Calm Energy Accounting for Multithreaded Java Applications

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

Anthony Canino
acanino1@binghamton.edu
SUNY Binghamton
Binghamton, NY, USA

Khaled Mahmoud
kmahmou1@binghamton.edu
SUNY Binghamton
Binghamton, NY, USA

Rachit Saxena
rsaxena3@binghamton.edu
SUNY Binghamton
Binghamton, NY, USA

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

ABSTRACT

Energy accounting is a fundamental problem in energy management, defined as attributing global energy consumption to individual components of interest. In this paper, we take on this problem at the application level, where the components for accounting are *application logical units*, such as methods, classes, and packages. Given a Java application, our novel runtime system Chappie produces an *energy footprint*, i.e., the relative energy consumption of all programming abstraction units within the application.

The design of Chappie is unique in several dimensions. First, relative to targeted energy profiling where the profiler determines the energy consumption of a pre-defined application logical unit, e.g., a specific method, Chappie is *total*: the energy footprint encompasses all methods within an application. Second, Chappie is *concurrency-aware*: energy attribution is fully aware of the multithreaded behavior of Java applications, including JVM bookkeeping threads. Third, Chappie is an embodiment of a novel philosophy for application-level energy accounting and profiling, which states that the accounting run should preserve the *temporal* phased power behavior of the application, and the *spatial* power distribution among the underlying hardware system. We term this important property as *calmness*. Against state-of-the-art DaCapo benchmarks, we show that the energy footprint generated by Chappie is precise while incurring negligible overhead. In addition, all results are produced with a high degree of calmness.

CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

Energy Accounting, Energy Profiling, Power Disturbance, Concurrency

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409703

ACM Reference Format:
Timur Babakol, Anthony Canino, Khaled Mahmoud, Rachit Saxena, and Yu David Liu. 2020. Calm Energy Accounting for Multithreaded Java Applications. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368089.3409703

1 INTRODUCTION

Application-level energy management has emerged as an important dimension of energy-efficient computing, with solutions sharing a common premise: the *logical units* within an application matter in energy management. In this paper, we revisit *energy accounting*, a fundamental problem that has been extensively studied at the lower layers of the computing stack [20, 52]. Our new focus is on the application level: determining the energy consumption of individual application *logical units* given the total energy consumption of the application. This problem subsumes important questions in green software development, e.g., which methods consume the most energy in an application, which methods lead the underlying system to a higher-power state, or how to compare the energy/power consumption of two methods. Answering these questions systematically may significantly impact the state of the art of energy-aware programming [8, 13, 16, 37, 49, 53], energy-adaptive software framework design [14, 21, 30, 34, 39], energy testing and debugging [12, 23, 27, 38, 40, 42], and energy-oriented approximation [5, 11, 29, 47].

At first glance, the problem of application-level energy accounting may appear deceptively simple. To determine the energy consumption of method *m*, a naive approach may reduce it to a trivial energy *measurement* problem: one may instrument *m* at its entrance and exit program points, obtain a pair of energy readings from the underlying hardware, and compute the difference of the two. This naive approach however is flawed — or at best impractical — for two reasons.

Accounting Totality. Important questions such as which method consumes the most energy in an application requires the knowledge of energy consumption for all methods in an application. Iterating over every method with an instrumentation approach does not scale for realistic applications; the overhead becomes unrealistically high (see §2) when all methods are accounted for at the same time.

Multi-threading. More fundamentally, the instrumentation-based approach is concurrency-oblivious, which may lead to significant

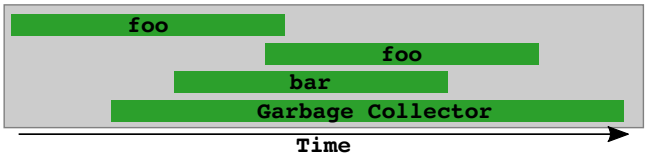


Figure 1: The Challenge of Accounting for the Energy Consumption of Method `foo` with a Multi-threaded Application

over-accounting. This is particularly bad news for server-type applications where multi-threaded programs on parallel platforms are the rule, not the exception. Consider Figure 1. The instrumentation-based approach may attribute the entire energy consumption during the execution of `foo` to the method, without considering a portion of such consumption results from the execution of `bar`, or even another instance of `foo`. In addition, managed language runtimes may have co-running Virtual Machine (VM) bookkeeping threads, such as the garbage collector.

Beyond these two challenges in energy accounting system design, a third one exists against producing high-quality results.

Disturbance. Any runtime system design that profiles the application needs to ensure that the runtime profiling mechanisms do not significantly alter the behavior of the original application. This well-known problem is usually coped with through ensuring the execution time overhead of the profile run is negligible compared to the original run. Profiling for energy, however, introduces a new layer of subtlety. The profiler behavior may alter the power state of the underlying hardware, leading to two consequences. First, a change in the power states is correlated with the CPU frequency change, so that execution time is no longer a reliable indicator of disturbance. Second, as energy is the accumulated effect of power, the disturbance of power consumption alters the very behavior that the profiler tries to a observe, challenging the validity of the results from energy accounting.

1.1 Chappie

We present Chappie, a *total, concurrency-aware, disturbance-mitigated* application-level energy accounting system for Java applications. With Chappie, the execution of each application can produce an *energy footprint*, i.e., the relative energy consumption of all programming logical units within the application. For example, Figure 2 is a top-10 energy footprint for a realistic Java application `h2` [3]. It shows `ValueDataType.compare` is the most energy-consuming method in the application. Furthermore, it shows the relative energy consumption difference between any pair of methods, such as `Page.binarySearch` and `ValueDataType.compareValues`. In addition to methods, Chappie is customizable so that the unit in an energy footprint can be courser-grained as classes and packages, or finer-grained as calling contexts to methods.

At its heart, Chappie is a novel *sampling-based* runtime system that draws information from multiple layers of the computing stack, and composes it to provide the energy footprint. On the JVM level, Chappie samples the per-thread call stacks, determining which methods are currently executing at each sample. On the hardware level, Chappie is able to sample the raw energy readings. When

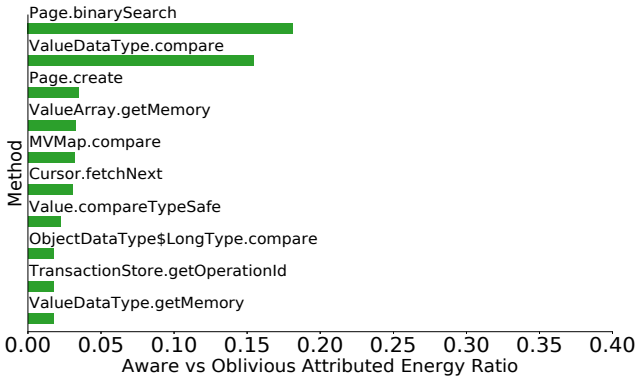


Figure 2: Method-Grained Energy Footprint for `h2` (The X-Axis represents the percentage of energy consumption consumed by a particular method. The Y-Axis lists the methods with top-10 energy consumption, with the most consuming one at the top.)

multiple threads are active upon the receipt of a raw energy sample, Chappie is able to distribute a share of the latter among all running methods, each of which resides on the call stack of an active thread. The core algorithm of Chappie combines the high-level information with the lower-level one, with full awareness of concurrency. Despite its cross-layer nature, Chappie is a *lightweight* design with no modifications to the application code, the compiler, the JDK, the JVM, the OS, or the hardware.

Another highlight of Chappie is it treats accounting disturbance as a first-class concern. While overall execution time overhead and overall energy consumption overhead may provide some insight into how an accounting system may affect the application to be accounted for, they are insufficient to quantify disturbance in energy accounting.

1.2 Calm Energy Accounting

Chappie features *calm energy accounting* with a novel fine-grained metric to ensure the accounting system does not *temporally* and *spatially* disturb the energy behavior of the original application, so that the energy footprint faithfully captures the characteristics of the original application. To elaborate, let us consider the scenario in Figure 3, where an application runs on a 2-core machine, `CPU1` and `CPU2`, each of which may operate at two CPU frequencies, `1GHz` and `2GHz`. The metric of *calmness* subsumes two ideas.

