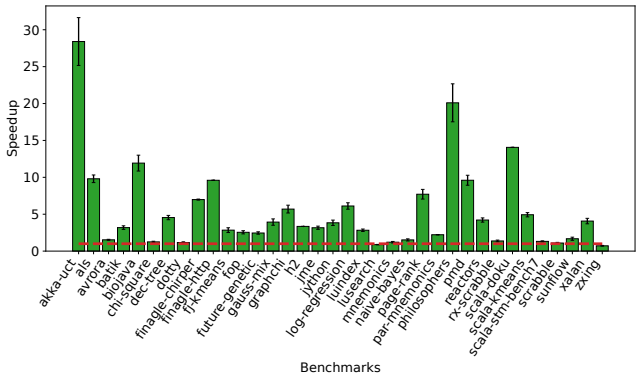


(d) Comparison with From-Scratch Training

Figure 6: LucRETius for Machine-to-Machine Transfer Learning. (Subfigure c is derived from a and b, together with subfigures a and c of Fig. 5. For a dot labeled with "i to j" in this subfigure, the X-axis p% means that if the median percentage error of from-scratch training on machine μ_j is q%, the median percentage error of LucRETius transfer learning from machine μ_i to μ_j is $p + q$%; the Y-axis r means that if the training time of from-scratch training on machine μ_j is t, the training time for LucRETius transfer learning from machine μ_i to μ_j takes $\frac{t}{r}$ time.)



(a) Precision



(b) Data Collection Time Speedup

Figure 7: A Detailed Per-Benchmark View on LucRETius Effectiveness (The transfer scenario is from μ_B to μ_C. In a, the mean and the standard deviation for the energy prediction of each benchmark is shown, against the ground truth actual measured energy consumption. The per-benchmark speedup is computed as the data collection time needed for from-scratch training with minimal iterations for this benchmark, divided by data collection time used by LucRETius training for the same benchmark.)

model for μ_A. (b) the result of from-scratch learning serves as a comparative baseline for understanding the effectiveness of LucRETius. For the same example above, we compare the (transfer) power model built for μ_B by LucRETius against the from-scratch training for μ_B.

Two observations can be made. First, from-scratching training can lead to high precision. Across four systems, the mean absolute percentage error (MAPE) is 0.93%-3.18% and the median absolute percentage error of 0.36%-0.62%. Second, from-scratch training can be time consuming: the training time ranges from 8.25 hours to 12.95 hours [3].

[3]This ratio of time between data collection and model construction is consistent with existing machine learning research, including Large Language Models (LLMs). In the latter setting, it is often publicized that model construction takes significant time. In reality, that portion of time is still dwarfed in magnitudes by the amount of time used to generate and collect data for training.

5.2 RQ1, RQ2: Cross-Machine Transfer Learning

The characteristics of LucRETius are shown in Fig. 6. The first three subfigures show the relative error (in MAPE), the absolute error (in watts), and training time (in seconds). Fig. 6(d) shows the effectiveness of the LucRETius power model for transfer learning from machine μ to μ', relative to from-scratch training on μ'.

LucRETius can lead to competitive power prediction with only a fraction of the training time of from-scratch training. Across all 12 transfer studies, the median percentage errors range from 1.01% to 1.75%, and the mean percentage errors range from 1.69% to 3.84%. Standing alone, this is a competitive result in power modeling, where existing literature generally reports $1 - 10\%$ [6, 8, 30, 47, 62]. Relative to from-scratch training, Fig. 6(d) shows LucRETius is only slightly less precise, by $0.44\% - 1.13\%$, while the speedups in the training are significant. For example, in the case of transfer learning from μ_B to μ_C, the training time speeds up by 8.07×, while there is a 0.65% precision loss relative to from-scratch training on μ_C.

