Eflect: Porting Energy-Aware Applications to Shared Environments

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

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

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

ABSTRACT

Developing energy-aware applications is a well known approach to software-based energy optimization. This promising approach is however faced with a significant hurdle when deployed to the environments shared among multiple applications, where the energy consumption effected by one application may erroneously be observed by another application. We introduce Eflect, a novel software framework for disentangling the energy consumption of co-running applications. Our key idea, called *energy virtualization*, enables each energy-aware application to be only aware of the energy consumption effected by *its* execution. Eflect is unique in its lightweight design: it is a purely *application-level* solution that requires no modification to the underlying hardware or system software. Experiments show Eflect incurs low overhead with high precision. Furthermore, it can seamlessly *port* existing application-level energy frameworks — one for energy-adaptive approximation and the other for energy profiling — to shared environments while retaining their intended effectiveness.

CCS CONCEPTS

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

KEYWORDS

Energy Accounting, Energy Profiling, Power Disturbance, Concurrency

ACM Reference Format:
Timur Babakol, Anthony Canino, and Yu David Liu. 2022. Eflect: Porting Energy-Aware Applications to Shared Environments. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510145

1 INTRODUCTION

Energy efficiency in server-class computation platforms has significant impact on environmental sustainability as well as operational cost. *Energy-aware applications* [14, 35] emerge as a critical dimension of energy-efficient computing. An energy-aware application is *aware* of its own energy consumption during the execution, and the observation can in turn guide application-specific energy optimization. In software engineering, two prominent use scenarios of energy-aware applications are *energy-adaptive approximation* [3, 7, 9–11, 18, 19, 23, 29, 30, 33, 41] and *energy profiling* [2, 4, 5, 8, 15, 17, 22, 25, 27]. In Fig. 1 (a), application `A` may adpatively adjust some application-specific accuracy parameter, `acc` in the example, based on prior energy consumption of some work unit, `w`. In Fig. 1 (b), an energy profiler `C`, whose runtime is *de facto* an energy-aware application, may wish to determine the energy consumption of different logical units within an application, such as which method consumes more energy, `m1` or `m2`.

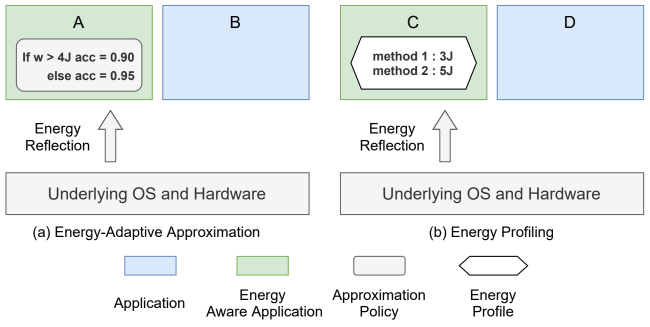


Figure 1: Energy Awareness in Shared Environments

1.1 The Challenge: Shared Environments

Energy-aware applications are supported through *energy reflection*, i.e., the ability of an application to observe its own energy consumption. Naively exposing hardware energy data [12] to the application however is fundamentally unsound in *shared environments* where multiple applications co-exist. To gain intuition on why well-designed energy-aware applications may ill-behave in shared environments, let us revisit Fig. 1. In Fig. 1 (a), a co-running application `B` — energy-aware or not — may incur energy consumption. The raw energy consumption `w` during the execution of the work unit may include that incurred by `B`. As a result, application `A` may degrade its accuracy to 0.9 even though the energy consumption *due to its execution* remains lower than 4J. Similarly, the hardware energy consumption during the `m1` (or `m2`) execution in `C` may include that incurred by `D`: the quality of profiling results is severely discounted due to the noise introduced by `D`.

The unfortunate phenomenon here is a fundamental hurdle against the *portability* of energy-aware applications: a well-behaved software system at the idealized exclusive *development* site ill-behaves at the realistic shared *deployment* site. The lack of portability directly challenges the *correctness* of energy-adaptive approximation, and the *usability* of in-situ energy profilers. From time-shared OS to cloud computing platforms, running applications in a shared environment is a basic need in modern computing. In that sense, the portability of energy-aware applications in shared environments is critical for the future adoption of this promising technique.

This overlooked problem is hitherto only addressed by a small number of OS-centric solutions [16, 32, 34, 38–40]. In other words, to support energy-aware *applications*, service providers must modify existing *OS infrastructure* that impacts all applications and all users on the server, a challenging case for adoption in practice. This is particularly bad news to *software developers*: the feasibility of their solution (energy-aware applications) becomes dependent of whether the OS in the deployment environment has adopted the OS features that commodity OS do not support, severely limiting the benefit of energy-aware applications.

1.2 Our Solution: Eflect

We introduce Eflect, a novel and lightweight software framework that allows energy-aware applications to gracefully port to a shared environment *without any modification to the OS*. The key insight of Eflect is to introduce an intermediate layer of *energy virtualization* during energy reflection, so that the application in a shared environment can only observe *its* share of energy consumption, with the illusion that it is exclusively responsible for the energy consumption when it performs energy reflection. Eflect virtualization significantly simplifies the *interface* between the application and the underlying system, and prevents the otherwise impedance mismatch between the development site and the deployment site. The portability of energy-aware application is achieved in that an energy-aware application can make the strong assumption that all its energy consumption observation indeed results from its own execution, regardless whether the deployment environment is shared.

The central problem Eflect addresses is to compute an application's share of energy consumption in a shared environment. It continuously monitors both the application behavior and the underlying system behavior, with several unique features:

Whole-Stack Monitoring with Minimal Interface. Determining the application share of energy consumption in a shared environment requires a clear accounting of the complex behavior of the systems stack. Eflect is a sophisticated framework that internally monitors a rich number of states: the activities of individual threads within an application, the activities of individual applications in a shared environment, the mapping between the application threads and the OS threads, the affinity of the threads and their migration, and the raw energy consumption by the CPUs across multiple energy domains. The most important observation however is that Eflect is capable of hiding these details within itself, and *leaving a minimal interface to the energy-aware application.* The design philosophy can be summarized as, "let us worry about these details, so an energy-aware application developer does not have to."

An Application-Level Solution with Decentralized Monitoring. Unlike existing OS-centric solutions, Eflect is built at the level of the *application*, a significantly more lightweight approach without the need of any VM/OS/hardware modification. Each energy-aware application in a shared environment is monitored by an individual instance of Eflect, and there is no coordination across different Eflect instances. This fundamentally decentralized design principle is a win for incremental deployment: an Eflect-monitored application can not only work with existing OS and hardware, but also can it co-run with arbitrary applications: the co-running applications may or may not be monitored by Eflect, and may not even be the same language runtime.

Fine Granularity. Eflect is a *fine-grained* and *online* solution that informs the share of energy consumption for *every time interval* (in millisecond scale) of the application execution of *every thread* in the application while the application is running in a shared environment. Under the hood of Eflect, the activities of the application as well as the activities of the entire underlying system are also monitored at this fine granularity, and the relative ratio between the two is used to determine the energy reflection for each time interval and each thread in each application.

1.3 Results

We evaluate Eflect through co-running state-of-the-art multithreaded Java benchmarks with representative server workloads from the Dacapo benchmark suite [6], together with Tensorflow [1]. Our results show Eflect is capable of achieving energy virtualization with high precision and low overhead.

As end-to-end case studies, we further investigate how to integrate Eflect with two application-level energy management frameworks — Aeneas [10] for energy-adaptive approximation, and Chappie [2] for energy profiling — and seamlessly transition them to a shared environment.