Temporal Calmness. The accounting run should preserve the *power phased behavior* of the original application [19, 31], i.e., the power consumption of an application may vary from timestamp to timestamp and together they may form a pattern. This well-known phenomenon results from time-dependent application characteristics such as parallel vs. serial phases latent in multi-threaded applications, and CPU-intensive vs. I/O-intensive phased behaviors. The power variation is usually enabled by Dynamic Voltage and Frequency Scaling (DVFS) [43], a standard feature supported by the vast majority of CPUs and enabled as default by most operating systems. In Figure 3b, observe that the accounting run drives both

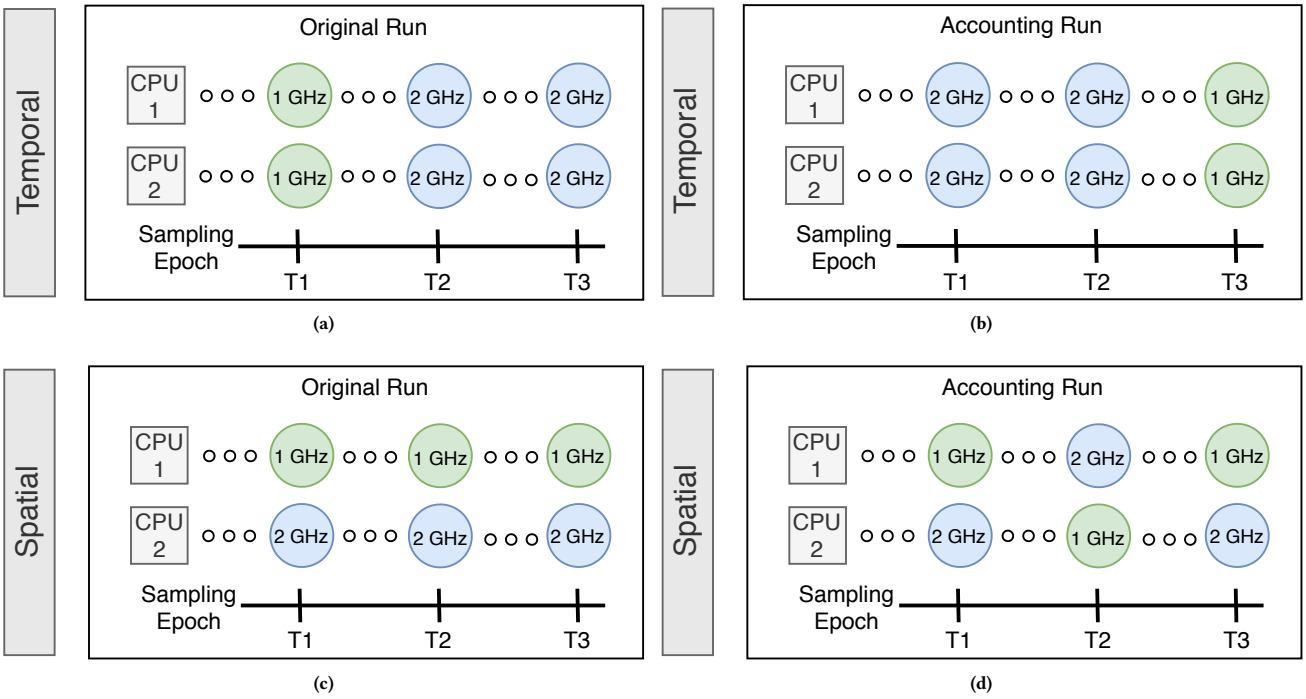


Figure 3: Temporal Power Disturbance (Figure (a) (b)) and Spatial Power Disturbance (Figure (c) (d))

CPU1 and CPU2 to operate at 2Ghz at an earlier timestamp than the original application, as shown in Figure 3a. Even though the accumulated energy of the two runs may well correspond, the two runs do not exhibit the same power behavior, which we call *temporal power disturbance*. In a nutshell, temporal power disturbance is a symptom of the phased behavior change of the application.

Spatial Calmness. The accounting run should preserve the *power distribution behavior* of the original run, i.e., the power consumption of the underlying system may vary from CPU core to CPU core and together they may form a pattern. This well-known phenomenon results from application characteristics such as symmetric vs. asymmetric workloads, and the level of parallelism enabled by the application-scheduler interaction. The power variation is again enabled by DVFS, where a higher workload usually drives the CPU to a higher frequency state — hence higher power consumption — and vice versa. In Figure 3c, the power distribution between CPU1 and CPU2 are lopsided, whereas in Figure 3d, the two are more balanced. Even though the accumulated energy of the two runs may well correspond, which we call *spatial power disturbance*. In essence, spatial power disturbance is a symptom of the workload behavior change of the application.

We evaluate Chappie over 11 state-of-the-art DaCapo benchmarks with diverse and realistic workloads. We show that Chappie can successfully produce the energy footprint for all benchmarks. By judiciously setting the sampling rates, Chappie can achieve both temporal calmness and spatial calmness for all benchmarks. When calmness is achieved, the time and energy overhead are negligible, with an average runtime overhead of 3.15±3.03% and an average energy overhead is 0.84±1.09% across all benchmarks. We further show that the produced energy footprints are precise: the energy footprint converges when samples from additional runs are included.

Contribution. This paper makes the following contributions:

- A novel and customizable energy accounting design that produces an *energy footprint*, illustrating the relative energy consumption of all program logic units.
- A *sampling*-based *cross-layer* design that allows for *concurrency-aware* energy accounting for multi-threaded Java applications
- A novel *calmness* metric for quantifying power disturbance in application-level energy accounting
- An evaluation demonstrating the effectiveness of Chappie in understanding the energy behavior of realistic Java application

Chappie is an open-source project, hosted at an (anonymous) website `https://github.com/pl-chappie/chappie`.

2 CHAPPIE MOTIVATION

In this section, we briefly motivate the need for Chappie quantitatively.

Accounting Totality with Instrumentation. In order to support totality in energy accounting, an instrumentation-based approach would either need to iteratively instrument each method, or instrument all methods at once. The former would lead to a large space to consider. The latter approach, despite requiring few runs, may

Table 1: Overhead with the Instrumentation-Based Approach (The "instrumentation X%" column shows the time overhead when every method in the benchmark is instrumented and X% of the invocations are randomly selected to take energy readings. The instrumentation was performed through `javassist` [15]. All benchmarks appearing in this section are from the DaCapo benchmark suite [9].)

	instrumentation 100%	instrumentation 5%
`sunflow`	189x	164x
`batik`	118x	117x
`xalan`	30x	29x
`h2`	25x	16x

Table 2: Energy Over-Attribution due to Concurrency-Oblivious Accounting (The first column lists top-consuming `h2` methods reported by Chappie, and the second column reports the ratio of over-attribution if concurrency were ignored.)

method name	over-attribution
`ValueDataType.compare`	2.88x
`Page.binarySearch`	2.20x
`MVMap.compare`	5.82x
`MVMap.binarySearch`	9.27x
`LocalResult.next`	3.43x

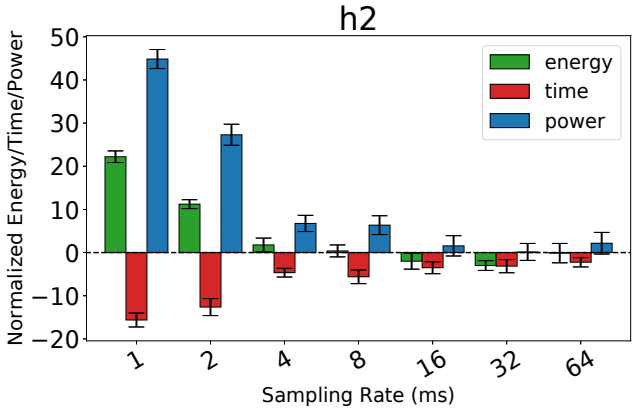


Figure 4: `h2` Disturbance (The X-Axis represents different sampling rates. The Y-Axis represents the normalized difference between the accounting run and the original non-accounting run. Each group consists of three bars, the energy/time/power consumption. The error bars indicate standard deviation.)

lead to severe overhead. Table 1 shows the overhead of the latter approach. Even with the aggressive omission of energy readings, that approach still yields a significant overhead. A compiler-based approach of selective instrumentation may help, but with the gap being so large, it is unlikely such an approach would produce results with practically viable overhead.

Note that even if the instrumentation-based approach can get over the hurdle of long iteration or significant overhead, such an approach is still fundamentally concurrency-oblivious: during the execution between the method start and end, other methods may be co-running.

Concurrency-Oblivious Accounting. To quantify the impact of concurrency awareness, we construct an experiment to show how much energy over-attribution could happen should Chappie ignore multi-threading. As this experiment is only for motivation purposes, we take the simple approach of assigning 100% of each energy sample to a method if its host thread is active in that sampling interval; when multiple threads are active, duplicated assignments are possible. We show these results in Table 2. The take-away message here is that a concurrency-oblivious energy accounting design may lead to significant over-attribution of method energy consumption.

Power Disturbance. Our investigation into calmness was motivated by the counter-intuitive behavior of some benchmarks during the early stage of the Chappie evaluation.

Our intuition was that the bookkeeping of accounting may carry some overhead so the accounting run would be slower than the original run. Some experiments however revealed an opposite trend. As shown in Figure 4, if Chappie had set the sampling rate at 1 millisecond (ms), the execution of `h2` would turn out to be nearly 18% *faster* than the original non-accounting run!

The key to resolving this baffling mystery is power consumption. As it turns out, the sampling rate of 1ms would lead to nearly a 50% of power increase: the sampling-based approach periodically "pokes" the application, preventing the application from sleeping during downtime. In these scenarios, DVFS likely increases CPU frequency to handle the higher workload, which in turn allows the application to run faster.

