Energy Exchanges: Internal Power Oversight for Applications

Melanie Kambadur Martha A. Kim
Columbia University
{melanie,martha}@cs.columbia.edu

Abstract

This paper introduces energy exchanges*, a set of abstractions that allow applications to help hardware and operating systems manage power and energy consumption. Using annotations, energy exchanges dictate when, where, and how to trade performance or accuracy for power in ways that only an application's developer can decide. In particular, the abstractions offer* audits *and* budgets *which watch and cap the power or energy of some piece of the application. The interface also exposes energy and power* usage *reports which an application may use to change its behavior. Such information complements existing system-wide energy management by operating systems or hardware, which provide global fairness and protections, but are unaware of the internal dynamics of an application.*

Energy exchanges are implemented as a user-level C++ library. The library employs an accounting technique to attribute shares of system-wide energy consumption (provided by system-wide hardware energy meters available on newer hardware platforms) to individual application threads. With these per-thread meters and careful tracking of an application's activity, the library exposes energy and power usage for program regions of interest via the energy exchange abstractions with negligible runtime or power overhead. We use the library to demonstrate three applications of energy exchanges: (1) the prioritization of a mobile game's energy use over third-party advertisements, (2) dynamic adaptations of the framerate of a video tracking benchmark that maximize performance and accuracy within the confines of a given energy allotment, and (3) the triggering of computational sprints and corresponding cooldowns, based on time, system TDP, and power consumption.

1. Introduction

Today's computer systems have more complex and critical power and energy management needs than ever before. In 2011, 70% of the returned Motorola devices were due to battery life complaints attributable to applications [8]. Another recent study revealed that 65%-75% of the energy consumption of free applications is spent on third-party advertisement code [33]. In datacenters, energy- rather than time-based accounting may soon be the norm, as it has been shown benefit both datacenter owners and customers alike [19].

Power and energy efficiency have been first-class design goals for more than a decade [28]. At the base of the stack, hardware has become dynamically adjustable, offering a range of supply voltages, operating frequencies, and sleep states. In the middle, the operating system tunes the hardware based on the measured and expected needs of the applications running on top. For example, the Linux kernel has policies to manage processor idle states and frequency scaling [17], and work is in progress to develop power-aware scheduling algorithms that aim to schedule applications such that resource slow-downs can be made for longer periods or at a larger scale [15, 45, 25].

These hardware and operating system adjustments can proceed without any application-level changes. By some standards, this is desirable, as many application developers either cannot or will not modify their programs. However, keeping applications out of the energy management picture also limits the potential efficiency. As we contend in Section 2, application-level energy management not only can but must be used for optimal energy and power efficiency. Because hardware and operating systems do not have intricate knowledge about each application's needs, they must be conservative in their adjustments. In contrast, applications, knowing their own needs, can more aggressively trade performance and/or accuracy for power savings.

The *energy exchanges* presented here empower programmers to mandate when, where and how to trade performance and accuracy for power and energy use. Using the simple but expressive application-level directives provided by energy exchanges, programmers can implement feedback-driven changes to the application's behavior, for example, switching from precise to sloppy decoding in a video player as battery life declines, reducing parallelism in a smart phone game if other applications are elevating processing power, or bypassing some nonessential part of a robot's activity for a time so that it can run longer in between charges.

Energy exchanges offer several advantages over energy profiling and existing application-level energy management techniques. First, the exchanges are simple to use and integrate into existing programs and systems. They require no new hardware, operating systems, languages, or compilers. Instead, they require only short software annotations supported by a library extension. Second, unlike many software management strategies, e.g., compiler-inserted DVFS hints, that tune the same hardware knobs as the OS but in an application-specific way, energy exchanges allow the application to adjust itself. Moreover, they change its behavior based on online energy, power, and temporal feedback, so that a program trades performance or accuracy only when necessary. Finally, these exchanges are extremely flexible. They can be used to modify just about anything a developer might want to change about their program's behavior, offer energy and power usage feedback, and can trigger adjustments preemptively if desired. This expressivity means that a single energy-exchange augmented application will function across a range of different inputs, architectures, and degrees of parallelism.

Energy exchanges are usable now, via an open source library extension for C++ programs. The syntax and semantics of this library are presented in Section 3, along with several canonical examples that demonstrate the simplicity and breadth of use of the exchanges. Section 4 describes an implementation of energy exchanges. It uses a novel technique that monitors programmer indicated regions of interest on a per-thread granularity using just the system-wide hardware energy meters available on recent processors. The capabilities of the implementation are experimentally demonstrated in Section 5, where energy exchanges prioritize the energy of a game over a third-party advertisement, adapt simulation framerates to optimize quality while meeting a variety of pre-set energy budgets, and trigger computational sprints followed by precisely compensating cooldowns. The paper closes with a discussion of related work in Section 6.

2. Background

Before describing our new abstractions, we explain why applications need to help the hardware and OS manage energy and power for optimal efficiency. We also outline the prerequisites for an application to be able to do so.

2.1. Why Applications Must Help Manage Energy

Hardware power management techniques typically involve reducing the activity and therefore the power draw of various hardware components. Often this can be accomplished without affecting overlying applications, for example, by putting processors to sleep when they are idle [31]. However, similarly often power savings techniques do impact program performance, for example, if a processor oversleeps and a program must wait for it to wake up, or if the clock frequency is reduced while a processor is in use.

Operating systems and compilers can help by scheduling application-level resource requests to maximize the opportunity for hardware power savings, possibly even exploiting heterogeneous architectures [6]. However, because of the potential for losses in program performance, *any adjustments made at the hardware or system level must be conservative*. These lower levels of the system stack are missing two critical control mechanisms that prevent them from achieving peak energy efficiency. First, the hardware and OS cannot reduce the amount of resources requested by applications in the first place. Only applications have the ability to request fewer processing and memory resources, and, if they are judicious in their requests, both energy and power can be saved. Second, hardware and the operating system do not know when or by how much it is acceptable to slow down a program, which is why they must err on the side of caution. Compounding the caution, most existing power tuning controls operate on a whole-core or whole-socket granularity, with potentially more than one application sharing the core or socket. At times when it is appropriate for one application to trade performance for power, it may not be for the another, preventing the system from taking advantage.