To the best of our knowledge, Eflect is the first application-level solution for porting energy-aware applications to a shared environment. Overall, this paper makes the following contributions:

- a novel application-level software framework for energy reflection in shared environments
- a systematic evaluation on the precision, overhead, and scalability of the framework
- a case study on how our framework may port existing energy-adaptive approximation solutions to shared environments
- a case study on how our framework may port existing energy profiling solutions to shared environments

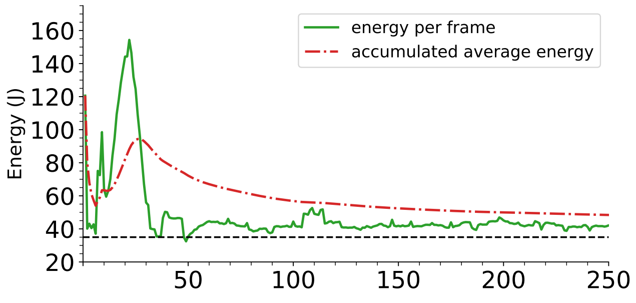
Eflect is an open-source project, and it is endowed with an API that allows the framework to be programmed for different porting needs. The project is hosted on an anonymous GitHub site `https://github.com/pl-eflect`.

2 MOTIVATING SCENARIOS

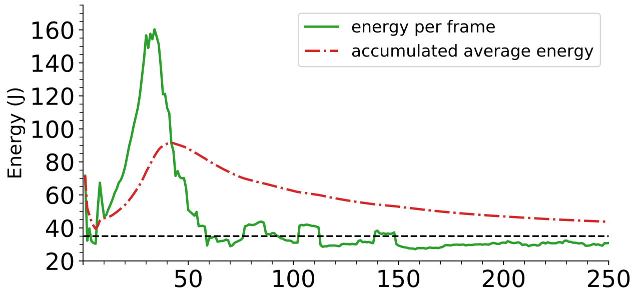
In this section, we motivate the need for Eflect with concrete examples of challenges when deploying energy-aware applications in shared environments.

2.1 Porting Energy-Adaptive Approximation

One example of an energy-adaptive approximation framework is Aeneas [10]. In Aeneas, alternative algorithms and application parameters can be programmed as *knobs*, and a goal for optimization can be defined by the programmer as a *reward*, e.g., the joules consumed for some unit of work in the application. At runtime, Aeneas dynamically adjusts program knobs to find a reward-optimal

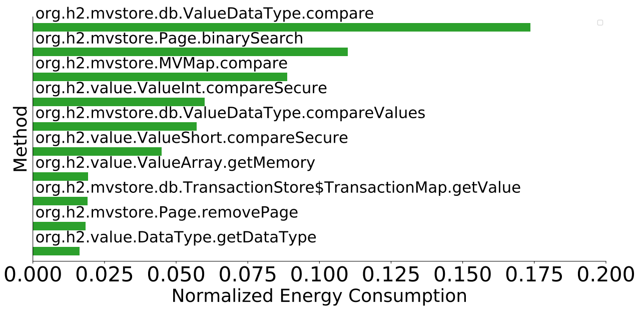


(a) an AENEAS run of `sunflow` when co-run with another `sunflow`

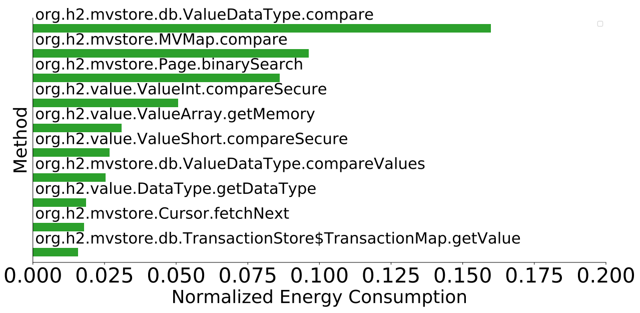


(b) an AENEAS run of `sunflow` when co-run with `eclipse`

Figure 2: AENEAS-Monitored Co-Runs of `sunflow` with Another Application (Each graph is a representative trace of a run. The X-Axis represents the units of work (frames) in elapsed time. The dotted horizontal line represents the SLA.)



(a) h2 profile in a single-application environment



(b) h2 profile when co-run with `sunflow`

Figure 3: CHAPPIE 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.)

combination of knobs that meets the reward target, also known as Service Level Agreement (SLA). For example, AENEAS may monitor `sunflow` [6], a popular ray tracing benchmark. A knob can be set for managing the level of image post-processing quality: less post processing produces an approximated image with less energy consumption. Processing each frame of `sunflow` can be defined as a unit of work, and 35J can be set as the SLA for processing a frame. In an environment where only one AENEAS runtime is in place, experiments show that AENEAS is effective in converging on the knobs that meet the SLA [10]. In other words, the energy consumption per frame will approach 35J "as time goes on."

Porting AENEAS to a shared environment however is non-trivial. Figure 2 shows two runs of an AENEAS-monitored `sunflow`, each in a shared environment co-running with another application. First, in both subfigures, AENEAS appears to have a difficult time to converge at the target SLA of 35J quickly. Indeed, in Fig. 2a, as the 250th frame is still rendered with more than 35J, it is unclear if the SLA will ever be reached. Second, the convergence behavior of `sunflow` of Fig. 2a and that of Fig. 2b are different. In other words, the same energy-aware application with the same setting may end up behaving differently depending on *what other application* is co-running.

2.2 Porting In-Situ Energy Profilers

An energy profiler example is CHAPPIE [2]. Given an application, CHAPPIE is able to produce an *energy footprint*, i.e., the relative energy consumption of program methods. Figure 3a shows a footprint produced by CHAPPIE—the top-10 energy consuming methods are shown for brevity—for h2 [6], a transactional database. Here it shows the top energy-consuming method of the application—`ValueDataType.compare`—and the relative difference in energy consumption between any pair of methods.

Porting CHAPPIE for deployment-site in-situ energy profiling in a shared environment, however, is a challenge. Figure 3b shows energy footprint produced by CHAPPIE for h2 in the presence of a co-running instance of `sunflow`. While the top consuming method `ValueDataType.compare` remains stable, the ranking of the remaining methods is different from the result in a single-application environment. For example, `ValueDataType.compareValues` and `ValueArray.getMemory` swap their positions in the ranking. The root problem is that when the energy consumption of a method, say `ValueDataType.compareValues`, is profiled by CHAPPIE, the co-running `sunflow` also incurs energy consumption, which may be incorrectly attributed to the method.

3 EFLECT DESIGN

Overall, EFLECT performs whole-stack monitoring to determine the "share" of energy consumption an application incurs in a per-thread and per-time-interval manner. An algorithm specification is

Algorithm 1 Eflect Algorithm

```
1  typedef Pid Int // process ID
2  typedef Tid Int // thread ID
3  typedef Eid enum {$E_1, E_2, \ldots E_n$} // energy domain ID
4  typedef Energy Float // energy in joules
5  typedef Activity Float // activity in jiffies
6  typedef Sample struct {
7    $\epsilon$ : Map⟨Eid, Energy⟩ // per domain energy
8    $\eta$ : Map⟨Eid, Activity⟩ // per domain activeness
9    $\delta$ : Map⟨Tid, Eid⟩ // thread affinity
10   $\gamma$ : Map⟨Tid, Activity⟩ // per thread activeness
11 }
12 typedef VEnergy Map⟨Tid, Energy⟩ // virtual energy
13 $\mathcal{V}$ : List⟨VEnergy⟩ // virtual energy store
14 INTERVAL : Int // sampling interval

15 function Main
16   p ← GetCurretProcess()
17   loop at rate (INTERVAL)
18     s ← DoSample(p)
19     $\mathcal{V} \overset{+}{\leftarrow}$ DoVirtualize(s)

20 function DoSample(p : Pid): Sample
21   s ← new Sample
22   for e in {$E_1, E_2, \ldots E_n$} do
23     s.$\epsilon$[e] ← EflectEnergy(e)
24     s.$\eta$[e] ← EflectActivity(e)
25     for t in GetDomainThreads(e) do
26       if p = GetPid(t) then
27         s.$\delta$[t] ← e
28         s.$\gamma$[t] ← EflectActivity(t)
29   return s

30 function DoVirtualize(s : Sample ): VEnergy
31   v ← new VEnergy
32   for each (t, e) in s.$\delta$ do
33     v[t] = $\frac{s.\gamma[t]}{s.\eta[e]} * s.\epsilon[e]$
34   return v

35 function ReadEnergy(p : Pid ): VEnergy
36   V ← new VEnergy
37   for each v in $\mathcal{V}$ do
38     V $\overset{+}{\leftarrow}$ v
39   return V

40 function EflectActivity(Tid) : Activity // obtain thread activity
41 function EflectActivity(Eid) : Activity // obtain domain activity
42 function EflectEnergy(Eid) : Energy // obtain domain energy reading

43 function GetCurrentProcess() : Pid // obtain current running process
44 function GetDomainThreads(Eid) : Tid[] // obtain processes in an energy
   domain
45 function GetPid(Tid) : Pid // get thread's process id
```

shown in Algorithm 1. For readers unfamiliar with the application-system interaction w.r.t. energy consumption, we first provide a background.