The moral of the story is that accounting, if designed naively, may introduce disturbance and significantly alter the energy behavior of the original application. The good news is power disturbance will be reduced as the sampling rate decreases: as sampling slows to 32ms, the power consumption difference between the accounting run and the original run is only 3%.

3 CHAPPIE DESIGN

In this section, we provide a high-level specification for Chappie runtime and metric design.

3.1 Runtime Design

On the top level, the Chappie runtime is specified by the ChappieRuntime function in Algorithm 1. Here, Chappie continuously samples raw energy consumption (Line 12) and JVM stack information (Line 14) and combines the two to produce an *attribution* (Line 17), i.e., how each logical unit of the monitored application (such as a method) may be assigned with a portion of the energy consumption reading obtained from the underlying system. Structurally, each attribution $\mathcal{A}$ is a mapping from a logical application unit LUnit to its share of energy consumption EType which is an abstract representation of joules. Notation $\emptyset$ represents an empty map.

Algorithm 1 Chappie Sampling

```
1  typedef Tid Int // thread ID
2  typedef Etype Float // energy in joules
3  typedef Lunit String ∪ {⊥} // accounting logical unit
4  EPOCH : Int // VM sampling interval
5  A : Map<Lunit,Etype> // Output attribution
6  T : Set<Tid> // VM threads
7  V : Map<Tid,Lunit> // thread-indexed logical units
8  function ChappieRuntime
9   A ← ∅
10  loop at rate (EPOCH)
11   V ← ∅
12   ϵ ← ESample()
13   for t in T do
14    V[t] ← Abstract(Stack(t))
15   α = |{t|V[t] ≠ ⊥}|
16   for v in V and v ≠ ⊥ do
17    A[v] ← A[v] + ϵ × 1/α
18 function ESample : EType // obtain energy reading
19 function Stack(Tid) : Stack // obtain thread stack
20 function Abstract(Stack) : LUnit // transform stack to logical unit
```

The goal of each JVM sampling step (Line 14) is to obtain an abstract representation (the Abstract function) of the runtime stack of the running threads (the Stack function). The latter returns either the runtime stack frame information if a thread is active, or empty otherwise, with $\textsc{Abstract}(\emptyset) = \bot$. Given an energy sample ϵ, the algorithm first counts the number of active threads α, and each active thread — and its associated logical unit — will be attributed with a fraction $\frac{1}{\alpha}$ of ϵ. The share of energy attributed to each logical unit is accumulated in $\mathcal{A}$, at Line 17.

Chappie features an extensible and customizable design. Depending on how the Abstract function and the Lunit type are concretized, the algorithm can express a variety of granularities in application-level energy accounting. Our default implementation supports energy accounting over *deep application methods*, which assigns energy consumption to an application (i.e., non-library) method when either the said method is at the stack top, or it is the calling context to a library method that is at the stack top. Chappie allows users to customize the Abstract function. Currently, additional versions have been implemented for *context-sensitive* method energy accounting, and class/package/thread energy accounting (see §5).

3.2 Metric Design

We now provide a rigorous definition for calmness. Its essence lies upon the similarity between the runtime characteristics of an accounting run against a *reference run*, i.e., the original application run when the accounting system is not at work. To simplify the matter, we first consider the idealized case where one instance of application execution is sufficient to capture the runtime characteristics. Let $\mathbb{EPOCH}$ represent the set of epochs in the form of $\mathbb{N}^+$, with the first epoch of each run starting at epoch 1. Let $\mathbb{CORE}$ the set of CPU cores and $\mathbb{FREQ}$ the set of observable CPU frequencies. Each run

benchmark	workload	methods	total threads	active threads	execution time (s)
avrora	large	576	71	69	175.98
batik	large	924	9	8	17.41
biojava	default	103	7	6	22.76
eclipse	default	5423	706	18	59.14
graphchi	huge	124	53	50	246.78
h2	large	3676	954	39	115.91
jython	default	1426	7	6	10.63
pmd	large	950	8	6	54.65
sunflow	large	257	88	46	61.13
tomcat	large	3214	109	106	27.0
xalan	default	1071	47	46	9.0

Figure 5: Benchmark Statistics (Workload refers to the data size specified by DaCapo for each benchmark. Methods shows the number of unique methods appeared in the trace. Total threads shows the number of the threads created throughout the lifetime of the application. Active threads shows the maximum number of the concurrent threads at any epoch.)

benchmark	rate	batches	PCC	SE	RMSE
avrora	64	6	0.9944	0.0145	0.0027
batik	8	2	0.9998	0.001	0.0005
biojava	128	3	0.9968	0.0139	0.022
eclipse	16	4	0.995	0.0019	0.0002
graphchi	16	5	0.9981	0.0089	0.0066
h2	32	2	0.9977	0.0046	0.0013
jython	32	2	0.998	0.0027	0.0005
pmd	16	2	0.9922	0.0079	0.0024
sunflow	64	2	0.9945	0.0088	0.004
tomcat	16	2	0.9999	0.0007	0.0004
xalan	16	2	0.9994	0.0013	0.0005

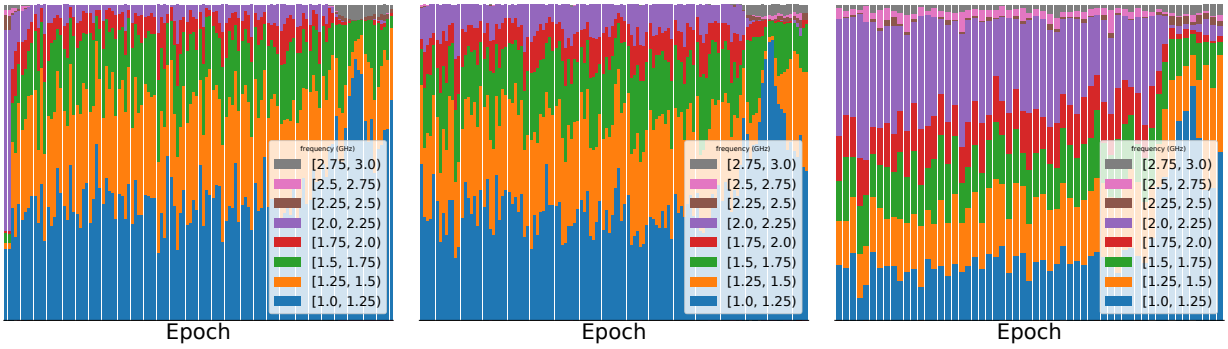
Figure 6: Accounting Parameters (Rate refers to the sampling rate for each benchmark. Batches refers to the number of data collection runs for the accounting of each benchmark. PCC shows the correlation between the energy footprint produced from (n-1) batches and that produced from n batches. SE shows the standard error of the PCC. RMSE shows the root means square error.)

produces a set of samples $S \in \mathcal{P}(\mathbb{EPOCH} \times \mathbb{CORE} \times \mathbb{FREQ})$, where each sample $\langle e; c; f\rangle$ intuitively says the CPU frequency of core c at epoch e is f. First, let us introduce some auxiliary functions.

Definition 3.1 (Epoch Count). We say a run with samples S consists of $eCount(S)$ epochs where $eCount(S) \triangleq \frac{|S|}{|\mathbb{CORE}|}$.

Definition 3.2 (Core Count). We say there are $cCount(S, e, f)$ number of cores operating at frequency f and epoch e. where $cCount(S, e, f) \triangleq |\{c|\langle e; c; f\rangle \in S\}|$. From now on, we use metavariable $m \in [0..|\mathbb{CORE}|]$ to represent the core count.

To study temporal calmness, we intuitively wish to characterize how the power consumption — manifested by CPU frequencies — "flows and ebbs" over time. We first introduce a function for computing the CPU frequencies given a particular epoch e in a particular run. Observe that in our sample space, there are $|\mathbb{CORE}|$ number of samples for each epoch, one from each core. Rather than assuming a fixed distribution, we represent the frequencies as a *distribution* to preserve generality. Specifically,



(a) `no method sampling` (b) `32ms method sampling` (c) `1ms method sampling`

Figure 7: Temporal Distribution of CPU Core Frequency for `h2` (The X-Axis represents the epoch series of the benchmark execution, with the first epoch and the last epoch indicates the beginning and end of execution. Each bar represents a particular frequency range, whose height indicates the normalized number of CPU cores at that frequency at that epoch. For illustration, we divide all frequencies into 8 ranges. For example, throughout the majority of (a), the orange bars show that 25% of cores execute between 1.25 - 1.5 GHz, until the tail end, where 5% of cores execute at that range.)

Definition 3.3 (Temporal Characterization). We use $TC(S)$ to compute the *temporal characterization* for a run over samples S. $TC(S)$ computes to an element in $\mathbb{EPOCH} \mapsto (\mathbb{FREQ} \mapsto [0,1])$, i.e., from each epoch to a *frequency distribution* . Formally, for each $e \in eCount(S)$ and each $f \in \mathbb{FREQ}$, $TC(S)(e)(f) \triangleq \frac{cCount(S,e,f)}{|\mathbb{CORE}|}$.