Applications know their own needs, and can decide exactly when, where, and to what degree performance should be traded for power. Moreover, while the system can trade only performance for power, the application has a second currency at its disposal: accuracy. Absent application-level information, the system cannot interfere with the function of a program. For optimal efficiency, *applications must be engaged in power and energy management.*

2.2. Requirements of Application Level Energy Management

Tools to monitor application-level energy use [20, 42] and compile time energy optimizations [39, 18, 47] each have more than a decade of history. More recently, application-level management strategies have been proposed, for example EnerJ which approximates data types to save energy [37], and Eon which creates a new flow language for writing energy efficient software [40]. Despite this, application-level energy management has yet to enter mainstream software development. One reason for this is that application level energy management has not historically been as commercially viable as hardware and operating system level management techniques. Until recently, while people were willing to pay more for mobile devices with better battery life or for servers with lower energy costs, they generally did not consider the energy efficiency of applications. There was thus little incentive for developers to invest in making their code energy- or power-efficient.

However, opinions about software-level energy use have started to change. Embedded devices, now accounting for 98% of manufactured microprocessors [12], demand low-energy software, users have power monitor applications to identify energy hogs [33], and computer scientists are coming to the realization that energy can be better saved with software's help than without [13]. System-level energy savings techniques are ubiquitous and have many loyal users. Any application-level technique will need to offer substantial and complementary gains. Successful techniques must also be flexible and easy to use. An ideal application-level management solution should exploit conservation opportunities that the hardware and operating system cannot to *complement existing energy saving techniques* at lower levels of the system stack; *be easy for programmers to use*, ideally requiring no special hardware, new operating system, or new languages, and not necessitating complete program re-writes; *be as agnostic as possible to the underlying architecture and inputs* so that developers do not have to manage multiple versions of their software; and *consider and account for other applications* running on the same machine that may also be managing their own energy. To our knowledge, no current solution fulfills all of these requirements, but as the next sections show, energy exchanges attempt to do so.

3. Energy Exchanges

The primary goal of energy exchanges is to empower applications to trade performance and/or accuracy as needed to improve efficiency or keep power and energy consumption within pre-determined bounds. Energy exchanges offer short software annotations to be inserted by software developers into new or existing programs. This section describes the semantics of these annotations using, as needed, the syntax offered by our C++ library implementation which is the subject of Section 4.

3.1. Audits, Budgets, and Usage Records

Runtime power-performance and power-accuracy trade-offs require additional code to set up runtime resource monitoring and to adjust programs as a result of these measurements. Both additions are simplified with two new abstractions: *audits* and *budgets*. Audits and budgets both enable programs to observe the runtime, energy, and power consumption of arbitrary regions of code. Each audit or budget is associated with a block of source code, enclosed in a pair of corresponding curly braces as shown on the right of Figure 1. At runtime, these braces delineate the region of execution to which an audit or budget applies. These regions can perform arbitrary computation, including nesting other audits and budgets or spawning new threads. Nested audits and budgets will also count towards the outer region's usage, while child threads' activity will be attributed to the enclosing region until such time as the thread terminates or the original thread exits the block. In our C++ implementation of energy exchanges `audit` and `budget` should be though of as new reserved keywords.

Audits and budgets differ with respect to how they trigger programs to react to their measurements. Audits are passive measurements, allowing programmers to make adjustments only after the associated region has completed. When the region finishes, the programmer receives a usage report, via a command of the form `report(usage_t* u);`. This command effectively declares an instance of `usage_t*` named `u` which, when the command finishes, will be a C++ `struct` populated with information about the associated region, namely the energy, average power, and runtime, in joules, Watts, and seconds respectively as shown in Figure 1. The application is then free to use this information to make any desired adjustments.

In contrast, budgets are preemptive. When initiated they set a cap consisting of a metric (energy, average power, or time), a value that the metric should not exceed, and a unit of measurement. During execution of the associated region, if the metric meets the indicated cap value, execution of the region will be immediately terminated, including any child threads spawned therein, and control transferred back to the `if_exhausted` block that follows the original budget. There, the programmer may specify his or her reactionary adjustments.

3.2. A First Example

As a first example, we show how mobile applications can use energy exchanges to prevent energy overconsumption by third-party advertisement code. In this scenario, the main application is a game that runs repeated rounds of play and displays an ad in between each round. The developer would like each advertisement to consume no more than 20% of the energy consumed by the previous round of play, and if it does, she would like the ad to be killed. Additionally, she wants to track how many times an advertisement went over its allotted budget. The code below shows how energy exchanges concisely express all of this functionality:

```
1  int overages = 0;
2  while (! quit) {
3    audit {
4      game.play_round();
5    } (usage_t *round_use);
6    double energy = round_use->energy;
7    budget (ENERGY, energy*0.2, J) {
8      ad.run();
9    } if_exhausted (usage_t *ad_use) {
10     // ad is implicitly killed if control jumps here
11     overages++;
12   }
13 }
```

First the developer initializes a variable `overages` to count the number of over-consuming ads. Next, the game loop starts to be stopped only by a user quitting the game. Each round of play (line 4) is wrapped by an audit which captures the round's dynamic energy, power, and runtime in an instance of `usage_t*` called `round_use` (line 5). This usage record is accessed (line 6) to calculate an energy budget for the advertisement equal to 20% of the usage of the previous round. The budget wraps calls to an advertisement which runs as a separate thread (lines 7-9). Following the budget is an `if_exhausted` block (lines 9 - 12) which tells the program what to do in the event an ad exceeds the pre-set budget. Exhaustion immediately terminates code initiated within the budget region, in this case the advertisement thread, and control jumps to the inside of the top of the `if_exhausted` block. There, the developer updates the overages counter, after which, exhaustion or not, the outer round iteration loop resumes.

Alternatively, if the developer did not wish to interrupt energy-hogging advertisements, she could have used an audit in place of the budget. Had she done that, rather than stopping the ad mid-execution, she might have it to complete, but restricted runs of future ads based on the energy consumed by

```
typedef enum {
  AVERAGE_POWER,
  ENERGY,
  TIME
} metric_t;

typedef enum {
  W, mW, uW,
  J, mJ, uJ,
  s, ms, us
} unit_t;
```

```
extern double TDP, MIN_POWER, MAX_POWER;

typedef struct {
  double energy; // in J
  double average_power; // in W
  double wall_time; // in s
} usage_t;
```

```
audit {
  // code to profile
} report(usage_t *u);
// reaction to profiled usage

budget (/* metric_t */ m, /* double */ v, /* unit_t */ u) {
  // code for which budget has been set, to be
  // preempted if budget is exhausted
} if_exhausted (usage_t *u) {
  // reaction to exhausted budget
}
```

Figure 1: Energy Exchanges Abstractions in C++ Library Form. With the new syntax provided by our initial implementation, audits or budgets can be used to profile power, energy, and time either passively or preemptively. The code which follows resource profiling is a reaction to the runtime measurements stored in the usage records, either a reduction to performance or accuracy, or some other side effect such as a data backup.

earlier ad runs.