A Primer on the System Stack. Server-class computers generally consist of multiple sockets. In mainstream multi-core CPUs, a socket coincides with the conceptual notion of the *energy domain*, the hardware unit where energy consumption is reported, and where the voltage and frequency can be independently adjusted through Dynamic Voltage and Frequency Scaling (DVFS) [28]. In other words, when one core in an energy domain is adjusted to a particular frequency, all cores in the same energy domain are adjusted to the same frequency. In our algorithm, we represent these energy domains by $\{E_1, E_2, \ldots E_n\}$ (Line 3). The per-domain energy consumption is represented as a mapping ϵ in the Sample structure collected at each time interval.

During the lifetime of an application, a thread may migrate from one CPU core to another. The CPU core occupied by a thread is known as *affinity*. It is represented as a mapping δ in the Sample structure. As the energy domain is the granularity for which we can obtain raw energy readings, it also serves as the unit for thread affinity: a thread may migrate between CPU cores within an energy domain but will not produce an observable difference in its contribution to the underlying energy readings.

The activities of individual applications in the OS are represented through some abstract notion of time used by a process. Time keeping is a basic service universally supported in practical OS. In Linux-compatible systems, the most widely known representation of time that indicate process/thread activities is the *jiffy*. For example, Linux periodically updates the number of jiffies the threads of a process consumes in the user mode and in the kernel mode. In addition, the same form of time keeping is also supported for hardware, i.e., how much time the CPU is actively running *any* process. In our algorithm, the thread activity information is represented as a mapping γ, and the hardware activity information is represented as a mapping η, both in the Sample structure.

Algorithm Overview. Eflect first obtains the process ID of the monitored application through GetCurrentProcess (Line 16). As shown by the top-level loop at Line 17, Eflect continuously monitors the behavior of an application, and at each INTERVAL, it samples the underlying hardware and operating system resources (Line 18), and performs virtualization (Line 19), i.e., determining the share of energy consumption attributable to the application thread of interest. Each step of virtualization produces a piece of *virtual energy* data, i.e., the energy consumption due to the application, represented as VEnergy. Structurally, each piece of virtual energy data is a mapping from each thread in the application (Tid) to its respective energy consumption (Energy). As Eflect proceeds, it accumulates VEnergy data in structure $\mathcal{V}$, which Eflect exposes to the monitored application—we will discuss this interface at the end of this section. Notation $\leftarrow$ is standard assignment, and $\overset{+}{\leftarrow}$ denotes list addition. Let us now focus on the two core steps in the loop, sampling and virtualization.

Sampling. Function DoSample (Lines 20 - 29) represents a single sampling of underlying hardware and operating system resources. Each step of sampling populates a Sample (Lines 6-11). Eflect gathers the energy (EflectEnergy) of each energy domain and records this information in map ϵ at Line 23. Hardware activity, in the form of jiffies per energy domain, is also sampled, stored in map η at Line 24. For each thread that belongs to the application we monitor (Line 26), we record its affinity in map δ and its activity in map γ, at Line 27 and Line 28. Function EflectActivity is overloaded to inspect OS activities either by thread, or by hardware.

Virtualization. We compute the "share" of energy consumption due to the application through function DoVirtualize (Line 19). It returns a value of VEnergy, i.e., a mapping from the application's threads to their computed share of energy consumption, from the raw Sample collected from the underlying system. Logically, this means we assign a thread a share of energy consumption based upon its "slice" of activity on an energy domain.

Recall that we use OS time (jiffies) to represent the level of activities of a thread. The curious question is why *time* serves as a good attributing factor for *energy*. In physics, energy is computed as the multiplication of *power* and time. An important insight here is that within an energy domain, all cores share the same voltage and frequency, and as a result, the *same power* according to DVFS design. More concretely, given a Sample `s` with thread affinity mapping $\mathtt{s}.\delta$, for each thread `t` and its associated domain `e` (Line 32), we compute $\frac{\mathtt{s}.\gamma[\mathtt{t}]}{\mathtt{s}.\eta[\mathtt{e}]}$, i.e., the proportion of `e`'s activity that came from `t`. Because we have energy measurements associated with energy domains, we attribute a portion of the domain energy consumption ($\mathtt{s}.\eta[\mathtt{e}]$) to thread `t` based upon its proportional activity on that domain on Line 33.

Virtualization hides the details of *where* a thread performed its work on the system. If a thread `t` migrates from energy domain $\mathtt{E}_1$ to $\mathtt{E}_2$, DoSample will capture `t` latest location upon a subsequent sampling; the associated round of DoVirtualize will take the threads latest energy domain into account. From the application's viewpoint, it sees that its threads consume energy, but does not need concern itself with any of the details.

Overall, DoVirtualize (and its helper DoSample) must be aware of underlying system details — OS jiffies, OS affinity, CPU energy domains —- but all such details are hidden from the client application: the latter only needs to be aware what share of energy consumption it is attributed for. Recall the key design philosophy of our framework (§ 1): let Eflect worry about the details so that an application developer does *not* need to.

Abstraction and Customization. Eflect is an open programmable framework that supports flexible customization. In our algorithm specification, it is important to observe that our discussion on energy domains, thread affinity, and thread activity are all *abstract*, independent of specific hardware topology, OS file system structures, or platform-specific representations. While these latter details may be important to the implementation, they are agnostic to the overall procedures of DoSample and in particular DoVirtualize. In other words, our virtualization algorithm itself is platform-independent.

Eflect exposes an API for developers to specialize Eflect to their own systems. These functions are listed at Lines 40- 45. We also provide a simple programming interface for Eflect to interact with the application it monitors. The primary function is ReadEnergy, which returns the accumulated energy attributed to the monitored application, defined as the sum of entries in $\mathcal{V}$. Several API variants are also supported, such as timer-based read. The simple design here is aligned with the design goal of Eflect, exposing a minimal interface to the application. In addition, it decouples the application from the virtualization process: the application can query *its* energy consumption at a pace it wishes, which may differ from `INTERVAL`.

For example, in our current implementation (see § 4), raw energy consumption values are read through querying a hardware register, a common feature in Intel architectures. Alternatively, users working with alternative CPU architectures may need to re-implement the ReadEnergy function to address their specific architectures, or query the A/D converter values of the external energy meter if energy values are obtained through a meter. The remaining logic of Eflect does not need to be changed. As a second example, our current implementation focuses on the most dynamic portion of energy consumption in server-class platforms — the CPU and memory sub-systems — but a user can also write her own ReadEnergy function to further include energy consumption values of network adaptors, storage systems, *etc*, depending on need.

4 EFLECT IMPLEMENTATION

Eflect is implemented in Java, and its runtime shares the JVM of the monitored application. We now describe our default implementation of our API on a multi-socket Intel platform with Linux (details in § 5.1).