Indeed, if we treat the frequency at epoch e in the sample space S as a random variable, $TC(S)(e)$ is its probability mass function (PMF).

To study spatial calmness, we intuitively wish to characterize how CPU frequencies "spread out" across cores. An intuitive representation is to show which cores operate on each frequency. This intuition carries some subtlety. First, there are $eCount(S)$ number of snapshots for the characterization of spreading out to be considered. The overall spatial characterization throughout an entire run can be intuitively viewed as each epoch introducing an observation characterizing how CPU frequencies are spread out for that epoch. Second, scheduling is fundamentally non-deterministic and threads may migrate from one another. For designing a comparative metric like ours, this means that a comparative study for a fixed core's behavior is meaningless. What matters is *how many* cores operate at a particular CPU frequency. With these two elaborations, we define:

Definition 3.4 (Spatial Characterization). We use $SC(S)$ to compute the *spatial characterization* for a run over samples S. $SC(S)$ computes to an element in $\mathbb{FREQ} \times \mathbb{CCOUNT} \mapsto [0,1]$, i.e., a *frequency-coreCount bivariate distribution*, defined as $SC(S)(f,m) \triangleq \frac{|\{e | m = cCount(S,e,f)|\}|}{eCount(S)}$.

Indeed, if we treat both the frequency and the core count in the sample space S as a random variable, $SC(S)$ is their bivariate PMF.

Definition 3.5 (Calmness). We say an accounting run with samples S is calm relative to a reference run with samples S_0 iff:

- [Time Correspondence] $eCount(S) \approx eCount(S_0)$
- [Temporal Correspondence] for any e such that $e <= \mathbf{min}(eCount(S), eCount(S_0))$, $TC(S)(e) \backsim TC(S_0)(e)$
- [Spatial Correspondence] $SC(S) \backsim SC(S_0)$

where $\approx$ and $\backsim$ abstractly represent the similarity between two natural numbers and two distribution respectively, which we will contretize in §4.

Time correspondence captures that the accounting run must have similar execution time as the reference run. The temporal correspondence and spatial correspondence enforce that the two runs must have similar power characteristics. With energy being accumulated power over time, the criteria above together say that the energy characteristics between the accounting run and reference run are similar, the essence of *calm energy accounting*.

Multi-Iteration Runs. The discussion so far has idealistically assumed that there is only *one* reference run and *one* accounting run. As most experiments are repeated to take into the fundamental non-determinism in the software-hardware stack, the reference run (as well as the accounting run) may consist of multiple executions of the application, each of which we call an *iteration*. Our calmness metric can be defined for multi-iteration runs in a nearly identical manner as Definition 3.5, with small changes. Let us assume a k-iteration run produces samples $S_1, \ldots, S_k$ respectively, we can follow Definition 3.5 to construct the calmness metric over samples $S_1 \cup \cdots \cup S_k$, with $|\mathbb{CORE}|$ redefined as the number of CPUs multiplied by k. Intuitively, this implies we can conceptually view the k-iteration run as one parallel run of k instances of the application over k times of physical CPUs. This formal view simplifies our algorithm specification: we do not need to repeat all definitions we introduced earlier for multi-iterations runs. Practically, this means we can merge all the samples we collected from different iterations by reusing the formal definition we have given for a single iteration. For example, Fig. 8 shows the equivalent view of a three-iteration run on a two-core machine (with `CPU1` and `CPU2`) of an application

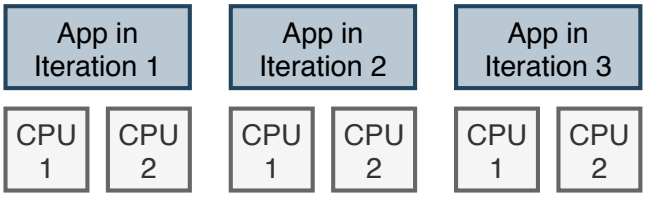


Figure 8: A Multi-Iteration Run

app. Since both our reference run and profile run consist of multi-iterations in our experiments, this view has been adopted in our data analysis over calmness, to be reported in the next sections.

4 CHAPPIE IMPLEMENTATION

Thread Model Overview. We implement the Chappie runtime as a thread of its own, in the same JVM runtime as the application being accounted for. The top-level loop construct in the specification is implemented with a timer. The thread periodically wakes up, with no busy waiting.

The global structure $\mathcal{T}$ in the specification requires thread-safe access. To prevent it from becoming a bottleneck, we implement it with a *delayed buffer*. Whenever a thread is started (or exited), an entry is added (or removed) from this delayed buffer to indicate the change. Periodically at every epoch, the thread that runs the Chappie runtime will retrieve from the buffer and apply the changes — adding a thread or removing a thread — to its exclusively held $\mathcal{T}$ structure. As a result, $\mathcal{T}$ does not require synchronization, and only the delayed buffer is implemented as a synchronized `Queue`.

Call Stack Sampling. Traditional approaches for accessing the thread stacks — such as through the `Thread.getStackTrace` — incur significant overhead. To circumvent this restriction, we resort to a Hotspot VM API, the `AsyncGetCallTrace` method, to sample the call stack, a method also used by popular profilers such as the async profiler [2], which our implementation builds on. To integrate with the async tool which supports asynchronous stack sampling, we maintain a buffer to keep the stack samples, and use timestamps to align them into each epoch.

The async profiler can sample both Java stacks and the native stacks — the latter may result from either though JNI invocation or JIT compilation — and Chappie can handle both as a result.

Hardware Energy Reading and CPU Affinity. We rely on Intel's RAPL [18] interface to obtain energy samples at the granularity of CPU power domains (sockets). Energy samples were collected using jRAPL [36], a Java library for interfacing with RAPL. We are able to sample both CPU package and DRAM energy. As our experimental platform consist of multiple sockets, we treat each socket as a separate locale for attribution (Algorithm 1) and combine data together. As the OS scheduler may migrate a thread from one socket to another, we maintain the CPU affinity information to keep track of the socket — and hence the locale of attribution — a thread belongs to.

Metric Implementation. Predicate $e \approx e'$ in Definition 3.5 is implemented as the normalized difference between e and e' is less than 5%. The similarity between distributions ($\backsim$) is implemented through the standard metric of Pearson's Correlation Coefficient (PCC) between the pair of distributions. Two distributions are considered similar iff their PCC > 0.85. In general, PCC above 0.7 is considered strongly correlated.

The *TC* and *SC* functions both use PMFs to represent the distributions. Although modern CPUs only publish a small number of CPU frequencies, the observed CPU frequencies are much more diverse. For example, in our experiments, we observed 225 distinct frequencies in our data samples. This implies that if the original PMF is used, there are a large number of elements in the vector for PCC computation. This is a well-known problem, and we have applied Freedman-Diaconis rule to bin similar frequencies together.

Temporal correspondence in Definition 3.5 relies on epoch-wise distribution similarity. While time correspondence establishes that the number of epochs for the reference run and those for the accounting run are similar, a small difference may still exist. Definition 3.5 takes the approach of only considering the less number of the epochs between the two runs. In practice, we found that if an accounting run is (slightly) slower than the reference run, the delayed effect w.r.t. the frequency behavior often exhibits gradually as time goes on. To capture this gradual shift, we use a simple interpolation approach to make the two runs match on epochs: if the number of epochs for the reference/accounting run is e_0 and e_0' respectively, we adjust each raw sample in the accounting run $\langle e; c; f\rangle$ to $\langle e'; c; f\rangle$ where $e' = e \times \frac{e_0'}{e_0}$. Since e_0 and e_0' are similar, this adjustment only affects a small portion of samples.

5 CHAPPIE EVALUATION

5.1 Evaluation Methodology

We evaluated Chappie on a dual socket Intel E5-2630 v4 2.20 GHz CPU server, with 10 cores in each socket and 64 DDR4 RAM. Hyper-threading is enabled. The machine runs Debian 4.9 OS, Linux kernel 4.9, with the default Debian `powersave` governor. All experiments were run with Java 11 on top of Hotspot VM build 11.0.2+9-LTS. When each experiment is performed, the OS has no other workload.

Chappie is evaluated over the DaCapo benchmark suite [9], of its recent version evaluation-git+8b7a2dc, released in June 2019. This release includes state-of-the-art workloads such as `graphchi` and `biojava`. As long(er)-running applications are more interesting w.r.t. energy management, we focus on benchmarks whose execution time is around or above 10 seconds. Among the 13 benchmarks that fit into this criterion, we excluded 2 of them: the majority of application code of `tradesoap` and `tradebeans` is run in a new *process* — as opposed to a new *thread* — whose accounting Chappie currently does not support without modifying benchmark source code.