3.3. Four Steps to Add Exchanges to Any Application

Energy exchanges are designed to be simple to learn and quick to add into existing programs. Users must consider the following four factors when inserting exchanges:

1. Decide where in the program's source code to curb power and energy. This can be decided via profiling — potentially through the use of preliminary audits — or based on a developer's prior knowledge of an application. In Section 5.2 where we write energy exchanges into a lengthy and unfamiliar benchmark program, we simply inserted audits in `main()` to quickly find the energy hotspot.
2. Determine whether an audit or a budget is more suitable. If preemption is a desired part of a power and energy use based adjustment, a budget should be used; otherwise audits should be used.
3. Identify the power, energy, or execution time conditions that should trigger a runtime change. To match a range of potential uses, energy exchanges offer several ways to set these conditions. As in the previous example, energy, power, or time limits can be set relative to earlier measurements. They can also be set relative to the built-in, platform-specific global variables `TDP`, `MIN_POWER`, or `MAX_POWER`, or as absolute numbers with a variety of preset units as shown in the first two columns of Figure 1.
4. Elect how to change the behavior of the application based on the observed resource usage thus far. While performance and accuracy trade-offs are going to be specific to individual applications, we have identified several canonical types which are described below. In our case studies we found that one or a combination of these standard styles was often sufficient to express the desired management policy.

3.4. Canonical Uses of Energy Exchanges

Although it may not be immediately obvious, many programs can trade performance or accuracy for energy or power, usually without major revision. In many cases, a trade can be made by modifying just a single variable and adding a few lines of annotations. Consider a GPS or bluetooth application on an embedded device that normally polls for a signal every second. Energy exchanges can switch the polling period to once in a minute, say if the more frequent polling pushed power consumption to within 95% of the peak device power. With the higher delay time between polls, the embedded device would have time to enter deeper processor or memory sleep states, saving a significant amount of power and energy.

```
poll_period = 1; // originally, poll GPS once per second
while (gps_on) {
  budget(POWER, MAX_POWER*0.95, W) {
    wait(poll_period);
    location = poll_signal();
  } if_exhausted(usage_t *u) {
    poll_period = 60; // poll GPS once per min if power is a constraint
  }
}
```

In the preceding example, some small amount of functionality was periodically skipped within a loop. Functionality could also be skipped at a larger scale, as in the first example when the execution of an advertisement was cut short. In that example, the resource allocation of the low priority advertisement was pegged to the consumption of the high priority game. Instead, what if a high priority application is to run after a low priority application? In that case, it might be appropriate to allow the low priority application to run normally, but to reserve a certain amount of energy for the higher priority application's later use.

As an example, consider a sleep-aid mobile application that plays soothing nature sounds through the night, and later sounds a gradual alarm clock in the morning. The nature sounds should play as long as possible, but it is imperative that sufficient battery charge remain until morning for the alarm to sound. Energy exchanges could reserve the necessary energy (equal to `alarm_needs`) with a `budget` region that stops the nature sounds as soon as the reserve is approached:

```
double total_budget = ...;
double alarm_needs = ...;
// reserve alarm_needs energy to play an alarm clock in the morning
budget (ENERGY, (total_budget-alarm_needs), J) {
    // play nature sounds as long as possible
} if_exhausted(usage_t *u) {
    // enter extreme low power state until time for alarm clock
}
```

Although our current library does not support it, a different implementation of the energy exchange abstractions could help an operating system to manage multiple applications at a time. *Power vampires* are household appliances that draw standby power while they are plugged in but not in use. Similarly, mobile applications may draw power while they are running in the background but not interacting with users. To get rid of mobile power vampires, an operating system could wrap a budget around an application any time it begins to run in the background. If an application consumed more than, say 10% of the overall system power while it was supposedly sleeping, the operating system could kill the offending application.

Rather than reducing the functionality of an application, a swap of equivalent but differently implemented algorithms could be made. Certain fast running algorithms draw high amounts of power; in power constrained situations such applications could be swapped for slower but less power-hungry applications. At a less drastic scale than full algorithmic swaps, energy exchanges could be used to manage energy efficient *algorithmic compositions*. Just as optimally performing algorithms might be amalgamations of multiple algorithms (for example, `std::sort` uses merge sort followed by insertion sort when the sublists become small [2]), multiple algorithms may need to be combined for optimal power savings. Energy exceptions could help make amalgamations more flexible, taking into account architecture specific power measurements, input size, or any other required factors.

Parallelism is another classic power-performance trade-off that can be readily exploited with energy exchanges. As parallelism increases, power typically also increases and runtime decreases. Conversely, as parallelism decreases, power typically decreases and runtime increases. Energy exchanges can adjust parallelism dynamically depending on current power and performance requirements, as demonstrated in Section 5.3.

Although energy exchanges have been presented so far as a means to make changes within application, they could also be used to trigger side effects that change something outside of an application. Suppose that a video player wants to warn users when a video has consumed a significant fraction of the device's total battery life. Assuming a battery capacity of 88.2kJ, or 42.5Wh as in the iPad 2 [40], the exchange code snippet below issues a user warning for any video playback that consumes more than 50% of a full charge.

```
1  double full_charge = 88200;
2  budget (ENERGY, 0.5*full_charge, J) {
3    video.play(0.0); // play video from start
4  } if_exhausted (usage_t *u) {
5    printf("Video consumed 50\% of battery\n");
6    video.play(u->wall_time); // restart video after warning
7  }
```

Note that in line 6, the wall time stored in the usage record collected by the if_exhausted clause is accessed to restart the video where it was cut off.