Comparing Intel and AMD machines, we observe that the AMD machine (μ_D) appears to have a larger standard deviation in the relative prediction error (see Fig. 6a). As it turns out, this does not result from the architectural differences between Intel and AMD, but more from the significant difference in the design power of different CPUs: the AMD CPU in μ_D operates at 20W-35W for most workloads, significantly lower than the three Intel machines. To understand why this matters, observe that μ_D *neither* has a larger absolute error *nor* a larger standard deviation for absolute error in power prediction (see Fig. 6b). In other words, what the results show is that a misprediction in the same amount of wattage may have an amplified relative error in a lower-power machine than a higher-power one.

Another observation that can be traced to the same cause is that a transfer-to-μ_D training in general takes more time than its counterpart transfer-from-μ_D training. The reason is that our band β (see § 3.2) is set as a relative value, which implies a smaller wattage range for a lower-power machine. As a result, any transfer learning to μ_D has a smaller margin to meet the target during validation, leading to extended training time. In the most extreme case, μ_C-to-μ_D training, the transfer learning time can exceed the from-scratch training (with minimal iterations). The good news is that such results only happen when a transfer happens from a higher-power machine to a lower-power machine. For most interesting use scenarios, it is more likely a lower-power machine (such as μ_D) — a "test machine" — is used for building the teacher model, subsequently transferred to a higher-power machine (such as μ_C), in which LucRETius is significantly more effective than from-scratch training.

Finally, observe that transfer learning is slightly more prone to outliers. As a symptom, note that our mean percentage errors are generally higher than median percentage errors. To understand this better, we delve into the details in one instance of transfer learning, from μ_B to μ_C, in Fig. 7. Here, note that while LucRETius yields high precision for the vast majority of benchmarks, one benchmark, `scala-doku` has an error of over 20%. As it turned out, `scala-doku` was also among the worst performers in the teacher model itself, with an 8% error. In our context, this means that a poor prediction in the teacher model is likely to be amplified in transfer learning as well.

5.3 RQ3: Cross-Machine Event Sensitivity

Let us now delve deeper in the language runtime events and their roles in power modeling. Fig. 8 shows the per-event importance in the student model, and Fig. 9 shows the importance in the teacher model. The metric we use for determining event importance is SHAP [58], a popular game-theoretic indicator. Intuitively, a higher absolute SHAP value implies that the associated feature is more influential in the decision making process described in the decision tree.

Before we interpret the graphs further, let us make clear that a higher absolute SHAP value in the student model and that in the teacher model have different implications for power modeling. The latter is perhaps easier to understand: the more important a feature is in the teacher model, the more impactful it is in influencing the level of power consumption. The former, however, is more subtle. Recall that the student model captures the difference between the

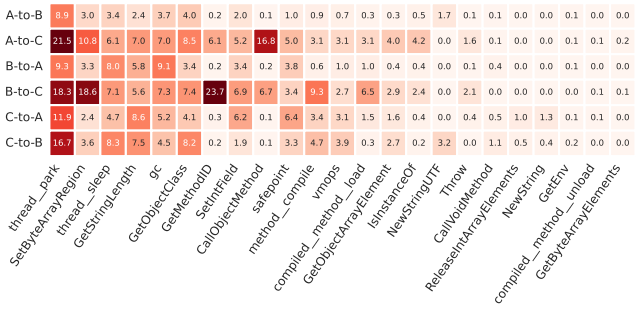


Figure 8: Feature Importance in the Student Model (The X-axis indicates the language events used for model construction. The value in each cell indicates the absolute SHAP value of the event on the X-axis in the Student decision tree built for the transfer indicated on the Y-axis. A higher absolute SHAP value is rendered with more color. Events are sorted on the X-axis based on average SHAP values across all transfers.)

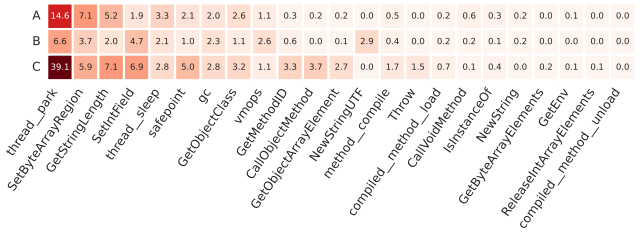


Figure 9: Feature Importance in the Teacher Model (The value in each cell indicates the average absolute SHAP value of the event in the teacher decision tree built for the machine indicated on the Y-axis through from-scratch training.)

actual power consumption on the destination machine and the power prediction of the destination machine by naively reusing the teacher model (§ 3). In other words, a more important feature for this model shows it is more likely to influence *power change* from one machine to another, i.e., how *sensitive* an event in influencing power when machine changes.