The statistics of the benchmarks are shown in Figure 5. Here the execution time refers to average the benchmark running time over all of its iterations which we discuss shortly. The DaCapo harness setup time is excluded. The benchmarks are realistic Java applications with thousands of methods and diverse characteristics in multi-threading, both in terms of total created threads and concurrently running threads.

Figure 6 summarizes the parameter settings used for the accounting of each benchmark. Recall that calm energy accounting requires

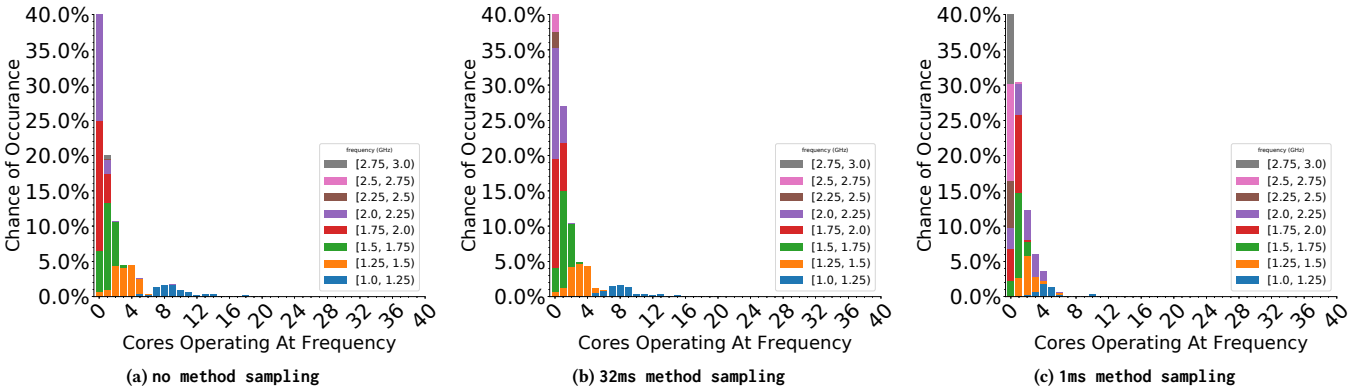


Figure 9: Spatial Distribution of CPU Core Frequency for h2 (The X-Axis shows the number of cores observing the same frequency in the same epoch. The Y-Axis shows the probability that a frequency is observed. For illustration, we divide all frequencies into 4 ranges. For example, the red bar on the 3rd tick on (c) says that there is a 15% chance that 3 cores run in frequency range 2.5 - 3.0+ GHz at the same time. The sum of heights of all bars equals to 1.

minimal disturbance to the original benchmark, Chappie sets a distinct sampling rate (as defined by EPOCH in Algorithm 1) for each benchmark to ensure calmness, shown in the rate column. The section of this rate will be the focus of §5.2. To gain confidence in the precision of the energy footprints, Chappie relies on a combination of cold/hot executions to collect data: each *batch* is a distinct JVM instance that subsumes *iterations*, i.e., hot execution instances of a benchmark. We elaborate on the number of batches in Section 5.4. With each batch, we perform 10 iterations by default and follow the standard practice of discarding the data from the first 2 iterations. As exceptions, we perform 30 iterations (and discard the first 2) for sunflow and jython, because of the relatively large standard deviation of their execution time (>10%) if fewer iterations are used for data collection.

5.2 Achieving Calmness

In this section, we study how the calmness metric is used for determining the judicious sampling rate for each benchmark. Before we present the result for all benchmarks, we begin with a visual elaboration on temporal correspondence and spatial correspondence, the cornerstones of calmness.

Figure 7 shows the temporal behavior of h2 under 3 different settings: the reference run, the Chappie-accounting run under the selected sampling rate 32ms, and the Chappie-accounting run if the sampling rate were set at 1ms. Within each sub-figure, time elapses from left to right. Its time-dependent variation is an illustration of power phased behaviors. As one can see, the shape of the 32ms-sampling result is significantly more similar to that of the reference run, than the 1ms-sampling result. The temporal correspondence definition in Def. 3.5 is intended to capture the similarity of the former (and the disimilarity of the latter). One interesting observation is that the 1ms-sampling result shows more CPU cores are likely to be driven into a higher frequency; this is consistent with our intuition that a higher sampling rate may have significantly increased activity in CPU cores, rendering them into higher frequencies due to DVFS. Finally, observe that Figure 7(c) consists of fewer epochs, because the run completes significantly faster than the reference run.

Figure 7 shows the spatial behavior of h2 under the same 3 different settings as earlier. Here, we care about how a frequency "spreads out" across cores. If all cores have the same frequency at the same time, it indicates a "perfectly balanced" power consumption across cores. Within each sub-figure, this is indicated by the rightmost point on the X-axis. As one can see, the shape of the 32ms-sampling result is again significantly more similar to that of the reference run, than the 1ms-sampling result. The spatial correspondence definition in Def. 3.5 is intended to capture the similarity of the former (and the disimilarity of the latter). Specific to h2, the relative unbalancedness is not surprising: h2 as an in-memory database is known to be an I/O-intensive benchmark with a low degree of parallelism.

Finally, Figure 10 summarizes the 3 components used to determine calmness: time overhead, temporal correspondence, and spatial correspondence across all benchmarks. The full results including standard deviation are deferred to the Appendix. If a data point is missing, it means the data is out of the range (of our interest). Based on our profiling calmness studies, we have selected the sampling rate for each benchmark with results shown in the second column of Figure 6.

5.3 Producing Energy Footprints

Method-Grained Accounting. The energy footprint reported by Chappie can be of various logical units of abstraction. The default unit, i.e., each entry in the energy footprint, is the *deep application method*, as discussed in §3. In our reported data, a method is viewed as a library method if it belongs to a class whose qualified name starts with java, jdk, sun, and apache.commons. A method belonging to any additional third-party library is treated as an application method.

Fig. 11 illustrates a portion of this footprint — the top-10 energy-consuming methods — for graphchi, a concrete instance of the

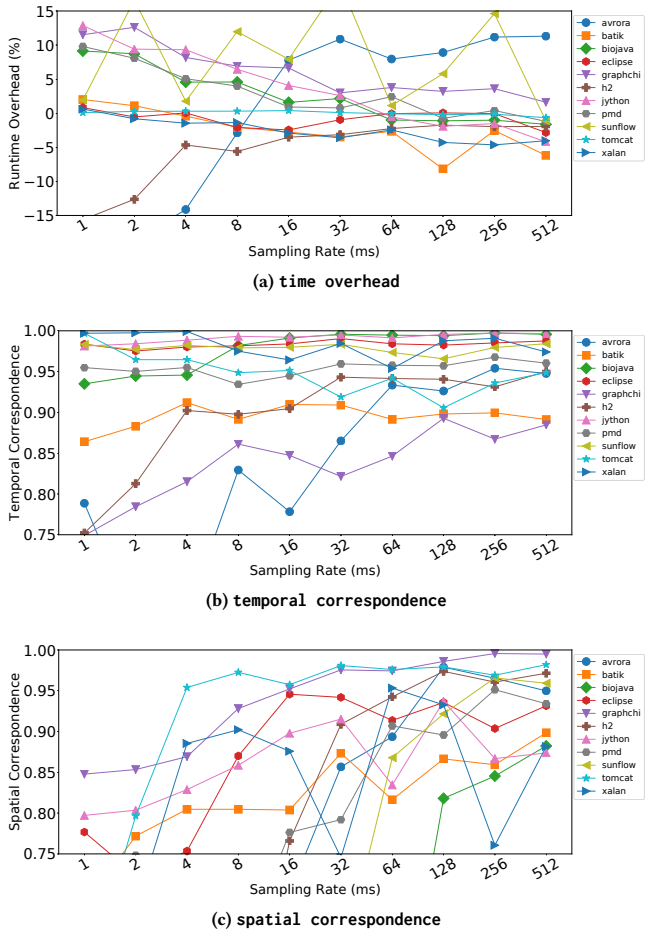


Figure 10: A Summary of Benchmark Calmness (For both sub figures, each line indicates a benchmark and the axis indicates the method sampling rate.)

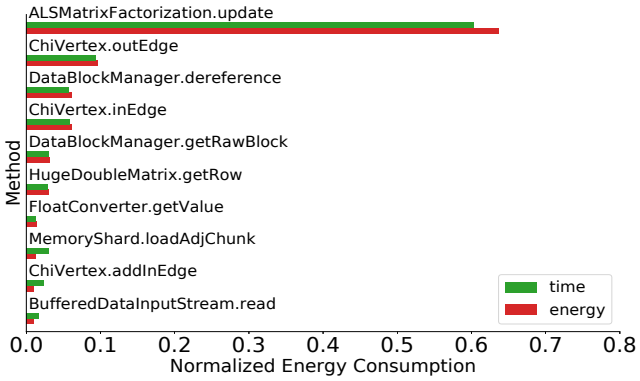


Figure 11: Top 10 Energy-Consuming Application Methods for `graphchi` (Each green/red bar indicates the normalized energy/time of a top consuming method. Energy is directly computed by Chappie according to our algorithm specification. Time is approximated by the number of samples multiplied by the length of the sampling interval.)