Another side effect option is to backup intermediate data to a file. Cloud computing services such as Amazon's EC2 allow users to pay for compute time by the hour and type of computation (for example, high-I/O or high memory usage cost more) [1]. With energy use dominating datacenter costs, some have suggested that charging users per-Watt is a fairer pricing policy [41]. If a user pre-pays for a specific amount of energy or power, they will want to make the most of it by computing up to the budget. Just as in the sleep-aid example, a small amount of energy can be reserved to backup work just before a budget expires. Imagine that the developers of a contextual image classifier have purchased 5 MJ of cluster service to run their classifier. The audit in the following code snippet accumulates the computational energy consumed as each feature vector in the classifier is processed:

```
1  double backup_energy = ... // energy_needed_for_backup
2  double compute_e = (5e6 - backup_energy);
3  for (i=0; i<NUMPIXELS; i++) {
4    audit {
5      feature_vectors[i] = calc_feat_vector(pixel_vector[i]);
6    } report(usage_t *per_pixel);
7    compute_e -= per_pixel->energy;
8    if (compute_e < per_pixel->energy) {
9      printf("Stopping execution to backup after \
10       processing %d pixels\n", i);
11     save_to_file(feature_vectors, ``backup.dat'');
12     break;
13   }
14 }
```

When the energy budget remaining is less than the backup energy needed plus the amount of energy used to process the last pixel (lines 7-8), execution of the classifier stops and a backup snapshot is saved (line 11).

4. Implementation

NRGX (pronounced *energy-ex*) is a proof of concept, software-only implementation of the energy exchange interface described in Section 3.

4.1. Overview and Energy Accounting

At a high level, NRGX monitors several activities: an application's active threads including individual usage, pending audits and budgets, and system wide utilization and energy consumption – the latter as exposed by standard hardware energy meters. To track the application's live threads NRGX interposes on calls to pthreads which create or destroy threads. NRGX) is notified of opening and closing audits and budgets directly via the application's use of the exchange directives described in Section 3. When the application first opens an audit or budget, NRGX creates a monitor thread that periodically takes the following readings:

- E_{sys}: system-wide energy reading obtained from Intel's *Running Average Power Limit (RAPL)* [9]
- U_{sys}: system-wide CPU time from `/proc/stat`
- U_{tid}: CPU time for each application thread *tid* from `/proc/<pid>/tasks/<tid>/stat`

The frequency of these readings is configurable, but for the experiments presented in the following section ranged from 10-100 Hz.

The core innovation of the NRGX implementation is in using a *single, coarse energy meter reading to track energy consumption within individual applications*. It is a two step process. First, with each new set of readings, we compute the energy usage of all existing individual threads in proportion to their share of total system usage. Specifically energy for the i^{th} sample of thread *tid* is:

$$E_{tid,i} = \frac{U_{tid,i} - U_{tid,i-1}}{U_{sys,i} - U_{sys,i-1}} \times (E_{sys,i} - E_{sys,i-1})$$

NRGX maintains an application thread tree and information about the nested structure of the audits and budgets created by each thread. It uses this information in the second phase where each thread's sampled energy, $E_{tid,i}$, is accumulated in the usage record of any open audits or budgets of the thread itself (*tid*) and its ancestors.

These same structures are used to identify which threads should be terminated when a budget is exhausted. Rather than energy usage flowing upwards towards the enclosing audits or budgets, the exhausted budget propagates downwards killing all subthreads spawned during the violating region. When done NRGX uses an exception to transfer control to the `if_exhausted` block.

4.2. Usage Logistics and Limitations

NRGX is open source and can be downloaded from `http://redacted`. After the user has added exchange directives to their application as described in Section 3 and demonstrated in Section 5, he or she then must link against NRGX i.e., `-lnrgx`, and ensure that the NRGX shared object file is loaded first, i.e., via `LD_PRELOAD or LD_LIBRARY_PATH`. The only other requirement of the current implementation is that, once compiled, the application needs to be executed by a sudo-er. This is because Linux does not currently expose even read-only access to the RAPL registers to non-sudoers. We believe this work is an example of why it would be beneficial for Linux to do so in the future.

Because the RAPL registers are updated every 1ms and the CPU usage every 10ms (at 100 jiffies/sec), they cannot be used to audit or budget sub-10ms windows of execution. However, to make a difference in energy consumption the application needs to adjust much larger chunks of runtime, so this is not a problematic limitation. On the low end of sampling frequency, samples need to be taken frequently enough to detect overflow – e.g., the RAPL counters overflowed roughly every 10-20 seconds on our machine – and to promptly detect and address exhausted budgets. Sampling at 10-100 Hz met all of these concerns and incurred negligible time or power overhead above the un-monitored application.

5. Experimental Demonstrations

We now demonstrate the use of energy exchanges with three full-scale case studies that use NRGX for a range of purposes: prioritizing the energy of a minesweeper game over third-party advertisements in Section 5.1, framerate adaptation to optimize performance for a given energy budget in Section 5.2, and enforced cooldowns after computational sprints in Section 5.3. All of the experiments in this section use a Dell PowerEdge R420 server with a dual socket Intel Sandybridge E5-2430, each with 6 cores and 12 threads, and a total of 24GB of DRAM. The machine has Linux kernel version 3.9.11 and Ubuntu 12.04.2. Intel sleep states [31], turbo boost [7], and the ondemand frequency governor [32] are turned *on* for all the experiments to show that energy exchanges easily combine with existing energy saving techniques at the system and hardware level.

5.1. Restricting Advertisement Energy

A recent study estimated that 77% of the top free applications in the Google Play store were ad-supported [22]. With free applications making up 91% of total downloads [16], most applications in use are financially supported by advertisements. Mohan et al. [26] found that on average, 23% of an ad-supported application's overall energy is consumed by the advertisements. Coupled with the fact that 70% of all Motorola devices were returned as a result of resource-greedy applications [8], developers and mobile providers alike have an incentive to ensure that advertisements consume only their fair share of energy.

Energy exchanges can help application developers manage energy priorities between their applications and advertisements, even if their applications call upon third-party advertisements with unpredictable energy demands. Using similar syntax to the first example of the paper (Section 3.2), we put energy exchanges into a real application that calls a pathologically unpredictable simulated advertisement of our own devising. The game is a text-based minesweeper game, from [24], and the simulated ads are pthreads that perform random amounts of computationally and I/O intensive busywork. The minesweeper game has potentially long rounds of play with unpredictable duration. Rather than budgeting the advertisement's energy use relative to each round of play as in Section 3.2, we set a timer to check-in on the game's energy consumption every two seconds. The energy consumption is measured via an audit, then fed into a budget that wraps the advertisement threads as Section 3.2 showed. The budget restricts advertisements to 20% of the energy consumed by the previous two seconds of game play. If an advertisement exceeds this budget, it is terminated, and another advertisement will not begin until another two seconds have been played.