First, many events are influential in both determining power consumption and the *change* in power consumption: `thread_park`, `thread_sleep`, and `SetByteArrayRegion` have appeared both in the top-5 of Fig. 8 and the top-5 of Fig. 9. The more interesting question here is why they also contribute significantly to power *change*. When systems scale from a 20-core to a 64-core, thread behavior (`thread_park` and `thread_sleep`) is likely to change radically. This is particularly true that both benchmark suites (§ 4) are designed to scale the number of threads when the underlying systems scale. The scaling impact of `SetByteArrayRegion` is also not difficult to understand. When the number of cores increase, the data path between the CPUs and the memory system becomes more of a bottleneck: the underlying system — from caches to interconnect — may go through significant behavior change, and hence power change.

```
JNI_ENTRY_NO_PRESERVE(jsize, jni_GetStringLength(JNIEnv *
    env, jstring string))
  HOTSPOT_JNI_GETSTRINGLENGTH_ENTRY(env, string);
  jsize ret = 0;
  oop s = JNIHandles::resolve_non_null(string);
  ret = java_lang_String::length(s);
 HOTSPOT_JNI_GETSTRINGLENGTH_RETURN(ret);
  return ret;
JNI_END
```

Figure 10: The Implementation for a JNI-based Method (`GetStringLength`)

Some events are more influential in power change in the presence of platform change than in power consumption itself. For example, gc is a mildly important feature in Fig. 9 but it is consistently more important in all transfers in Fig. 8. This implies that the (power) behavior of GC may change significantly when the underlying system scales. The complexity of GC is well-known. From a power perspective, GC in the presence of machine change may be particularly complex, because both the CPU system (how workloads are distributed in the presence of concurrent GC) and the CPU-memory interaction (how heap is managed) are faced with scalability concerns, and both of them have non-trivial impact on power change.

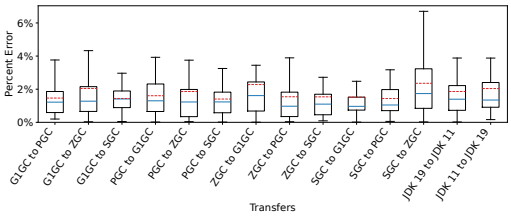
Finally, these results call into question the power efficiency of the Java Native Interface (JNI) in the presence of machine scalability. We were initially surprised that three events, `GetStringLength`, `GetObjectClass` and `GetMethodID`, appear to go through significant power behavior change in the presence of machine change according to Fig. 8. We inspected the source code where these events are generated, one of them shown in Fig. 10. Here, for `GetStringLength`, the internal representation of Java's Unicode-based strings are maintained in the C runtime, and the call for retrieving the string length must cross the JNI boundary. The two other events are produced in the same program pattern[4].

5.4 RQ4: Configuration-Update Transfer Learning

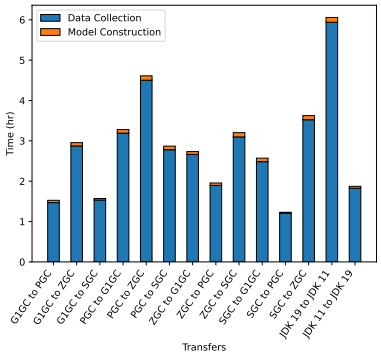
An important use scenario of LucRETius is to fine-tune power models when the configurations of the same machine evolve. In this section, we use GC as an example to show how LucRETius can help build power models.