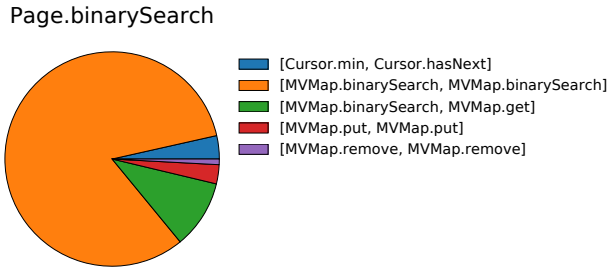


Figure 12: Context-Sensitive Method Accounting for top 10 energy-consuming method of `h2`. (Each slice with context [X,Y] indicates that a method is called by Y which in turn is called by X. Note that one of the calling context is recursive.)

namesake data-intensive graph processing system [35]. The DaCapo's benchmark implements ALS matrix factorization, an iterative algorithm with graph traversal and updates. The energy footprint generated by Chappie corroborates the nature of this benchmark, with the top ranked method being `ALSMatrixFactorization.update`. The next three highest-ranked methods are related to graph traversal, with methods `ChiVertex.outEdge` and `ChiVertex.inEdge` for accessing the edges of a vertex, and `DataBlockManager.dereference` for fetching the value associated with an edge.

It is interesting to observe that energy consumption and execution time do not always correspond. In `graphchi`, method `ALSMatrixFactorization.update` has a higher normalized energy consumption than its execution time, indicating that the system is in a higher-power state. This method is mathematical in nature, and our results can be intuitively explained through a well-known phenomenon: with default governors, compute-intensive workloads often lead the CPU to a higher-power state. As a counter example, `MemoryShard.loadAdjChunk` has a lower power, consistent with the fact that this method is I/O-intensive.

Calling Context-Grained Accounting. Through providing different concrete Abstract functions (see Section 3), Chappie is a general framework that can be customized to account for programming abstractions at different levels of granularity. For example, Chappie can report method energy consumption in a context-sensitive manner, i.e., accounting for different calling contexts separately. Fig. 12 provides a finer-grained view into two top consuming methods for `h2`, an in-memory database. Our results show that the majority of energy consumption for `Page.binarySearch` comes from the (recursive) calling context of `MVMap.binarySearch` which aligns with our understanding of binary search algorithms. This example shows that Chappie at the context-sensitive granularity provides additional context that paints a fine-grained picture for understanding the energy behavior of `h2`.

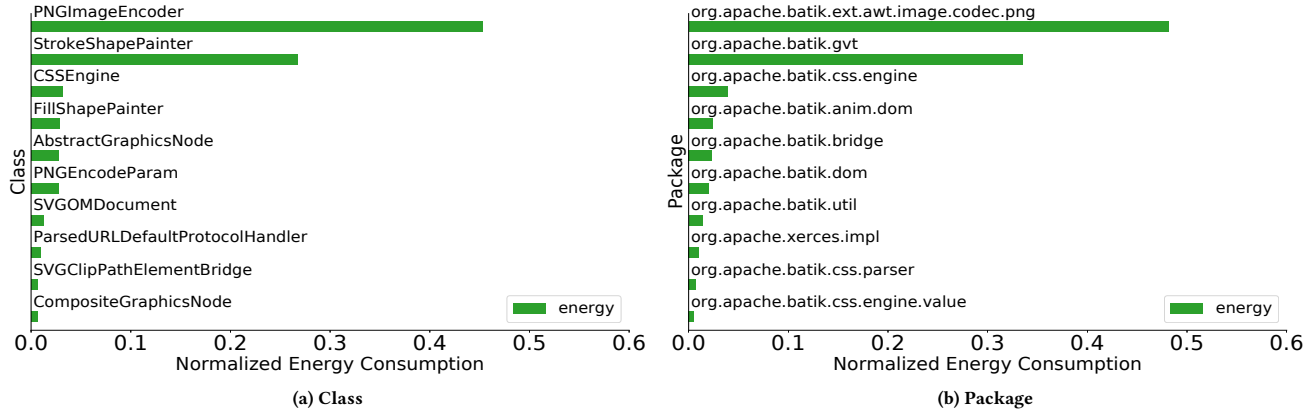


Figure 13: Top 10 Energy-Consuming Application Classes and Packages (Each green normalized energy of a top consuming class or package.)

Class- and Package-Grained Accounting. Alternatively, Chappie can be customized with an Abstract for Java class- and package-level granularity. Figure 13 shows the result for `batik`, an Apache toolkit for transforming and rendering Scalable Vector Graphics (SVG) [1]. The DaCapo benchmark focuses on the use scenarios of transforming SVG files into Portable Network Graphic (PNG) images and rendering them. Our results show that `PNGImageEncoder` and `StrokeShapePainter` classes are responsible for a majority of `batik` energy consumption, which happens to be aligned with the main features of this application: transformation and rendering. As classes are often the abstraction units for dividing programming tasks among developers in large-scale software development, Chappie class-level energy footprint provides insight on which programmers are the critical link on developing energy-conscious software. One granularity coarser, Chappie can further demonstrate energy consumption at the package level. For `batik`, half of its energy consumption results from PNG codec, as shown in the package `org.apache.batik.ext.awt.image.codec.png`.

5.4 Overhead and Precision

As a disturbance-mitigated approach, Chappie is fundamentally overhead-averse: it selects a sampling rate only when calmness is achieved, leading to minimal time overhead (according to time correspondence in Def. 3.5) and minimal power/energy overhead (according to temporal/spatial correspondence in Def. 3.5). For all benchmarks operating at the selected sampling rate, the average time overhead is 3.15±3.03% and the average energy overhead is 0.84±1.09%.

With multi-batch runs as part of the design, Chappie is also constructed with precision as an inherent goal. Intuitively, the ground truth of energy consumption of a method is the accumulation of the energy consumption from all of its instructions. This is equivalent to a sampling-based approach where the number of samples approaches infinity. One standard metric to study the approximation to the ground truth is the convergence of results, i.e., whether introducing more samples may significantly change the results.

To achieve this goal, we study the extent that the energy footprint may change when data from an additional batch are introduced. Intuitively, if introducing additional batches of samples can lead to little change in the energy footprint, convergence is achieved. We compute the PCC between the data of `n-1` batches and that of `n` batches, and set the batch parameter for the benchmark as `n` if the PCC is greater than 0.99. The `batches` column of Figure 6 shows the batch setting for each benchmark. Most benchmarks require only 2 batches — the minimum number in a relational approach we take — to achieve PCC > 0.99. The remaining benchmarks, e.g., `avrora`, exhibit higher variability, but observe that each still converges to our high PCC requirement within a limited number of batches.

6 RELATED WORK

Energy accounting is a classic problem at lower layers of the computing stack. Examples include iCount [20] at the digital circuit level and Currentcy [52] at the OS level. With the primary goal of attributing a global energy budget to individual components, totality is implicit in energy accounting solutions. This paper is a systematic study of bringing energy accounting to the application level, where the individual components at concern are methods, calling contexts, classes, and packages.

At the application level, energy accounting and energy profiling overlap in their overall goal of characterizing the runtime characteristics of an application. While accounting is implicitly total, profiling may or may not. This is why instrumentation remains a viable approach in existing energy profilers as they many choose to study the runtime characteristics of individual logical units one by one. This approach is particularly common in empirical studies, where the energy consumption of specific program features is reported based on instrumenting such features. With a feature focus — e.g., the use of the concurrent collections API [25, 45], or the impact of data access patterns [36] — instrumentation can be a feasible solution as it can be placed for one code block at a time. In this use scenario, Chappie may be useful in improving the precision of profiling by making the profiler concurrency-aware. Eprof [41] accounts for smartphone energy consumption through tracking

I/O system calls and pre-defined SDK-NDK routines, producing a breakdown on important smartphone use scenarios related to the use of 3G network, screen, *etc.* E-Android [23] is a profiler that detects Android collateral energy consumption through tracing a set of pre-defined energy-critical events. These energy profilers may profile pre-defined application components, such as the software component for 3G network interaction or screen tracking, but their designs are geared toward physical components of the platform, without a full account of general logical units as Chappie does. Bokhari et al. [10] defines a conceptual energy profiling framework for resampling in the presence of measurement inaccuracy. Earlier energy profilers such as JouleTrack [48] and Powerscope [22] focus on measurement framework design, addressing *e.g.*, how to provide high-rate energy samples and how to synchronize the execution with the measurements.