Figure 2 shows resource measurements of two versions of the game being played by a user. The first version (at left) has no energy restrictions on the advertisements, while the second version (at right) restricts advertisements with budgets as previously described. Both graphs show the energy in Joules consumed by the minesweeper game in two second intervals over 50 total seconds of play. In our experiments, the advertisements sometimes slightly exceeded 20% of the

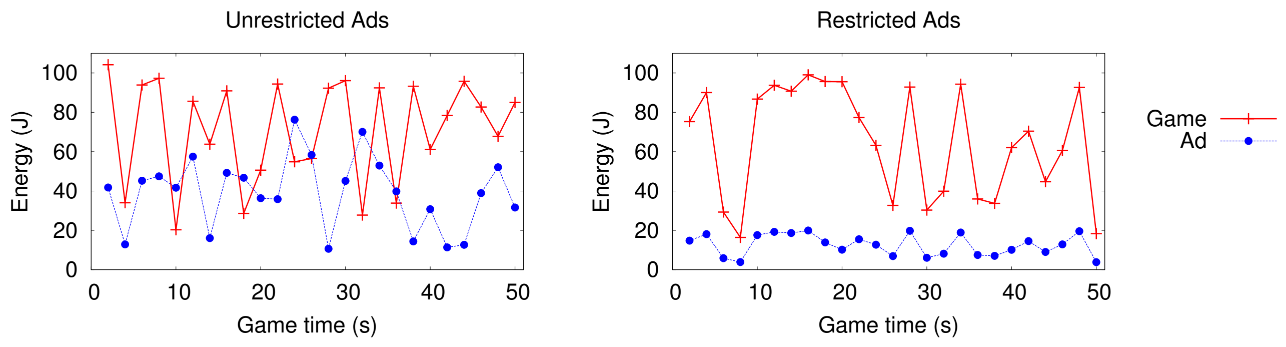

Figure 2: Energy exchanges can prioritize the energy of different parts of an application. In this example, calls to a simulated advertisement that computes semi-random amounts of work were inserted into an open-source minesweeper game. Both graphs show the energy consumed by the game and the advertisement over a series of two second intervals as a user plays. During the game depicted at left, advertisements are allowed to run freely. At right, energy exchange budgets kill any over-consuming advertisement threads to restrict advertising to 20% of the energy consumed by the game.

energy of the game even when restricted. This slight increase, which is always less than 1%, is a result of the latency between the `pthread_cancel` call made by NRGX on behalf of the budget, and the actual death of the advertisement thread.

5.2. Adaptive Framerate

Today's hand-held and embedded devices are expected to complete increasingly complex computational tasks with limited energy budgets. Mobile devices, for example, have moved well beyond the basic text processing and voice communication of fifteen years ago and now download and process media, browse the internet, and support graphic-rich games. At the same time, they are expected to function on extremely small power and energy budgets in order to operate for long durations on small, light-weight batteries.

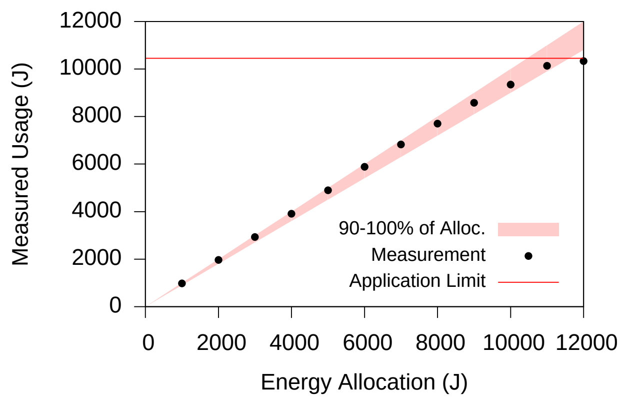

Figure 3: Energy exchanges allow code to adapt to meet a program's energy goals. This graph shows the results of using exchanges to augment `bodytrack` so that it maximizes quality while never exceeding various allocated energy budgets. The application adapts by dropping frames as needed based on the energy consumed so far, the total allocation, and the number of remaining frames.

Sometimes the best strategy to meet high computational demands under strict energy budgets is to reduce the functionality or accuracy of application services provided. To demonstrate real-world application level accuracy trade-offs, we augmented the `bodytrack` application from the Parsec benchmark suite [5] with energy exchanges. `Bodytrack` tracks the poses of a person recorded on multiple video cameras; the majority of this work occurs in a for-loop within the main function, which processes frames one at a time. We assume a scenario where bodytrack is given a strict energy allocation to complete its work. The goal is to maximize the performance and accuracy of the tracking without exceeding this strict budget allocated. The following code snippet uses an audit to adapt `bodytrack` to meet any arbitrary energy allocation:

```
1  #include "nrgx.h"
2  #define ALLOCATION 2000 // could be define as any value
3  ...
4  double per_frame = ALLOCATION/frames;
5  int framestep = 1;
6
7  for (int i = 0; i < frames; i=i+framestep) {
8    audit {
9      // DO FRAME PROCESSING
10   } record (usage_t *this_frame);
11   double energy = this_frame->energy;
12   ALLOCATION -= energy;
13   // if frame did not take 90−100% of allocation, reset framestep
14   if ((energy > per_frame) || (energy < 0.9*per_frame)) {
15     per_frame = (ALLOCATION/(frames - i));
16     framestep = (int)ceil(energy/per_frame);
17   }
18 }
```

The `ALLOCATION` constant (line 3) could be converted into a variable indicating an energy allocation based on the target platform and remaining battery charge. To keep track of the program's progress relative to the allocation, a `per_frame` allocation is calculated. The program begins by processing videos at an initial `framestep` of 1 (lines 6-8). As each frame is processed, an audit records its resource use (lines 9-11) and reduces the total energy allocation by the amount just consumed. The program then checks if the recorded use was within 10% of the per frame allocation to see if processing is on target to complete within the allocation (line 14). If it is not on target, the per frame allocation is updated, and the framestep is adjusted so that frames are dropped. On the other hand, if processing is more efficient than necessary, the framestep is reduced to improve quality.