Our implementation of the EflectEnergy API function resorts to the Intel RAPL [12] interface to obtain energy samples. These samples are collected using jRAPL [21], a Java library for interfacing with RAPL. As Intel RAPL reports energy samples per CPU socket, each CPU socket corresponds to an energy domain in our algorithm specification, i.e., $\mathtt{E}_1$ and $\mathtt{E}_2$. Each energy sample is a combination of socket energy consumption from CPU cores, CPU uncore (caches, TLB, etc), and memory.

Our implementation of the EflectActivity API function queries the Linux jiffy information through the `proc` pseudo-file system, so that each sample collection is effectively a file system read. Recall that EflectActivity is an overloaded function that may either collect hardware jiffies or thread jiffies. We gather the former through `proc`'s subdirectory `stat`, and the latter through `proc`'s subdirectory named by each PID. In each subdirectory, the jiffies are separated by user time, kernel time, interrupt time, and "nice" time (for invoking utilities or shell scripts with a particular priority). Our Activity data is the sum of them. In the case of hardware jiffies, the per-core data are also aggregated per energy domain.

The GetCurrentProcess, GetDomainThreads, and GetPID API functions are implemented with simple Linux features. The first function is implemented by the `GLIBC` syscall `getpid`. The second function is implemented by searching the `/proc/[Pid]/task` directory. The third function is implemented by parsing the data in `/proc/[Pid]/task/[Tid]/stat`.

The `INTERVAL` is set at 40ms. The considerations of this setting are three folds. First, jiffies in Linux are updated at 4ms by default, so any rate higher than 4ms/sample is unlikely to observe jiffy changes. Second, energy sampling, especially at a high rate, may disturb the energy behavior of the original application [2]. We choose to set the rate at one magnitude lower than 4ms, so that no benchmark we experimented is disturbed. Third, the 40ms appears to incur low overhead with high precision (detail in § 5.5).

One implementation consideration is that querying RAPL and the OS file system synchronously may serialize both. Worse, if one returns with a delay due to unexpected system events, both may be delayed. In our implementation, both are sampled *asynchronously*, each with a thread, producing information to a buffer. The threads are managed by a `ThreadPoolExecutor`. With this design, each iteration of DoSample only involve efficient buffer lookups.

5 EXPERIMENTAL VALIDATION

One experimental evaluation aims to answer the following questions. **RQ1**: can Eflect virtualize energy consumption of applications in a shared environment, with decentralized and fine-grained monitoring? **RQ2**: what is the precision, runtime overhead, and scalability characteristics of Eflect? **RQ3**: can Eflect help developers port their energy-aware applications to a shared environment, while retaining their effectiveness in energy management? We answer **RQ1** and **RQ2** in this section, and defer **RQ3** to § 6.

5.1 Experimental Setup

We evaluate Eflect on a dual socket Intel E5-2630 v4 2.20 GHz CPU server, with 64GB DDR4 of RAM. Each socket CPU has 10 cores, 20 "virtual" cores with hyper threading enabled. The machine runs Debian 4.9 OS, Linux kernel 4.9, with the default Debian power governor. All experiments were run with Java 11 on top of Hotspot VM build 11.0.2+9-LTS.

We evaluate Eflect through co-running state-of-the-art multi-threaded Java benchmarks with representative server workloads spanning scientific computing, databases, graphics, graph processing, and machine learning. Most of our benchmarks come from the latest release of the DaCapo benchmark suite [6] [1], which we select for two reasons. First, Dacapo is arguably the most widely used benchmark suite for evaluating the performance of (non-mobile) multi-threaded Java applications. Second, the newest release has significantly expanded the data size of several benchmarks (`sunflow` and `h2`), and more importantly, it includes some new benchmarks (such as `graphchi` for graph processing and `biojava` for bioinformatics) representative of the recent workloads. To further evaluate Eflect against machine learning workloads, we also managed to have Eflect monitor Tensorflow [1] in a shared environment. Each instance of Tensorflow classifies images using a DNN, Inception v3 [36].

For each experiment, we exclude harness setup time, and run a benchmark for 10 iterations with the first 5 discarded as a warmup. Fig. 4 shows benchmark statistics.

5.2 Eflect Virtualization

To answer **RQ1**, we evaluate Eflect's virtualization in a shared environment where each application is an instance of Eflect. As a result, each application produces a virtualized energy trace of its runtime. To visualize, we present the *normalized energy share trace*, defined as a time series where each data point is the fraction of energy attributed to the Eflect-monitored application over the entire hardware energy consumption of the same time interval.

Examples of homogeneous and heterogeneous workloads are presented in Figs. 5 and 6. In each figure, the time stamps start when all co-running benchmarks have started its 6th iteration, and end immediately before any of the co-running benchmarks completes its 10th iteration (recall that we discard the results of the first 5 iterations). We align the traces from each application based on timestamps, and "stack" them on top of each other. Since the

benchmark	workload	total threads	active threads	execution time (s)
avrora	large	71	69	175.98
batik	large	9	8	17.41
biojava	default	7	6	22.76
eclipse	large	706	18	59.14
graphchi	huge	53	50	246.78
h2	large	954	39	115.91
jython	default	7	6	10.63
pmd	large	8	6	54.65
sunflow	large	88	46	61.13
xalan	default	47	46	9.00
tensorflow	inception v3	45	21	23.39

Figure 4: Benchmark Statistics (Workload refers to the data size specified by DaCapo for each benchmark; their details can be found on Dacapo's website [2]. 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 interval. Execution time is averaged across runs when the application is not co-running with other applications. These statistics were produced with the experimental setup described in § 5.1).

decision-making of energy attribution is decentralized, the sum at each timestamp is not guaranteed to be 100%.

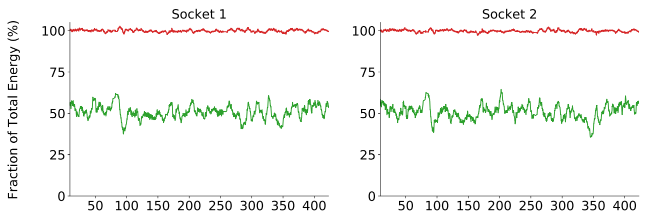
According to Eflect, both instances of `sunflow` in the homogeneous run observe approximately 50% of hardware energy over the course of the runs, aligned with our intuition of co-running two similar CPU-intensive applications. Note that Eflect does not result in a naive, even "split" of system energy: as one `sunflow` instance receives greater virtualized energy, the other receives less, as can be seen at timestamp 90.

Each of the two instances of `tensorflow` exhibits asymmetric workload distribution across two sockets. Interestingly, we found that when 2 instances of `tensorflow` co-run, each instance would choose one CPU socket to place the vast majority of its workload, and this results in asymmetric workload distribution. As shown in Fig. 5b, the green application instance dominates the use of socket 1, while the red application instance dominates socket 2. Despite the asymmetric behavior however, the key observation is that for both sockets, the "stacked" energy consumption approximates at 100%. This result is aligned with our intuition that Eflect divides up system energy across applications at a per time interval granularity, despite the decentralized instances of Eflect at work.

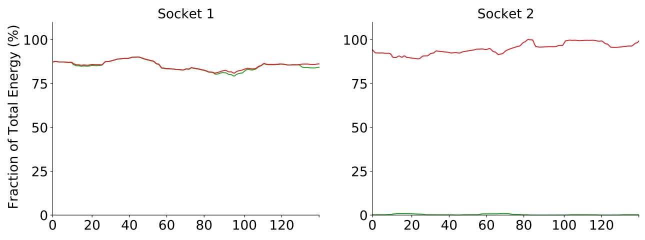
In the first heterogeneous case, shown in Fig. 6a, we examine `sunflow` and `h2`. The energy is not uniformly distributed, with nearly 95% being consumed by `sunflow`. This is not surprising as `sunflow` is CPU-bound, while `h2` is I/O-bound.