Another key feature unique to Chappie is its systematic investigation into power disturbance. We are unaware of existing energy profilers that provide metrics to quantify and overcome it. In non-energy profiler design, that a profiler may intervene and alter the behavior of the original application is a basic fact, motivating designers to reduce the overhead introduced by the profiler. For non-energy profilers, say a memory consumption profiler, the effect of power disturbance — if any — is of a lesser concern. This is in contrast with energy profiling, power disturbance may alter the very characteristic the profiler intends to capture. Sampling-based non-energy profilers are standard [28, 51, 54].

One area Chappie may positively impact on is application-level energy management and optimization, with solutions ranging from energy-aware programming languages and energy-adaptive frameworks. Several examples may demonstrate this synergy. Green [5] relies on online energy accounting to perform QoS calibration. PowerDial [30] and JouleGuard [29] may use energy feedback to make control decisions. LAB [34] needs to continuously account for energy consumption to balance latency, accuracy, and battery. Eco [53] must track energy consumption to match the application demand and the system resource supply. Aeneas [14] relies on online energy readings to enable energy optimization guided by reinforcement learning. With Chappie, these application-level energy efforts can gracefully extend to the more complex use scenarios where the application may be multi-threaded, and multiple applications may co-exist. Another area Chappie may provide essential support for is energy testing and debugging, an emerging research direction [6, 7, 12, 23, 27, 38, 40, 42, 46]. As energy and performance often go hand in hand, this direction may unify with performance bug studies [4, 17, 33, 44] to provide comprehensive software lifecycle support for software non-functional properties.

The dual of energy accounting is energy analysis, a bottom-up approach to determine the energy consumption of a program based on its building blocks. With a bottom-up design principle, these approaches are fine-grained by design. Instruction-level power analysis [50] may associate instructions with power profiles. Additional designs exist to perform energy analysis in a WCET-like setting [32], on the bytecode level [26], and the LLVM IR level [24] for instance. Energy analysis has a nearly orthogonal interest in illuminating program energy consumption to ours, and the two approaches may follow a classic duality in software research, reasoning vs. monitoring. Practically, it is unclear how related work adapt to scenarios with multi-threaded executions.

name	rate									
	1	2	4	8	16	32	64	128	256	512
avrora	-28.47±5.99%	-18.67±2.50%	-14.13±1.33%	-2.88±0.18%	7.78±0.48%	10.87±0.73%	7.95±0.76%	8.91±0.64%	11.17±0.86%	11.31±0.83%
batik	2.01±0.07%	1.10±0.05%	-0.42±0.03%	-2.06±0.08%	-2.76±0.15%	-3.48±0.16%	-2.65±0.11%	-8.17±0.31%	-2.54±0.10%	-6.18±0.25%
biojava	9.13±0.28%	8.72±0.32%	4.56±0.14%	4.58±0.15%	1.59±0.07%	2.16±0.07%	-1.00±0.04%	-1.11±0.04%	-1.02±0.03%	-1.63±0.05%
eclipse	0.82±0.02%	-0.55±0.01%	0.04±0.00%	-2.09±0.03%	-2.45±0.05%	-0.96±0.03%	-0.06±0.00%	0.05±0.00%	-0.06±0.00%	-2.84±0.06%
graphchi	11.52±0.11%	12.61±0.13%	8.18±0.10%	6.92±0.07%	6.63±0.06%	2.99±0.03%	3.77±0.04%	3.21±0.03%	3.60±0.03%	1.62±0.02%
h2	-15.63±0.25%	-12.63±0.25%	-4.66±0.05%	-5.61±0.09%	-3.51±0.05%	-3.15±0.05%	-2.24±0.02%	-1.75±0.02%	-1.96±0.02%	-1.93±0.02%
jython	12.86±2.55%	9.40±1.85%	9.32±1.78%	6.44±1.25%	4.07±0.80%	2.59±0.50%	-0.53±0.10%	-1.91±0.36%	-1.51±0.29%	-4.20±0.82%
pmd	9.79±0.38%	8.06±0.37%	5.07±0.15%	3.98±0.11%	0.92±0.02%	0.78±0.01%	2.43±0.05%	-0.75±0.01%	0.39±0.01%	-1.20±0.03%
sunflow	1.94±0.45%	16.58±3.53%	1.76±0.41%	11.96±2.86%	7.93±1.82%	19.05±4.02%	1.12±0.25%	5.79±1.33%	14.62±3.06%	-1.12±0.25%
tomcat	0.10±0.00%	0.26±0.00%	0.29±0.00%	0.30±0.00%	0.39±0.00%	0.10±0.00%	-0.08±0.00%	-0.21±0.00%	-0.13±0.00%	-0.66±0.01%
xalan	0.61±0.02%	-0.81±0.03%	-1.46±0.06%	-1.37±0.05%	-2.91±0.11%	-3.61±0.14%	-2.44±0.09%	-4.29±0.16%	-4.64±0.19%	-4.04±0.17%

(a) Overhead

name	rate									
	1	2	4	8	16	32	64	128	256	512
avrora	0.79±0.01	0.64±0.01	0.65±0.01	0.83±0.01	0.78±0.01	0.87±0.01	0.93±0.01	0.93±0.01	0.95±0.00	0.95±0.01
batik	0.86±0.03	0.88±0.02	0.91±0.02	0.89±0.02	0.91±0.02	0.91±0.02	0.89±0.02	0.90±0.02	0.90±0.02	0.89±0.02
biojava	0.93±0.01	0.94±0.01	0.95±0.01	0.98±0.01	0.99±0.00	1.00±0.00	0.99±0.00	0.99±0.00	1.00±0.00	1.00±0.00
eclipse	0.98±0.01	0.98±0.01	0.98±0.01	0.98±0.01	0.98±0.00	0.99±0.00	0.98±0.00	0.98±0.00	0.99±0.00	0.99±0.00
graphchi	0.75±0.01	0.78±0.01	0.82±0.01	0.86±0.01	0.85±0.01	0.82±0.01	0.85±0.01	0.89±0.00	0.87±0.01	0.88±0.00
h2	0.75±0.02	0.81±0.01	0.90±0.01	0.90±0.01	0.90±0.01	0.94±0.01	0.94±0.01	0.94±0.01	0.93±0.01	0.95±0.01
jython	0.98±0.01	0.98±0.00	0.99±0.00	0.99±0.00	0.99±0.00	0.99±0.00	0.99±0.00	1.00±0.00	1.00±0.00	1.00±0.00
pmd	0.95±0.01	0.95±0.01	0.96±0.01	0.93±0.01	0.94±0.01	0.96±0.01	0.96±0.01	0.96±0.01	0.97±0.01	0.96±0.01
sunflow	0.98±0.00	0.98±0.00	0.98±0.00	0.98±0.00	0.98±0.00	0.98±0.00	0.97±0.01	0.97±0.01	0.98±0.01	0.98±0.01
tomcat	1.00±0.01	0.96±0.03	0.96±0.02	0.95±0.02	0.95±0.02	0.92±0.03	0.94±0.02	0.91±0.03	0.94±0.02	0.95±0.02
xalan	1.00±0.02	1.00±0.01	1.00±0.01	0.98±0.03	0.96±0.04	0.98±0.02	0.95±0.03	0.99±0.02	0.99±0.01	0.97±0.02

(b) Temporal Correspondence

name	rate									
	1	2	4	8	16	32	64	128	256	512
avrora	0.08±0.12	-0.03±0.13	-0.27±0.12	0.10±0.13	0.70±0.10	0.86±0.07	0.89±0.06	0.98±0.02	0.97±0.04	0.95±0.04
batik	0.69±0.10	0.77±0.08	0.80±0.08	0.80±0.08	0.80±0.08	0.87±0.06	0.82±0.07	0.87±0.06	0.86±0.06	0.90±0.05
biojava	0.32±0.13	0.33±0.14	0.41±0.14	0.36±0.16	0.35±0.17	0.59±0.14	0.46±0.16	0.82±0.10	0.85±0.09	0.88±0.08
eclipse	0.78±0.10	0.72±0.11	0.75±0.10	0.87±0.07	0.95±0.05	0.94±0.05	0.91±0.06	0.94±0.05	0.90±0.06	0.93±0.05
graphchi	0.85±0.06	0.85±0.06	0.87±0.06	0.93±0.05	0.95±0.04	0.98±0.03	0.97±0.03	0.99±0.02	1.00±0.01	0.99±0.01
h2	0.04±0.12	0.16±0.12	0.50±0.10	0.47±0.11	0.77±0.08	0.91±0.05	0.94±0.04	0.97±0.03	0.96±0.04	0.97±0.03
jython	0.80±0.09	0.80±0.09	0.83±0.08	0.86±0.07	0.90±0.07	0.92±0.07	0.83±0.09	0.94±0.06	0.87±0.09	0.87±0.09
pmd	0.60±0.11	0.75±0.08	0.69±0.09	0.62±0.09	0.78±0.08	0.79±0.07	0.91±0.05	0.90±0.05	0.95±0.04	0.93±0.04
sunflow	0.48±0.12	0.46±0.13	0.39±0.13	0.49±0.13	0.47±0.13	0.61±0.13	0.87±0.08	0.92±0.07	0.97±0.06	0.96±0.06
tomcat	0.58±0.18	0.80±0.11	0.95±0.05	0.97±0.04	0.96±0.04	0.98±0.03	0.98±0.03	0.98±0.03	0.97±0.04	0.98±0.03
xalan	0.68±0.20	0.68±0.17	0.89±0.12	0.90±0.10	0.88±0.12	0.75±0.18	0.95±0.06	0.93±0.08	0.76±0.16	0.88±0.09

(c) Spatial Correspondence

Figure 14: Calmness Statistics (Rate refers to the sampling rate.)