Relative to the 11,000 lines of code in the entire bodytrack program, these additions are tiny. However, they are extremely powerful: with this handful of lines, the application now self-adjusts its behavior, making it possible to adapt the energy consumed by the program to any level desired. Figure 3 shows the actual energy consumed by the augmented bodytrack when given a range of energy allocations between 1000 and 12000 Joules. The input size of bodytrack was set to native and the `-O3` compiler flag was used for all of the experiments. The figure shows that the energy consumption always meets, but never exceeds the given allocation for all trials. Only as the allocation nears the maximum possible energy of the program (i.e., the energy consumed when the framestep is always equal to 1) do the experimental results veer away from ideal.

5.3. Power Sprints

This next use of energy exchanges is based on the idea of computational sprinting [35, 34]. In both of these works, Raghavan et al. suggest that some devices may be able to temporarily operate at higher power dissipations, and consequently higher temperatures, than would be allowed for a sustained period of time. If sprinting is done right, programs can briefly and strategically execute above TDP to execute more quickly or appear more responsive without overheating the chip.

Modulating suitable sprint and cooldown times is non-trivial. In their original work [35], the authors suggest (in Section 4.5) that an appropriate cooldown period should be based on TDP, or the thermal design point of the chip the program is running on, and should also factor in the length of the sprint period and the average power consumed during the sprint. They suggest the following formula for a sprint-compensating cooldown:

$$\textit{cooldown time (s)} = \frac{\textit{average sprint power (W)}}{\textit{TDP (W)}} \times \textit{sprint time (s)}$$

In their later work, the authors implement a hardware/software testbed for the sprints [34]. When they show sprinting for extended computations, they use solely time-based triggers to switch modes. In particular, one experiment used a fixed 5 second sprint period and 12.5 second cooldown.

With energy exchanges it is feasible to factor TDP and runtime power measurements to calculate the precise duration of cooldown required after a particular sprint. NRGX exposes TDP, and the power consumption during a sprint can be audited.

To compare time- and power-based sprinting, we implemented a C++ microbenchmark that searches a very long string for a particular substring. Substring searching is used in many important real world applications, such as genomic analysis and satellite image processing. Our microbenchmark has two modes: low power, in which a single thread searches for strings, and high power, in which N threads search concurrently. In both time- and power-based sprinting, we fix the sprint time at 5 seconds as in the earlier work. Time-based sprinting cools for 12.5 seconds no matter what, while power-based cools according to the actual sprint power using the formula above.

```
1  #include nrgx.h
2  ...
3  int num_threads = MAX_THREADS; // set to 24 threads
4  double time_balance = SPRINT_PERIOD; // set to 5 seconds
5  double power_consumed = 0.0;
6  double measurements = 0.0;
7  bool last_was_sprint = true;
8
9  while (strings_checked < TOTAL_STRINGS_TO_CHECK) {
10
11   audit {
12     // num_threads search for substring
13   } record (usage_t *use);
14
15   time_balance -= use->wall_time;
16   power_consumed += use->average_power;
17   measurements++;
18
19   if (time_balance <= use->wall_time) {
20     if (last_was_sprint) {
21       double avg_power = power_consumed/measurements;
22       // calculate new cooldown balance:
23       time_balance = SPRING_PERIOD* (avg_power/TDP);
24       power_consumed= 0.0;
25       measurements = 0;
26       num_threads = 1;
27       last_was_sprint = false;
28     } else {
29       time_balance = SPRINT_PERIOD;
30       power_consumed= 0.0;
31       measurements= 0;
32       num_threads = MAX_THREADS;
33       last_was_sprint = true;
34     }
35   }
36 }
```

The above code shows the implementation of that formula with energy exchanges. The program starts with a sprint, initializing a `time_balance` to 5 seconds (line 4) and setting MAX_THREADS to work searching, one region apiece. This work is wrapped in an `audit` (lines 11-13), which captures the average power and wall time. The wall time is subtracted from the running time balance (line 15), and the power the average power is accumulated (line 16) for later use. If the time balance has not been exhausted, and will not be exhausted before the next round of searches completes (line 19), parallel search continues. When the sprinting runs out of time, a new compensating time balance is calculated according to the formula (line 23), and the number of threads is set to 1 (line 26). Then, the program resumes searching in the serial, low-power mode until the calculated cooldown time budget is exhausted. At that time, the sprint time balance is restored to the original 5 seconds (line 29), and the number of threads is reset to 24 (line 32) to restart the high-power sprint mode.