Fig. 11 shows the precision and training time of transfer training for different GC configurations and JDK versions. Here, we consider 4 common GC configurations: G1 (`G1GC`), parallel GC (`PGC`), serial GC (`SGC`) and ZGC. Across 12 transfer scenarios, the percentage error of energy prediction ranges from 1.41% to 2.36%. In the meantime, the training time ranges from 1.23 hours to 4.61 hours. To set things in perspective, recall that the training time for from-scratch training for system μ_B under the default `G1GC` is around 13 hours (Fig. 5b). All other settings have a similar time for from-scratch training. Additionally, we find that changing the version of the JDK yields similar results: a percentage error of energy predictions between 1.86% and 2.04%, and a training time range from 1.87 hours to 6.06 hours.

[4]Implementations for the mentioned JNI calls can be found in https://github.com/openjdk/jdk/blob/master/src/hotspot/share/prims/jni.cpp, at lines 2153-2160 for `GetStringLength`, lines 1119-1125 for `GetMethodID`, and lines 1036-1045 for `GetObjectClass`.



(a) Precision



(b) Training Time

Figure 11: LucRETius for Power Modeling with GC Setting and JDK Change (Each plot shows the mean and the standard deviation of predicting the energy consumption of 36 benchmarks against the ground truth of actual measured energy consumption, where the power model is constructed with LucRETius under the transfer scenario specified under each box. All experiments are performed on machine μ_B.)

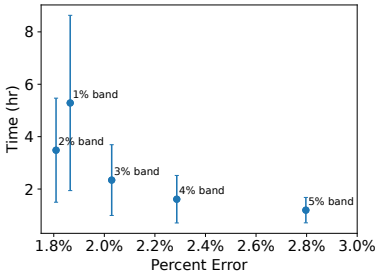


Figure 12: The Impact of Band Size on Training (X: mean percentage error over 6 transfers; Y: mean training time.)

5.5 RQ5: Ablation Studies

Band Size The algorithm of LucRETius has few parameters to customize. The most notable one is the band size (β in Algorithm 2). Intuitively, it means how strictly a model has to meet the precision target in the teacher model. The impact of this parameter setting can be found in Fig. 12, with our default choice as 2% (§ 4).

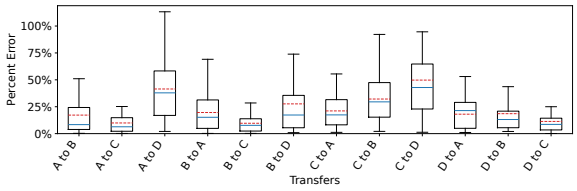


Figure 13: All Transfers with Linear Regression. (X: the transfer models; Y: mean precision.)

First, note that the choice of band size has relatively small impact on precision, but it has a significant impact on training time. For the former, the 5 band choices only have a difference of 1% percentage error, but the training time can vary from 1.19 hours to 5.29 hours. Our default choice of 2% represents a good trade-off between precision and training time: it has the smallest error while also having a moderate training time. Depending on the end-user need, it should be noted that other band choices may be equally useful. For example, LucRETius with the 5% band can complete training in 1.19 hours on average with a 2.8% error. For use scenarios where this error rate is acceptable, the rapid training rate may be preferable.

Second, the band setting of $X\%$ does not mean the final precision under that setting will be at least $X\%$ worse than the teacher model. For example, the 5% band only leads to an average of 2.8% error at the end, evidently *less than* 5% worse than the teacher model, regardless of what the error of the teacher model is. There are two reasons. First, the band only sets an upper bound for the termination of running a benchmark. For many benchmarks, by the time the condition is met, the error is less than the error of the teacher model plus 5%. Second, the data collection of a particular benchmark b may discontinue because the target plus the band is met, but the data for other benchmarks, say b', may continue to be collected. It is common that b' shares some commonality with b in terms of workload characteristics, so the newly collected data b' may continue strengthening the precision for predicting the power consumption for b.

Linear Modeling The power model constructed by LucRETius is precise, but by Occam's Razor, could a *simpler* model be built with comparable precision? We conducted an ablation study on building the student model with linear regression.

As shown in Fig. 13, the mean percentage error for 6 transfer scenarios range from 9.64% to 32.16%. This is a significant drop in precision from LucRETius (1.69% to 2.44% as shown in Fig. 6). As model construction takes a very small portion of training time (§ 5.1), linear modeling, even if faster than decision tree modeling, entails negligible reduction in overall training time.