7 CONCLUSION

Chappie is a novel runtime design for application-level energy accounting of multi-threaded Java applications with calmness as a new metric to quantify power disturbance in energy accounting. The project repository contains all data for all benchmarks, and a report for additional figures covering all benchmarks.

ACKNOWLEDGMENTS

We thank Doug Lea and Aleksey Shipilev for useful discussions on the `AsyncGetCallTrace` functionality and the honest profiler. This project is supported by US NSF CNS-1910532.

APPENDIX

The detailed data on runtime overhead, temporal correspondence, and spatial correspondence, with standard deviation information, are reported in Figure 14.

REFERENCES

[1] Apache batik,https://xmlgraphics.apache.org/batik.
[2] Async profiler, https://github.com/jvm-profiling-tools/async-profiler.
[3] h2 database engine, https://www.h2database.com/html/main.html.
[4] Alam, M. M. u., Liu, T., Zeng, G., and Muzahid, A. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), EuroSys '17, pp. 298–313.

[5] Baek, W., and Chilimbi, T. M. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI'10*, pp. 198–209.

[6] Banerjee, A., Chong, L. K., Ballabriga, C., and Roychoudhury, A. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Transactions on Software Engineering 44*, 5 (May 2018), 470–490.

[7] Banerjee, A., Chong, L. K., Chattopadhyay, S., and Roychoudhury, A. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, Association for Computing Machinery, p. 588–598.

[8] Bartenstein, T., and Liu, Y. D. Green streams for data-intensive software. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)* (May 2013).

[9] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*, pp. 169–190.

[10] Bokhari, M. A., Weng, L., Wagner, M., and Alexander, B. Mind the gap – a distributed framework for enabling energy optimisation on modern smartphones in the presence of noise, drift, and statistical insignificance. In *2019 IEEE Congress on Evolutionary Computation (CEC)* (2019), pp. 1330–1337.

[11] Boston, B., Sampson, A., Grossman, D., and Ceze, L. Probability type inference for flexibile approximate programming. In *OOPSLA '15*.

[12] Bruce, B. R., Petke, J., and Harman, M. Reducing energy consumption using genetic improvement. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015* (2015), pp. 1327–1334.

[13] Canino, A., and Liu, Y. D. Proactive and adaptive energy-aware programming with mixed typechecking. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017* (2017), pp. 217–232.

[14] Canino, A., Liu, Y. D., and Masuhara, H. 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* (2018), pp. 703–713.

[15] Chiba, S. Load-time structural reflection in java. In *Proceedings of the 14th European Conference on Object-Oriented Programming* (Berlin, Heidelberg, 2000), ECOOP '00, Springer-Verlag, pp. 313–336.

[16] Cohen, M., Zhu, H. S., Emgin, S. E., and Liu, Y. D. Energy types. In *OOPSLA '12*.

[17] Curtsinger, C., and Berger, E. D. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 184–197.

[18] David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. Rapl: Memory power estimation and capping. In *ISLPED '10*, pp. 189–194.

[19] Dhodapkar, A. S., and Smith, J. E. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture* (USA, 2002), ISCA '02, IEEE Computer Society, p. 233–244.

[20] Dutta, P., Feldmeier, M., Paradiso, J., and Culler, D. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)* (2008), pp. 283–294.

[21] Fei, Y., Zhong, L., and Jha, N. An energy-aware framework for coordinated dynamic software management in mobile computers. In *MASCOTS'04* (2004), pp. 306–317.

[22] Flinn, J., and Satyanarayanan, M. Powerscope: a tool for profiling the energy usage of mobile applications. In *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications* (1999), pp. 2–10.

[23] Gao, X., Liu, D., Liu, D., Wang, H., and Stavrou, A. E-android: A new energy profiling tool for smartphones. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (June 2017), pp. 492–502.

[24] Grech, N., Georgiou, K., Pallister, J., Kerrison, S., Morse, J., and Eder, K. Static analysis of energy consumption for llvm ir programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems* (2015), SCOPES '15, pp. 12–21.

[25] Gutiérrez, I. L. M., Pollock, L. L., and Clause, J. SEEDS: a software engineer's energy-optimization decision support framework. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014* (2014), pp. 503–514.

[26] Hao, S., Li, D., Halfond, W. G. J., and Govindan, R. Estimating android applications' cpu energy usage via bytecode profiling. In *Proceedings of the First International Workshop on Green and Sustainable Software* (2012), GREENS '12, pp. 1–7.

[27] Hao, S., Li, D., Halfond, W. G. J., and Govindan, R. Estimating mobile application energy consumption using program analysis. In *ICSE '13* (2013), pp. 92–101.

[28] Hirzel, M., and Chilimbi, T. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)* (2001), pp. 117–126.

[29] Hoffmann, H. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pp. 198–214.

[30] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. Dynamic knobs for responsive power-aware computing. In *ASPLOS '11*.

[31] Isci, C., and Martonosi, M. Identifying program power phase behavior using power vectors. In *In Workshop on Workload Characterization* (2003).

[32] Jayaseelan, R., Mitra, T., and Li, X. Estimating the worst-case energy consumption of embedded software. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)* (2006), pp. 81–90.

[33] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (2012), PLDI '12, pp. 77–88.

[34] Kansal, A., Saponas, S., Brush, A. B., McKinley, K. S., Mytkowicz, T., and Ziola, R. The latency, accuracy, and battery (lab) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA '13*, pp. 661–676.

[35] Kyrola, A., Blelloch, G. E., and Guestrin, C. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012* (2012), pp. 31–46.

[36] Liu, K., Pinto, G., and Liu, Y. D. Data-oriented characterization of application-level energy optimization. In *Proceedings of FASE 2015; JRAPL Home: http://kliu20.github.io/jRAPL/*.

[37] Lucia, B., and Ransford, B. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), PLDI '15, pp. 575–585.

[38] Ma, X., Huang, P., Jin, X., Wang, P., Park, S., Shen, D., Zhou, Y., Saul, L. K., and Voelker, G. M. edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), nsdi'13, pp. 57–70.

[39] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. Agile application-aware adaptation for mobility. pp. 276–287.

[40] Pathak, A., Hu, Y. C., and Zhang, M. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. HotNets-X '11, pp. 5:1–5:6.

[41] Pathak, A., Hu, Y. C., and Zhang, M. 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* (2012), EuroSys '12, pp. 29–42.

[42] Pathak, A., Jindal, A., Hu, Y. C., and Midkiff, S. P. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys '12*, pp. 267–280.

[43] Pering, T., Burd, T., and Brodersen, R. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design* (New York, NY, USA, 1998), ISLPED '98, Association for Computing Machinery, p. 76–81.

[44] Pinto, G., Canino, A., Castor, F., Xu, G. H., and Liu, Y. D. 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* (2017), pp. 765–775.

[45] Pinto, G., Liu, K., Castor, F., and Liu, Y. D. A comprehensive study on the energy efficiency of java thread-safe collections. In *International Conference on Software Maintenance and Evolution (ICSME 2016)* (2016).

[46] Sahin, C., Pollock, L., and Clause, J. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (New York, NY, USA, 2014), ESEM '14, Association for Computing Machinery.

[47] Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI'11*.

[48] Sinha, A., and Chandrakasan, A. P. Jouletrack-a web based tool for software energy profiling. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)* (2001), pp. 220–225.

[49] Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., and Berger, E. D. Eon: a language and runtime system for perpetual systems. In *SenSys '07*, pp. 161–174.

[50] Tiwari, V., Malik, S., Wolfe, A., and Lee, M. T. . Instruction level power analysis and optimization of software. In *Proceedings of 9th International Conference on VLSI Design* (1996), pp. 326–328.

[51] Whaley, J. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande* (New York, NY, USA, 2000), JAVA '00, ACM, pp. 78–87.

[52] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. Currentcy: A unifying abstraction for expressing energy management policies. In *In Proceedings of the*

USENIX Annual Technical Conference (2003), pp. 43–56.

[53] Zhu, H. S., Lin, C., and Liu, Y. D. A programming model for sustainable software. In *ICSE'15* (2015), pp. 767–777.

[54] Zhuang, X., Serrano, M. J., Cain, H. W., and Choi, J.-D. Accurate, efficient, and adaptive calling context profiling. *SIGPLAN Not. 41*, 6 (June 2006), 263–271.