Figure 4 shows four experiments which search a 120*B* char-

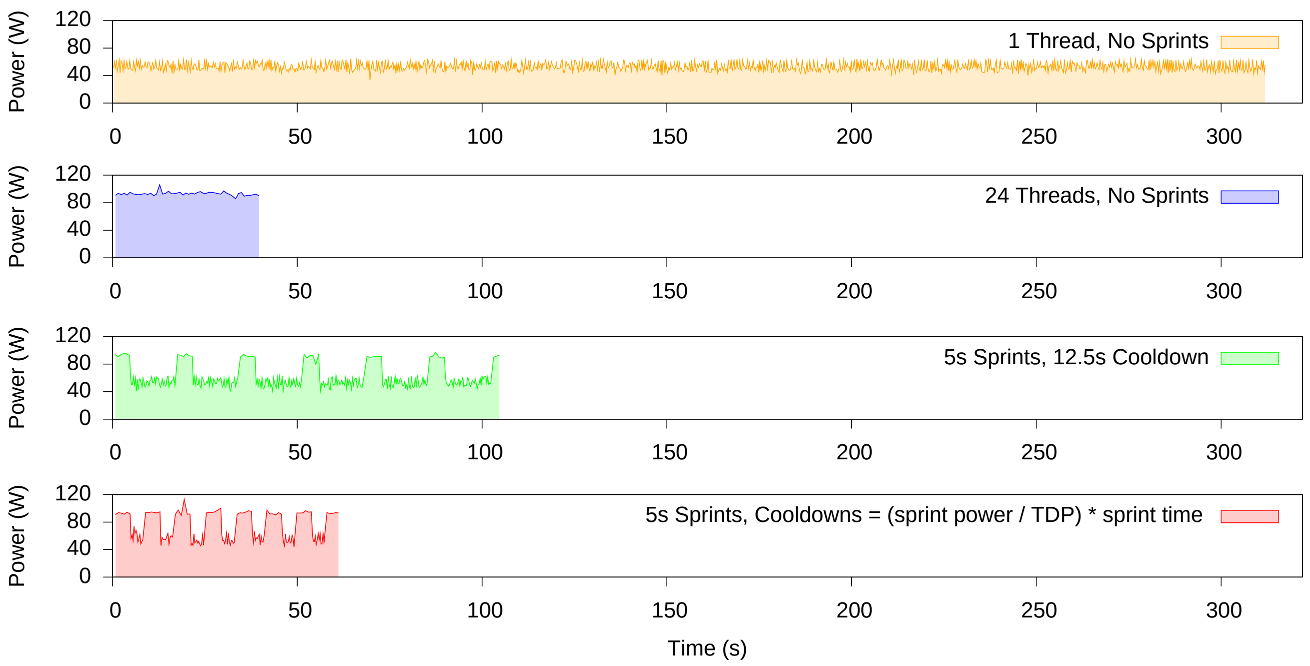

Figure 4: Energy exchanges can manipulate power-performance trade-offs, for example with the use of *power sprints*. Based on the idea that some devices can withstand only short bursts of high power dissipation, this experiment demonstrates how a substring search benchmark's parallelism can be varied and the power consumption dynamically monitored to optimize the windows of high and lower power computation.

acter string for an 8-character substring. The first graph in the figure shows power measurements for the duration of the program when it was run with 1 thread and no sprinting. The second graph shows the program's power over time when it had access to 24 processing threads, again with no sprinting. The third graph emulates the experiment from [34], following 5 second sprints with 12.5 second cooldowns. Finally, the fourth graph sprints using energy exchange augmented code as above, dynamically basing cooldown length on the previous sprint's power, duration, and the systems TDP.

Note that the average power is about 40-50 Watts higher with 24 threads than with 1. As far as energy, the increased power is more than offset by the performance gains, with the fully parallel version 8 times as fast as the serial version.

Comparing sprint policies, the fixed-cooldown policy takes 104 seconds to complete, while the power-based cooldown requires only 61, with the faster runtime attributable to the shorter cooldown periods.

6. Related Work

Energy management is not a new or uncrowded field. However, energy exchanges offer an important feature that profilers, application-specific system-level techniques, and existing application-level techniques do not provide. For the first time, energy exchanges let programmers easily make performance and accuracy trade-offs based on runtime feedback of runtime energy and power measurements of programmer-determined regions of interest.

Intra-application power measurement. A large body of research profiles application level power and energy, including tools such as eprof [33], PowerAPI [30], and JouleMeter [21]. Energy exchanges can also profile intra-application power and energy consumption, but go one step further by feeding these profiles back to the program so that it can make energy and power saving adjustments.

Application-specific system-level energy management. Dating back to the 1990s, operating systems researchers recognized energy as an emerging constraint and laid out cases for treating energy as another resource to be managed [43, 29]. ECOsystem [48] allocates *current*cy [49] to each process on the basis of a user-set target battery life and application priorities. Processes spend currentcy by using the CPU, memory, and disk, with balances managed using resource containers [4]. More recently, Cinder [36] introduced per-thread resource management, while ErdOS [44] uses strategic offloading to achieve battery lifetime goals, and DYNAMO [27] uses specialized middle-ware, a new operating system, and new hardware to dynamically scale CPU voltage and to adjust the backlight of mobile devices. Hardware and firmware power capping, as in IBM's Power7[46] or Intel's RAPL [10] chips, can limit the peak or average power usage of individual hardware components, such as memory or a single core, over a short or long period of time. As the "running average power limit" name suggests, RAPL allows the user to set a power limit and a period of time, creating an energy target which the hardware will endeavor to meet [9]. Intel has, among other things, applied support vector machines to correlate power caps with response time and throughput, in order to more efficiently meet SLAs [11]. Other techniques, such as power containers [38], rely on models to estimate the power and energy contributions of individual requests, in order to isolate and throttle power-hungry requests. These techniques are "enforced from below" making it impossible for the capped code to override the cap. Unlike these works, our energy exchanges allow programmers to incorporate energy and power usage feedback directly into their applications.

Adapting application behavior for energy savings. A number of existing works explore application-level techniques for adapting program behavior to conserve power or energy. The Odyssey/PowerScope system [14], an early entry in the field, extended Linux with a specialized file system to trade source code specifications of accuracy for energy savings. The Green compiler framework [3] has programmers write new energy efficient versions of functions and loops, each accompanied by a quantitative QoS budget that allow the compiler to guarantee energy savings at acceptable QoS costs. Eon is an energy-aware language [40] that requires new flow-state programs to be written and annotated by the level of power they will consume (e.g., high or low). The Eon runtime system then finds the most energy-efficient flow through the program. EnerJ [37] extends a Java compiler so that application level annotations can denote approximate data types. The annotations help a runtime system and specialized hardware choose appropriate energy saving execution strategies. Similarly, Flikker [23] introduces specialized hardware that trades energy consumption for data integrity based on application-level annotations. Our work differs from these approaches in two significant ways. First, it is much simpler to integrate into existing programs and systems, requiring no new languages or compilers, and no added middle-ware, no operating system changes, and no new hardware. Second, while the other works allow applications to have some influence over how energy is saved, none of them allow programmers to integrate energy and power feedback into their source code as energy exchanges do, resulting in fewer options for performance and accuracy trade-offs.

7. Conclusions

Given energy's status as a precious commodity, many have ideas about how to police its use. Solutions for power and energy management abound, from hardware to the operating system to the virtual machine and the compiler. For the first time, we allow applications to react to and manage their own power and energy consumption by trading performance and accuracy. The energy exchanges abstractions presented in this paper provide programmers simple mechanisms and a simple interface to measure a program's energy, power, and timing, and feed these measurements back into the program to make adjustments as it executes. Among other advances, energy exchanges empower applications to direct their own power sprinting, backup work before a battery dies, or prioritize the energy use of one thread over another. Energy exchanges also allow adaptive modifications of frame or pixel processing rates, switches between low power and high performance algoithms, and adjustments of parallelism. All of these features are immediately available as an open source, software-only C++ library which runs on commodity hardware.

References

[1] Amazon Web Services, Inc. Amazon ec2 pricing, 2013. http://aws.amazon.com/ec2/pricing/.

[2] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. *PetaBricks: a language and compiler for algorithmic choice*, volume 44. 2009.

[3] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 198–209, June 2010.

[4] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 45–58, 1999.

[5] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 225–236, 2012.