Modeling with Neural Networks We alternatively explored the use of neural networks (NN) as our student model, but ultimately found the approach undesirable for our dataset. We attempted to construct deep neural networks of 3, 5, 10, 15, 25, 50, 75, and 100 layers. For each layer size, we varied topologies of each layer in line with typical heuristic training patterns such as doubling and halving node counts between layers. We also tuned hyperparameters such as learning time, activation functions, and optimizers. Together, we attempted 40 different topology and hyperparameter configurations. In all configurations, the error rates are multiple magnitudes larger than those of LucRETius. Due to the large errors, an additional undesirable consequence is that the data collection for the student model always requires the same number of iterations as the teacher model (see § 3.2): there is no benefit of transfer learning in training time, and one might as well perform from-scratch training on the destination machine. We believe an NN-based transfer learning model does not perform well is a combination of two factors:

- (N1) it requires a large amount of data to reach accuracy;
- (N2) it requires significant model tuning to reach accuracy.

(N1) requires more data collection time. While we think (N2) can be further improved if we adopted a more systematic search, it would imply longer model construction time. Given that a primary benefit of a transfer learning approach is that it is more rapid than from-scratch learning, training time (data collection + model construction time) is of primary importance. In other words, while we cannot rule out the fact that an NN-based model could be more accurate when further improvements are made via (N1) or (N2), its training time is likely to be (significantly) longer than LucRETius. Given the relatively low error rate, LucRETius may represent a sweet spot with competitive accuracy and attractive training time.

6 RELATED WORK

Portability in power modeling is an underexplored problem with little prior work. In this section, we summarize related work on transfer learning, as well as understanding and optimizing the energy consumption of application runtimes.

Transfer learning [51] is a classic but rapidly developing technique in machine learning. Thanks to its success in addressing large-scale language modeling problems [50, 59, 63], this approach has received significant attention and rapid adoption. The methodology that composes pre-training (teacher) and fine-tuning (student) used by LucRETius is well known in deep learning. In recent years, transfer learning is successful in *model adaptation* — the task of molding a model to fit a new domain or range [33, 35, 41, 45, 61] — so much so that the nomenclature of transfer learning becomes more intertwined with the latter. The terminology used in this paper is closer to the original definition of transfer learning. In this sense, LucRETius's construction is based on *transductive* learning where the data is structurally identical but did not come from the same source. This is in contrast to *inductive* learning where the data comes from the same source but the target task changes. Transduction is common for language recognition [19, 55] and model compression techniques such as knowledge distillation [25].

GreenScaler [15] predicts the energy consumption of an application through a testing-centric approach: the prediction model is built on the energy behavior of test runs driven by automated random test generation where test selection is guided by a heuristic on CPU utilization. While *GreenScaler* does not share our design goal (portability), our prediction goal (per-interval power), or our technique (transfer learning), it is an interesting instance at the intersection of green software engineering and machine learning, which LucRETius also belongs to.

LucRETius has a complementary relationship with empirical studies on the energy efficiency of application runtimes in software engineering. The impact of thread management on energy consumption has been studied by Pinto et al. [53], and a more detailed study [52] on the ForkJoin framework [36]. Vincent [43] is an energy-efficient JVM that builds DVFS into just-in-time compilation. While existing work focuses on quantifying the energy impact of specific runtime features, LucRETius takes a feature-neutral approach, aiming to build a power model across diverse runtime events (in the challenging setting of porting power models).