In the second heterogeneous case in Fig. 6b, we examine `h2` and `pmd`. Although Eflect reports different utilization of each socket for the applications, the total energy is more evenly distributed, with `h2` consuming 51.74% of the total energy, `pmd` consuming 39.89%, and the rest is attributed to the OS. As a source code analyzer, `pmd` is also I/O bound, so it is expected that the energy is more evenly shared across the two I/O bound benchmarks. First, observe that the benchmarks exhibit phased behaviors (on both sockets), where energy utilization goes through significant fluctuation in a near periodic fashion. Second, despite the fluctuations, the sum of the two benchmark's normalized energy use is relatively stable, indicating the vast majority of energy use is distributed to the two

[1] version evaluation-git+8b7a2dc, released in June 2019
[2] https://dacapo-bench.org

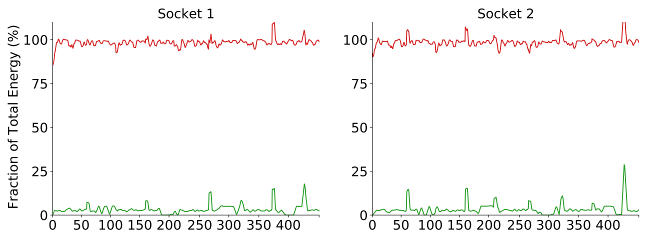


(a) Two co-runs of `sunflow`.

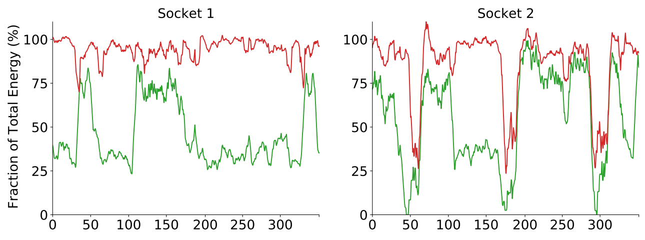


(b) Two co-runs of `tensorflow`.

Figure 5: Normalized Energy Share Traces of Homogeneous Co-Runs (The red line is vertically stacked on top of the green line, with the green line shows the normalized share of one application, and the red line shows the sum of that of both. The X-Axis represents the elapsed time.)



(a) Co-run of `sunflow` (green) and `h2` (red).



(b) Co-run of `h2` (green) and `pmd` (red).

Figure 6: Normalized Energy Share Traces of Heterogeneous Co-Runs (The red line is vertically stacked on top of the green line, with the green line shows the normalized share of one application, and the red line shows the sum of that of both. The X-Axis represents the elapsed time.)

applications. Some time stamps in socket 2 are the exception to this trend. This is possible when both benchmarks are waiting for I/Os.

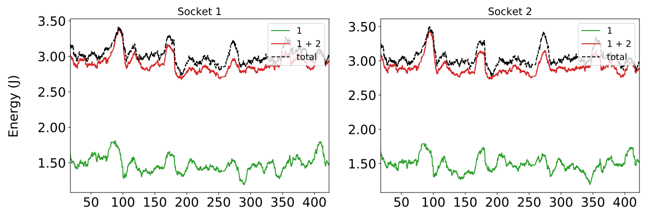


Figure 7: Raw Energy Traces for Co-Runs. (The red line is vertically stacked on top of the green line, with the green line shows the energy consumption (in joule) of one application during an interval, and the red line shows the sum of that of both. The black line shows total energy consumed by the hardware (socket) during the interval. The X-Axis represents the elapsed time intervals.)

Before we proceed, we would like to distinguish between the normalized energy share trace we introduced earlier and the *raw energy trace*, a time series of raw energy consumption. An example is shown in Fig. 7. As this figure shows, the hardware energy consumption does fluctuate as time elapses. The top flat-line in Fig. 5 and Fig. 6 instead reveals that the overall *share* (in percentage) of energy consumption attributed to the applications is close to all consumed by the hardware.

In the rest of the section, we address **RQ2**.

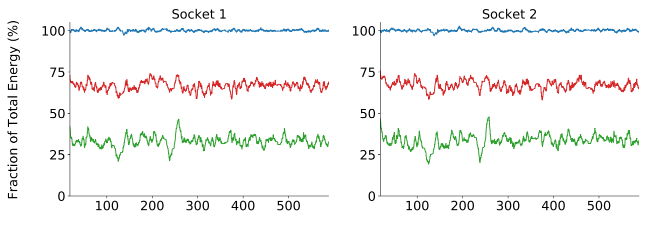
5.3 Scalability

We next evaluate how Eflect virtualization scales with an increasing number of applications. Scalability is important in applying Eflect in a production-like environment, where the server may consolidate applications.

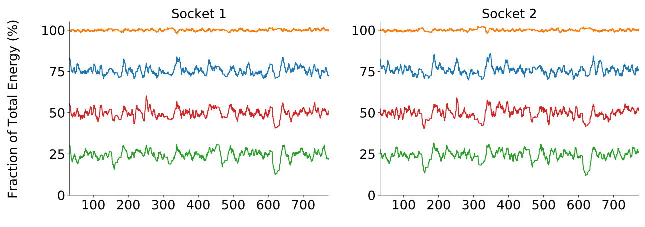
Results with additional instances are presented in Fig. 8. Overall, the data is consistent with what was shown in Fig. 5: as the number of applications increases, similar energy shares are intuitively divided among the co-running applications. The consistency here is remarkable in the presence of *decentralization*: each Eflect runtime makes independent decisions on how energy share is computed, and they *happen to* match up well without any coordination.

One interesting observation is that the variations in energy behavior become more pronounced as the application count increases: in the 5-co-run figure, the "up" and "down" of individual lines (except the top line) are more noticeable than those of a 3 co-run figure. The resource competition between applications is exposed, which is expected from CPU-intense workload. Despite the fluctuations of individual applications, Eflect is capable of achieving a stable sum: the top line — the sum of all individual energy shares — is around 100% and remains flat throughout the runtime.

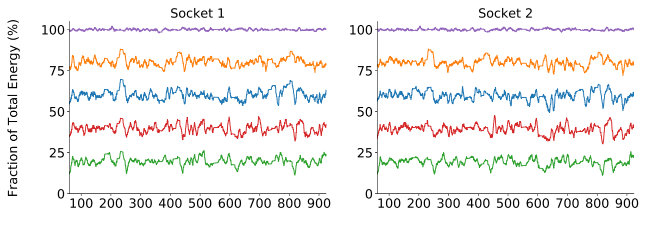
In Fig. 9, we further show a heterogeneous co-run of 4 Eflect-monitored applications. Again, `sunflow` dominates the energy share (around 75%), followed by `xalan` (around 18-20%), then by `graphchi` (around 4-5%), and finally by `h2`. This is aligned with our general knowledge on the CPU-intensiveness of these benchmarks. Furthermore, observe that the top line closely matches 100% of hardware energy consumption.



(a) 3 co-runs



(b) 4 co-runs



(c) 5 co-runs

Figure 8: Normalized Energy Share Traces of Co-Runs With 3-5 `sunflow` Co-Runs. (The red line is vertically stacked on top of the green line, with the green line shows the normalized share of one application, and the red line shows the sum of that of both. The X-Axis represents the elapsed time.)

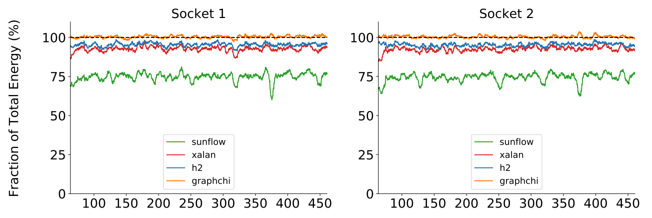


Figure 9: Normalized Energy Share Traces of Heterogeneous Co-Runs of `sunflow`, `xalan`, `h2`, and `graphchi`. (The red line is vertically stacked on top of the green line, with the green line shows the normalized share of one application, and the red line shows the sum of that of both. The X-Axis represents the elapsed time.)