[7] James Charles, Preet Jassi, Narayan S Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the intel® core^{TM} i7 turbo boost feature. In *2012 IEEE International Symposium on Workload Characterization*, pages 188–197. IEEE, 2009.

[8] ComputerWorld. Motorola CEO: Open android store leads to quality issues. http://www.computerworld.com.au/article/388831/motorola_ceo_open_android_store_leads_quality_issues/.

[9] Intel Corporation. Intel 64®and IA-32 architectures software developer's manual. http://download.intel.com/products/processor/manual/253669.pdf.

[10] H. David, E. Gorbatov, Ulf R. Hanebutte, R. Khanna, and C. Le. RAPL: Memory power estimation and capping. In *International Symposium on Low-Power Electronics and Design*, pages 189–194, August 2010.

[11] Martin Dimitrov, Kshitij Doshi, Rahul Khanna, Karthik Kumar, and Christian Le. Coordinated optimization: Dynamic energy allocation in enterprise workload. *Intel®Technology Journal*, 16:32–51, August 2012.

[12] Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009.

[13] Hadi Esmaeilzadeh, Ting Cao, Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. Looking back and looking forward: power, performance, and upheaval. *Communications of the ACM*, 55(7):105–114, 2012.

[14] Jason Flinn and Mahadev Satyanarayanan. Energy-aware adaptation for mobile applications. *ACM SIGOPS Operating Systems Review*, 33(5):48–63, 1999.

[15] 'Iñigo Goiri, Ryan Beauchea, Kien Le, Thu D. Nguyen, Md. E. Haque, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. Greenslot: Scheduling energy consumption in green datacenters. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2011.

[16] Matt Hamblen. Mobile app download tally will soar above 102b this year. Computer World, 2013. http://www.computerworld.com/s/article/9242516/Mobile_app_download_tally_will_soar_above_102B_this_year.

[17] Tate Hornbeck and Peter Hokanson. Power management in the linux kernel. 2011.

[18] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. *ACM SIGPLAN Notices*, 38(5):38–48, 2003.

[19] V. Jimenez, R. Gioiosa, F.J. Cazorla, M. Valero, E. Kursun, C. Isci, A. Buyuktosunoglu, and P. Bose. Energy-aware accounting and billing in large-scale computing facilities. *IEEE Micro*, 31(3):60 –71, May-June 2011.

[20] Ismail Kadayif, M Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Anand Sivasubramaniam. Eac: a compiler framework for high-level energy estimation and optimization. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 436–442, 2002.

[21] Aman Kansal and Feng Zhao. Fine-grained energy profiling for power-aware application design. *SIGMETRICS Performance Evaluation Review*, 36:26–31, August 2008.

[22] Ilias Leontiadis, Christos Efstratiou, Marco Picone, and Cecilia Mascolo. Don't kill my ads!: balancing privacy in an ad-supported mobile application market. In *Proceedings of the Workshop on Mobile Computing Systems & Applications*, page 2, 2012.

[23] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 213–224, March 2011.

[24] Marek Marczykowski and Krzysztof Sachanowicz. The saper project (a minesweeper game). Version X.0.14, 2013. http://marmarek.w.staszic.waw.pl/saper/.

[25] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 153–166, 2010.

[26] Prashanth Mohan, Suman Nath, and Oriana Riva. Prefetching mobile ads: Can advertising systems afford it? In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 267–280, 2013.

[27] S. Mohapatra, N. Dutt, A. Nicolau, and N. Venkatasubramanian. DYNAMO: A cross-layer framework for end-to-end QoS and energy optimization in mobile handheld devices. *Journal on Selected Areas in Communications*, 25(4):722–737, May 2007.

[28] T. Mudge. Power: a first-class architectural design constraint. *IEEE Computer*, 34(4):52 –58, April 2001.

[29] Rolf Neugebauer and Derek McAuley. Energy is just another resource: Energy accounting and energy pricing in the nemesis os. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 67–, 2001.

[30] Adel Noureddine, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. A preliminary study of the impact of software engineering on greenit. In *International Workshop onGreen and Sustainable Software (GREENS)*, pages 21–27, 2012.

[31] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. cpuidle: Do nothing, efficiently. In *Linux Symposium*, 2007.

[32] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, volume 2, pages 215–230. sn, 2006.

[33] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 29–42, 2012.

[34] Arun Raghavan, Laurel Emurian, Lei Shao, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M.K. Martin. Computational sprinting on a hardware/software testbed. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 155–166, March 2013.

[35] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P Pipe, Thomas F Wenisch, and Milo MK Martin. Computational sprinting. In *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2012.

[36] Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Apprehending joule thieves with cinder. In *Proceedings of the ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds*, pages 49–54, 2009.

[37] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: approximate data types for safe and general low-power computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2011.

[38] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power containers: an os facility for fine-grained power and energy management on multicore servers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 65–76, March 2013.

[39] Tajana Simunic, Luca Benini, and Giovanni De Micheli. Energy-efficient design of battery-powered embedded systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 9(1):15–28, 2001.

[40] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: a language and runtime system for perpetual systems. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 161–174, November 2007.

[41] Kayo Teramoto and H. Howie Huang. Pay as you go in the cloud: One watt at a time. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1546–1547, November 2012.

[42] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. Instruction level power analysis and optimization of software. In *Technologies for Wireless Computing*, pages 139–154. 1996.

[43] Amin Vahdat, Alvin Lebeck, and Carla Schlatter Ellis. Every joule is precious: the case for revisiting operating system design for energy efficiency. In *ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, pages 31–36, 2000.

[44] Narseo Vallina-Rodriguez and Jon Crowcroft. ErdOS: achieving energy savings in mobile OS. In *Proceedings of the International Workshop on MobiArch*, pages 37–42, 2011.

[45] Lizhe Wang, Gregor Von Laszewski, Jai Dayal, and Fugang Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs. In *Proceedings of the IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 368–377, 2010.

[46] M. Ware, K. Rajamani, M. Floyd, B. Brock, J.C. Rubio, F. Rawson, and J.B. Carter. Architecting for power management: The IBM POWER7 approach. In *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, pages 1–11, January 2010.

[47] Qiang Wu, Margaret Martonosi, Douglas W Clark, Vijay Janapa Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 271–282, 2005.

[48] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: managing energy as a first class operating system resource. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 123–132, 2002.

[49] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Currentcy: a unifying abstraction for expressing energy management policies. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2003.