7 THREATS TO VALIDITY

Our results are faced with several threats of external validity. First, our findings are only evidenced on Intel and AMD machines. The main constraint is that we need RAPL-supporting architectures to collect energy data (to establish the ground truth during the training phase). This should not be a fundamental challenge if physical meters are used. Additionally, our model was trained on data from 36 benchmarks. These benchmarks do not encompass the entirety of JVM-based application behaviors, and thus may result in misprediction for unknown workloads. Indeed, LucRETius follows a long line of power modeling research [8, 30, 39, 47, 60, 62] with a focus on establishing *predictability* (i.e., cross-machine power consumption *can* be predicted) for unknown traces from known workloads (i.e., new runs from known applications). Improving coverage (i.e., ensuring the power predictability of arbitrary unknown workloads) is an orthogonal issue largely hinging on data collection and curation. Furthermore, while we attempted linear regression and neural networks as well as decision trees (our model of choice), it is conceivable that better results could be found with a different model, or different model configurations. Finally, our experiments are all conducted over Java and Scala applications, and we have no experimental evidence on non-JVM languages. In principle however, our approach is not tied up to JVM-based languages: the only requirement is that the language runtime provides the ability to track its events.

8 CONCLUSION

LucRETius is a novel approach to precise and rapid power modeling by porting a power model pre-constructed for one machine to another. At its essence, LucRETius is a transfer learning model at the layer of language runtimes, whose events unify the features for model construction despite the diversity of machines. Our experiments show LucRETius can build precise power models while only taking a fraction of the training time required by model construction from scratch. Portable power modeling can significantly reduce the cost of today's practice of training each computer from scratch, provide scalable deployment to energy-aware application developers, and facilitate power modeling updates in the presence of software/hardware updates.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful suggestions and comments. This project is sponsored by the US NSF under CNS-2215016 and CNS-2426352.

REFERENCES

[1] [n. d.]. Power Capping Framework - The Linux Kernel documentation. ([n. d.]). https://docs.kernel.org/power/powercap/powercap.html

[2] Esmail Asyabi, Azer Bestavros, Erfan Sharafzadeh, and Timothy Zhu. 2020. Peafowl: In-Application CPU Scheduling to Reduce Power Consumption of in-Memory Key-Value Stores. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 150–164. https://doi.org/10.1145/3419111.3421298

[3] 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

[4] 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

[5] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting Energy Bugs and Hotspots in Mobile Apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 588–598. https://doi.org/10.1145/2635868.2635871

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] David Calavera and Lorenzo Fontana. 2020. *Linux observability with BPF: Advanced Programming for performance analysis and Networking*. O'Reilly Media, Inc., Sebastopol, CA.

[13] Anthony Canino, Yu David Liu, and Hidehiko Masuhara. 2018. Stochastic energy optimization for mobile GPS applications. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 703–713.

[14] 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

[15] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. 2019. GreenScaler: training software energy models with automatic test generation. *Empirical Softw. Engg.* 24, 4 (Aug. 2019), 1649–1692. https://doi.org/10.1007/s10664-018-9640-7

[16] André Luiz Tinassi DAmato, Linnyer Beatrys Ruiz, Anderson Faustino da Silva, and José Camargo da Costa. 2011. EProf: An accurate energy consumption estimation tool. *2011 30th International Conference of the Chilean Computer Science Society*. https://doi.org/10.1109/sccc.2011.28

[17] 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

[18] Benjamin Davy. [n. d.]. https://medium.com/teads-engineering/estimating-aws-ec2-instances-power-consumption-c9745e347959. ([n. d.]). https:

//medium.com/teads-engineering/estimating-aws-ec2-instances-power-consumption-c9745e347959 Retrieved July 21, 2025.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[20] 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

[21] 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)*. ACM, New York, NY, USA, 48–63. https://doi.org/10.1145/319151.319155

[22] 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

[23] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. 2013. Estimating mobile application energy consumption using program analysis. In *ICSE '13* (San Francisco, CA, USA). 92–101.

[24] 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

[25] Geoffrey Hinton. 2015. Distilling the Knowledge in a Neural Network. *arXiv preprint arXiv:1503.02531* (2015).

[26] 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

[27] 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 & Privacy* 01 (mar 2024), 2–15. https://doi.org/10.1109/MSEC.2024.3372488

[28] Intel. [n. d.]. Intel RAPL Interface Advisory, online at https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00389.html. Retrieved July 21, 2025.

[29] Canturk Isci and Margaret Martonosi. 2003. Identifying program power phase behavior using power vectors. In *In Workshop on Workload Characterization*.