5.4 Precision

Precision evaluation for energy accounting in the shared environment is a challenging task. In particular, note that it would be *unsound* to consider the energy trace of an application while it runs in isolation as a baseline and compare it against its energy trace while co-running with others. Indeed, co-scheduling (e.g., [13, 24]) is a classic technique for improving energy efficiency, and this optimization is rooted on the fact that an application while co-running may exhibit a *different* energy behavior (i.e., less energy) than when it runs in isolation.

The decentralized nature of EFLECT provides an interesting alternative for precision evaluation. A key insight from our discussion of Fig. 5, Fig. 6, and Fig. 8 is that the "top flat line" is aligned with our intuition of precision: despite the decentralized decision-making by individual EFLECT runtimes, the sum of their individual energy consumption should add up close to the hardware energy consumption. A significant misalignment between the two would be an indication that EFLECT mis-attributes energy.

We concretize our intuition with two metrics — *gap* and *correlation* — to study the precision of EFLECT. Gap is defined as the average normalized distance between the sum of raw energy consumption of individual applications and the raw energy consumption of the hardware per time interval.

Correlation is defined as the Pearson's Correlation Coefficient (PCC) between the stacked raw energy trace of the applications and the raw energy trace of the hardware. As a metric, gap indicates the overall mis-alignment between what EFLECT attributes and what the machine consumes. Correlation in addition offers further evidence on alignment by taking the temporal dimension into consideration.

The gap and correlation results are produced using the data collected in the virtualization experiments and are presented in Fig. 10a and Fig. 10b respectively. Observe that across all benchmarks with 2 co-runs, the average gap is low (averaged at 3.15%), indicating the misalignment is small. The correlation is high, with an average of 0.90, where 0.70 is the golden standard for strongly correlation. Overall, the combination result of low gap and high correlation provides evidence on the precision of EFLECT.

More encouragingly, our precision results scales with the number of applications. When compared with their counterparts of co-running 2 instances, co-running more than 2 instances does not appear to impact precision significantly, both in gap and correlation.

It is worth pointing out that EFLECT is capable of attributing the energy consumption due to *system calls* to the calling application, because the jiffies we use for attribution consist of both kernel mode jiffies and user mode jiffies for each application; see § 4.

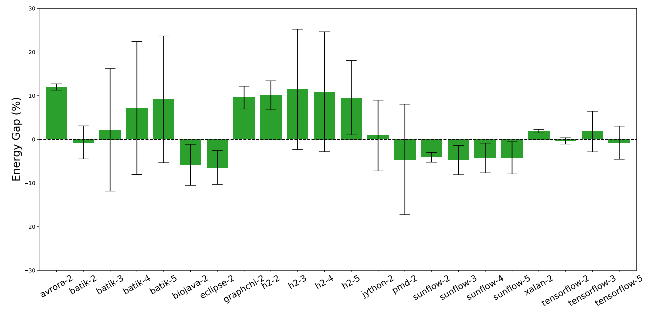
5.5 Overhead

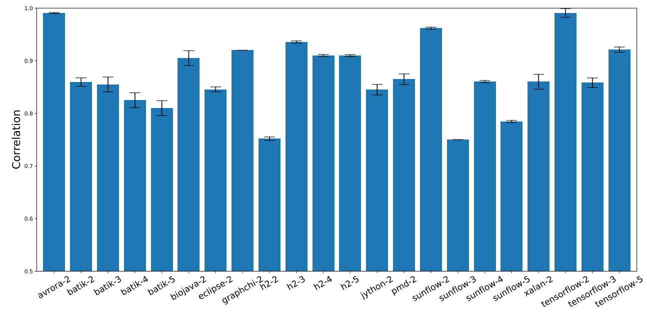
We now discuss the impact of EFLECT on execution time. We compute the percentage difference of the execution time between an original benchmark run and an EFLECT-monitored run. Over all benchmarks, EFLECT incurs a 1.66% overhead for co-running 2 benchmark instances, as shown in Fig. 10c.

The same figure shows overhead appears to scale reasonably well. As seen here, the overhead does not appear to follow a trend of increase when the number of applications increases. For example, as the number of runtime instances increases, neither `batik` nor h2 shows a pattern of increased overhead.

5.6 Raw/Additional Data

Our project website contains the full raw data. It also includes co-run scenarios that we leave out due to space, such as heterogeneous


(a) Gap


(b) Correlation

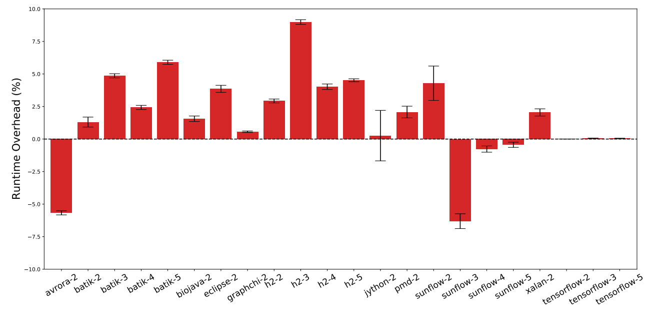
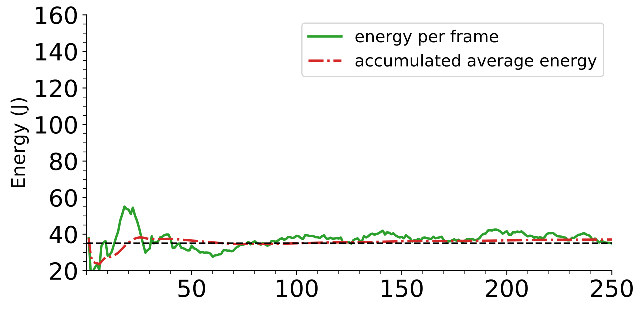
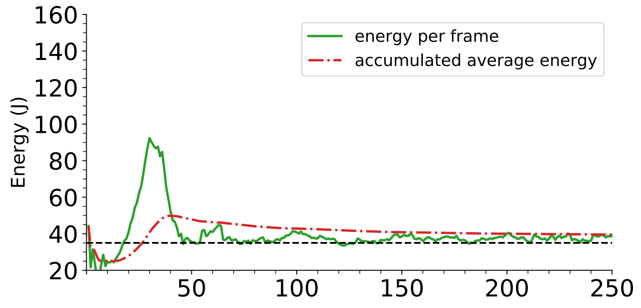

(c) Overhead

Figure 10: Precision and Overhead. The number after the benchmark name indicates the number of co-runs.



(a) co-run with another `sunflow`



(b) co-run with `eclipse`

Figure 11: EFLECT-Monitored AENEAS-Enhanced Co-Runs of `sunflow` with Another Application (The X-Axis represents the units of work (frames) in elapsed time. The dotted horizontal line represents the SLA.)

combinations (Fig. 9) or the precision and overhead results for other benchmarks (Fig. 10) when co-running applications are 3 or 5 instances.

6 EFLECT CASE STUDIES

To answer **RQ3**, we next perform two case studies of porting energy-aware frameworks with EFLECT.

6.1 EFLECT for AENEAS

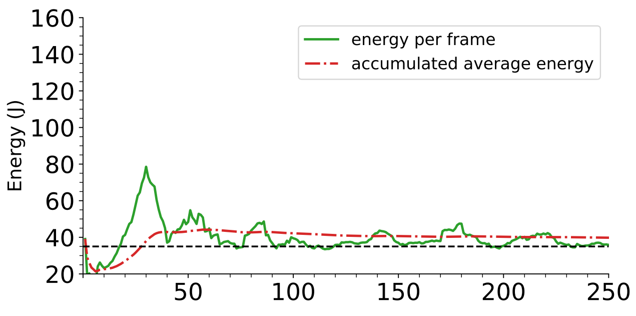
We first demonstrate the benefit of porting AENEAS to enable energy-adaptive approximation in a shared environment with EFLECT.