[30] 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

[31] Kostis Kaffes, Dragos Sbirlea, Yiyan Lin, David Lo, and Christos Kozyrakis. 2020. Leveraging Application Classes to Save Power in Highly-Utilized Data Centers. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 134–149. https://doi.org/10.1145/3419111.3421274

[32] Melanie Kambadur and Martha A. Kim. 2014. An Experimental Survey of Energy Management across the Stack. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 329–344. https://doi.org/10.1145/2660193.2660196

[33] Aman Kansal, Feng Zhao, Jie Liu, Nupur Kothari, and Arka A. Bhattacharya. 2010. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/1807128.1807136

[34] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3, 2, Article 9 (March 2018), 26 pages. https://doi.org/10.1145/3177754

[35] Nicholas D. Lane, Petko Georgiev, and Lorena Qendro. 2015. DeepEar: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (Osaka, Japan) *(UbiComp '15)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/2750858.2804262

[36] Doug Lea. 2000. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande* (San Francisco, California, USA) *(JAVA '00)*. Association for Computing Machinery, New York, NY, USA, 36–43. https://doi.org/10.1145/337449.337465

[37] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. 2019. A Survey of DevOps Concepts and Challenges. *ACM Comput. Surv.* 52, 6, Article 127 (Nov. 2019), 35 pages. https://doi.org/10.1145/3359981

[38] 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

[39] 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

[40] Ding Li, Angelica Huyen Tran, and William G. J. Halfond. 2014. Making web applications more energy efficient for OLED smartphones. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 527–538.

[41] Rui Li, Qianfen Jiao, Wenming Cao, Hau-San Wong, and Si Wu. 2020. Model Adaptation: Unsupervised Domain Adaptation Without Source Data. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 9638–9647. https://doi.org/10.1109/CVPR42600.2020.00966

[42] 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

[43] Kenan Liu, Khaled Mahmoud, Joonhwan Yoo, and Yu David Liu. [n. d.]. Vincent: Green Hot Methods in the JVM. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). 32:1–32:30.

[44] 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

[45] Hong Lu, Denise Frauendorfer, Mashfiqui Rabbi, Marianne Schmid Mast, Gokul T. Chittaranjan, Andrew T. Campbell, Daniel Gatica-Perez, and Tanzeem Choudhury. 2012. StressSense: detecting stress in unconstrained acoustic environments using smartphones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing* (Pittsburgh, Pennsylvania) *(UbiComp '12)*. Association for Computing Machinery, New York, NY, USA, 351–360. https://doi.org/10.1145/2370216.2370270

[46] 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

[47] 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

[48] 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

[49] Office of Energy Efficiency & Renewable Energy. 2023. Data Centers and Servers. https://www.energy.gov/eere/buildings/data-centers-and-servers. Retrieved July 21, 2025.

[50] Sinno Jialin Pan and Qiang Yang. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* 22, 10 (2010), 1345–1359. https://doi.org/10.1109/TKDE.2009.191

[51] David Perkins and Gavriel Salomon. 1999. Transfer Of Learning. 11 (07 1999).

[52] Gustavo Pinto, Anthony Canino, Fernando Castor, Guoqing (Harry) Xu, and Yu David Liu. 2017. Understanding and overcoming parallelism bottlenecks in ForkJoin applications. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 765–775.

[53] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Understanding Energy Behaviors of Thread Management Constructs. In *OOPSLA '14*.

[54] 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

[55] Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training. https://api.semanticscholar.org/CorpusID:49313245

[56] 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

[57] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. 2011. Energy Management in Mobile Devices with the

Cinder Operating System.

[58] 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

[59] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A survey of transfer learning. *Journal of Big data* 3 (2016), 1–40.

[60] 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

[61] Dong Yu, Kaisheng Yao, Hang Su, Gang Li, and Frank Seide. 2013. KL-divergence regularized deep neural network adaptation for improved large vocabulary speech recognition. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 7893–7897. https://doi.org/10.1109/ICASSP.2013.6639201

[62] 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

[63] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning. *Proc. IEEE* 109, 1 (2020), 43–76.