Fig. 11 shows two AENEAS runs of `sunflow` while the application is monitored by EFLECT, which corresponds to the same experiments we introduced in Fig. 2. As shown, EFLECT greatly enhances AENEAS optimization in a shared environment. In Fig. 11a, AENEAS quickly converges to the target 35J SLA after 50 frames, as opposed to its counterpart's sub-optimal convergence without EFLECT in Fig. 2. In addition, the `sunflow` co-run with `eclipse` in Fig. 11b also converges in a similar time frame.

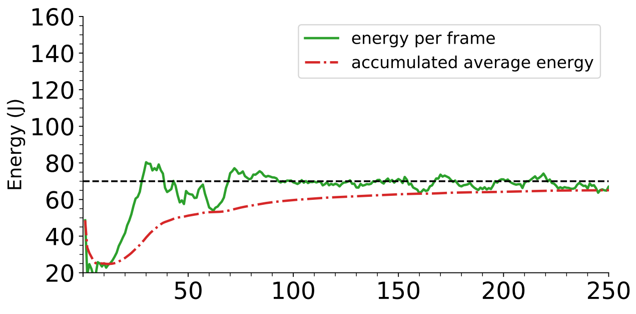
Fig. 12 shows a co-run of AENEAS-enhanced `sunflow` where both applications are AENEAS-enhanced, each targeting a *different* SLA. This presents a challenging environment for both EFLECT virtualization and the AENEAS optimizer: both instances of AENEAS-enhanced `sunflow`'s perform online adaptive approximation, with each instance potentially adjusting the system energy consumption while EFLECT tries to present a stable virtualization for the other. As it turns out, each instance of AENEAS-enhanced `sunflow` converges to the their respective target SLA.

6.2 EFLECT for CHAPPIE

For our second case study, we use EFLECT to port CHAPPIE energy profiling to a shared environment; in particular, we study the *stability* of CHAPPIE energy footprints of co-running applications. We first run CHAPPIE on a single instance of each application to produce a representative energy footprint, and then co-run two instances of each application with CHAPPIE using EFLECT as a virtualization layer to produce two energy footprints. We define the stability as



(a) sunflow instance with 35J SLA



(b) sunflow instance with 70J SLA

Figure 12: Eflect-Monitored Aeneas-Enhanced Co-Runs of 2 sunflow's with Different SLAs (The first application targets 35J SLA, and the second targets 70J. The X-Axis represents the units of work (frames) in elapsed time. The dotted horizontal line represents the SLA.)

benchmark	co-run 1	co-run 2
avrora	1.00 ± 0.01	1.00 ± 0.00
batik	0.98 ± 0.07	1.00 ± 0.03
biojava	1.00 ± 0.00	1.00 ± 0.01
eclipse	0.88 ± 0.10	0.87 ± 0.10
graphchi	0.89 ± 0.06	0.89 ± 0.07
h2	0.99 ± 0.04	0.99 ± 0.03
jython	0.89 ± 0.14	0.90 ± 0.14
pmd	0.99 ± 0.06	0.99 ± 0.07
sunflow	0.99 ± 0.04	0.99 ± 0.03
xalan	0.98 ± 0.03	0.98 ± 0.03
tensorflow	0.99 ± 0.00	0.99 ± 0.00

Figure 13: Energy Footprint Correlation between Single-Runs and Co-Runs with Eflect

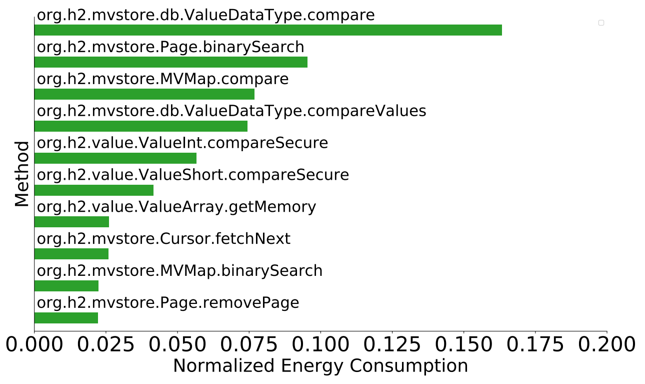


Figure 14: Energy Footprint for h2 in a Eflect Co-Run.

the PCC between each multi-application energy footprint and the single instance footprint, aligned through method names.

We detail these results in Fig. 13. The data shows that Chappie energy profiling results remain highly stable while being ported to the shared environment. The co-run energy footprints highly correlate with their single instance counterparts, which indicates that a programmer using Chappie+Eflect to profile an application will see consistent energy profiles even in a shared environment. In other words, despite co-running multiple applications may indeed change the energy trace of each individual application, the relative standing of *method* energy consumption within each application depends more on the logic within each application, and it remains stable.

Fig. 14 shows a computed footprint for h2. This footprint is consistent with the single-run footprint in Figure 3a, and an improvement over the shared environment case without Eflect in Figure 3b.

6.3 Development Effort

The customization effort for porting energy-aware frameworks with support for shared environments is minimal. In our integration of Eflect with Aeneas, we provide a re-implementation of Aeneas's Reward class whose valuate method now calls Eflect's ReadEnergy (§ 3). In the Chappie customization, we reimplemented their EnergySampler class to call ReadEnergy from the sample method. Their original source code tracks thread migration through sampling affinity. Since Eflect is already socket topology-aware, we simplified their code. In both cases, we were able to complete the customization within 1 hour.

7 THREATS TO VALIDITY

The primary target platforms that Eflect is designed for are server-class computing systems. Our experimental setup and benchmark selection reflect this assumption. We think Eflect may indeed be customized on other platforms where energy consumption matters, such as mobile/embedded systems. This can be achieved through reimplementing a small number of APIs defined in § 3, but the experimental effectiveness of Eflect on those platforms should not be taken for granted. The use scenarios of co-running multiple applications/apps in mobile/embedded systems do exist, but we think there is a more pressing need for support them in server-class systems where significant CPU/memory sharing across multiple applications is a basic requirement. In mobile systems for instance, the largest energy consumption components are the display [20], where sharing across different apps is limited.

8 RELATED WORK

The essential question of how to regulate the share of energy consumption to individual applications in a shared environment leads to "ground-up" redesigns of operating systems. One classic example is Currentcy [40] and its successor ECOSystem [39]. Just as an OS can manage how CPU/memory resources are allocated to processes, Currentcy manages how much energy is allocated or delivered to processes. Another example with a similar design goal is Cinder [32]. Strictly speaking, these solutions focus on energy allocation instead of energy attribution, so their goals are different from ours. That said, an energy-aware application in their frameworks is only aware of the share of energy/power delivered by the OS, so a parallel with ours can be set in terms of energy disentanglement. Unfortunately, extensive OS re-design is a hurdle to the wider deployment of these attractive solutions.

AppScope [37] is an energy metering framework for Android smartphones. It relies on an OS kernel modification to monitor kernel/app activities, and in turn provide a breakdown on energy consumption across hardware modules and systems/applications. Power sandboxes [16] is another OS principal for mobile devices, where an OS-level power sandbox can accurately attribute power with limited hardware sharing to consider. Both require OS changes. This requirement may be appropriate for their intended use context — mobile devices — but can be challenging for adoption in servers (§ 1). Further afield, power containers [34] is another OS solution that builds a linear power model over hardware performance counters on multi-core CPUs, such as cycles and last-level cache requests, which serves as the basis for attribution of global power to individual cores. this work differs from ours in that it addresses energy attribution for hardware components instead of applications. In Aequitus [31], multiple energy management applications that may adjust the frequency of the underlying CPUs (through DVFS) are given an equal slice of time in scheduling in a shared environment. The goal of that system is to ensure fairness in energy management.

An independent problem is energy profiling, i.e., how to attribute application energy consumption to individual software components and programming abstractions *within an application*. Eprof [26] and E-Android [15] are two examples that account for smartphone energy/power by attributing them to different software/hardware functionalities, such as phone, WiFi, or user-defined energy-consuming events. We reviewed Chappie in § 2.

Finally, popular tools such as the "screen time" feature on iPhones and "power usage" feature on many Android phones support basic functionalities of tracking the usage time of individual apps. Eflect is significantly finer-grained: its jiffy-based attribution is per time interval per energy domain. Our use of jiffies as an indicator for energy attribution is grounded on the fact that all cores in the same energy domain has the same power during the same time interval. The time-based energy estimation in the tools cited above however does not conform to this assumption.

9 CONCLUSION

Eflect is a novel application-level software framework aiming to port energy-aware applications to a shared environment. Eflect monitors the activities of the underlying system in a fine-grained manner, but exposes a minimal interface to the client application. Through energy virtualization, Eflect can guide energy approximation and energy profiling frameworks to achieve their intended goals while avoiding energy entanglement. Eflect is a lightweight approach that requires no modification to VM/OS/hardware, and can be adopted to commodity computer systems.

Reflections and Outlook. The portability challenge that Eflect addresses is based on our own experience. In prior efforts, we attempted to port existing energy-aware solutions — those thoroughly evaluated in non-shared environments — but they behaved poorly in shared environments. The Eflect design we eventually came up with is simple and easy to deploy, but the path leading to it was a learning experience. In particular, this is a design space where overhead, precision, and scalability all matter. For example, if we had ignored thread affinity or migration in our design, the precision would suffer. As another example, if we had not taken a sampling-based approach but instead relied on querying thread jiffies through instrumentation, the overhead would rise significantly.

In our opinion, the real excitement of Eflect is the *bridge* role it plays for future research: enables existing application-level energy-aware solutions to embrace platforms such as data centers and cloud servers with little additional effort. In essence, Eflect empowers developers with energy virtualization in a multi-application environment where energy-related developments would otherwise be convoluted due to energy entanglement. Our case studies in § 6 demonstrate two applications of Eflect in energy-related domains. Now that the bridge is built, new doors may be opened for exploring additional software engineering problems toward principled energy management in shared environments. For example, future research may systematically study the optimality and guarantees when multiple energy-aware applications are approximated at the same time in a data center setting. As another example, in-situ energy debugging tools may be designed for cloud servers, and new opportunities may exist for deployment-site energy bug fixing when coupled with state-of-the-art techniques of automated program repair. As yet another example, closer to the case studies described in the paper, a developer may use an Eflect-enabled Chappie to profile her Eflect-enabled Aeneas application in a shared environment. Such a combination may enable adaptive energy optimization with new knobs recommended in-situ in a server environment.

Acknowledgments. We thank Kenan Liu and Khaled Mahmoud for their help during the development. This project is supported by US NSF CNS-1910532.

REFERENCES

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 265–283.

[2] Babakol, T., Canino, A., Mahmoud, K., Saxena, R., and Liu, Y. D. Calm energy accounting for multi-threaded java applications. In *Proceedings of the 2020 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE* (Nov 2020), pp. 976–988.

[3] Baek, W., and Chilimbi, T. M. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI '10: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2010), pp. 198–209.

[4] Banerjee, A., Chong, L. K., Ballabriga, C., and Roychoudhury, A. Energy-patch: Repairing resource leaks to improve energy-efficiency of android apps.

IEEE Transactions on Software Engineering 44, 5 (2017), 470–490.

[5] 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.

[6] 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.

[7] Boston, B., Sampson, A., Grossman, D., and Ceze, L. Probability type inference for flexible approximate programming. In *OOPSLA 2015: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Oct 2015), pp. 470–487.

[8] 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.

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

[10] 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* (Nov 2018), pp. 703–713.

[11] Cohen, M., Zhu, H. S., Emgin, S. E., and Liu, Y. D. Energy types. In *OOPSLA '12: Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (Oct 2012), pp. 831–850.

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

[13] Dhiman, G., Marchetti, G., and Rosing, T. Vgreen: A system for energy efficient computing in virtualized environments. In *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design* (New York, NY, USA, 2009), ISLPED '09, Association for Computing Machinery, p. 243–248.

[14] Flinn, J., and Satyanarayanan, M. Energy-aware adaptation for mobile applications. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1999), SOSP '99, Association for Computing Machinery, p. 48–63.

[15] 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.

[16] Guo, L., Xu, T., Xu, M., Liu, X., and Lin, F. X. Power sandbox: Power awareness redefined. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 37:1–37:15.

[17] Hao, S., Li, D., Halfond, W. G. J., and Govindan, R. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), ICSE '13, IEEE Press, p. 92–101.

[18] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. Dynamic knobs for responsive power-aware computing. vol. 46, Association for Computing Machinery, p. 199–212.

[19] 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. vol. 48, Association for Computing Machinery, p. 661–676.

[20] Li, D., Tran, A. H., and Halfond, W. G. J. 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* (2014), pp. 527–538.

[21] Liu, K., Pinto, G., and Liu, Y. D. Data-oriented characterization of application-level energy optimization. In *Fundamental Approaches to Software Engineering* (2015), Springer Berlin Heidelberg, pp. 316–331.

[22] 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.

[23] Manotas, I., Pollock, L., and Clause, J. Seeds: a software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering* (2014), pp. 503–514.

[24] Merkel, A., Stoess, J., and Bellosa, F. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, Association for Computing Machinery, p. 153–166.

[25] Pathak, A., Hu, Y. C., and Zhang, M. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2011), HotNets-X, Association for Computing Machinery.

[26] 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.

[27] 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 *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2012), MobiSys '12, Association for Computing Machinery, p. 267–280.

[28] 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* (1998), ISLPED '98, pp. 76–81.

[29] Pinto, G., Castor, F., and Liu, Y. D. Understanding energy behaviors of thread management constructs. In *The 29th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'14)* (October 2014).

[30] Pinto, G., Liu, K., Castor, F., and Liu, Y. D. A comprehensive study on the energy efficiency of java's thread-safe collections. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016* (2016), pp. 20–31.

[31] Ribic, H., and Liu, Y. D. AEQUITAS: coordinated energy management across parallel applications. In *Proceedings of the 2016 International Conference on Supercomputing, ICS 2016, Istanbul, Turkey, June 1-3, 2016* (2016), O. Ozturk, K. Ebcioglu, M. T. Kandemir, and O. Mutlu, Eds., ACM, pp. 4:1–4:12.

[32] Roy, A., Rumble, S. M., Stutsman, R., Levis, P., Mazières, D., and Zeldovich, N. Energy management in mobile devices with the cinder operating system. In *Proceedings of the Sixth Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 139–152.

[33] 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 *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, Association for Computing Machinery, p. 164–174.

[34] Shen, K., Shriraman, A., Dwarkadas, S., Zhang, X., and Chen, Z. Power containers: An os facility for fine-grained power and energy management on multicore servers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 65–76.

[35] 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 *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2007), SenSys '07, Association for Computing Machinery, p. 161–174.

[36] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision, 2015.

[37] Yoon, C., Kim, D., Jung, W., Kang, C., and Cha, H. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, 2012), USENIX, pp. 387–400.

[38] Yoon, C., Kim, D., Jung, W., Kang, C., and Cha, H. Appscope: Application energy metering framework for android smartphones using kernel activity monitoring. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (USA, 2012), USENIX ATC'12, USENIX Association, p. 36.

[39] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. Ecosystem: managing energy as a first class operating system resource. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (2002), pp. 123–132.

[40] 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.

[41] Zhu, H. S., Lin, C., and Liu, Y. D. A programming model for sustainable software. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (2015), ICSE '15, IEEE Press, p. 767–777.