E-Team: Practical Energy Accounting for Multi-Core Systems

Till Smejkal and Marcus Hähnel, *TU Dresden;* Thomas Ilsche, *Center for Information Services and High Performance Computing* (*ZIH*) *Technische Universität Dresden;* Michael Roitzsch, *TU Dresden;* Wolfgang E. Nagel, *Center for Information Services and High Performance Computing* (*ZIH*) *Technische Universität Dresden;* Hermann Härtig, *TU Dresden*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/smejkal

This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

E-Team: Practical Energy Accounting for Multi-Core Systems

Till Smejkal[1], Marcus Hähnel[1], Thomas Ilsche[2], Michael Roitzsch[1], Wolfgang E. Nagel[2], and Hermann Härtig[1]

[1]Operating Systems Group,TU Dresden
[2]Center for Information Services and High Performance Computing (ZIH), TU Dresden
firstname.lastname@tu-dresden.de

Abstract

Energy-based billing as well as energy-efficient software require accurate knowledge of energy consumption. Model-based energy accounting and external measurement hardware are the main methods to obtain energy data, but cost and the need for frequent recalibration have impeded their large-scale adoption. Running Average Power Limit (RAPL) by Intel® enables non-intrusive, off-the-shelf energy monitoring, but only on a per-socket level. To enable apportioning of energy to individual applications we present E-Team, a non-intrusive, scheduler-based, easy-to-use energy-accounting mechanism. By leveraging RAPL, our method can be used on any Intel system built after 2011 without the need for external infrastructure, application modification, or model calibration. E-Team allows starting and stopping measurements at arbitrary points in time while maintaining a low performance overhead. E-Team provides high accuracy, compared to external instrumentation, with an error of less than 3.5 %.

1 Introduction

Energy has become the major factor constraining the utility of today's systems. For mobile platforms, which rely heavily on battery life, energy efficiency is an important differentiator for applications and devices. Being more energy efficient is a competitive advantage. In datacenters, energy is nowadays dominating the operation costs, necessitating energy-based payment models [21].

Accurate accounting of energy is paramount to optimize energy consumption and to enable energy-based billing. Software developers rely on energy consumption statistics to find and fix energy bugs [31] and improve the energy efficiency of their algorithms [18]. But software development already requires developers' attention to non-functional properties, like responsiveness and security. To enable energy efficient systems developers need a measurement infrastructure that is *easy to use* and *cost-effective* to deploy.

As energy characteristics often only manifest during runtime of the deployed application, such infrastructure must be *non-intrusive* in production environments by not incurring any performance loss or energy penalty when not in use. Still, enabling on-the-fly measurement of individual applications or parts of the system should be as easy as executing a simple command.

1.1 State of the Art

External measurement hardware is accurate [23], but can only provide machine-level measurements. Inference based solutions [35, 25, 7] are more flexible, but require calibration. We strive for a solution combining the respective advantages.

Intel introduced the Running Average Power Limit (RAPL) technology in Sandy Bridge™ CPUs [33]. It provides a power limiting infrastructure that is automatically calibrated during startup and exposes energy measurements. While not providing per-application energy values, it removes the need for expensive, specialized external measurement hardware and comes with zero setup effort. We introduce RAPL in Section 2.

Simple inference-based models, using CPU time or retired instructions, fail to accurately capture energy consumption of complex workloads making energy apportioning infeasible. We illustrate this point by measuring a busy loop that does not touch any data, and FIRESTARTER [14], a CPU burner application designed for high power-usage. Figure 1 shows the result of the experiment. We establish a baseline by measuring the energy consumption of each app in isolation using RAPL. Then we run both programs at the same time, scheduled by Linux' CFS, measure the system-level end-to-end en-

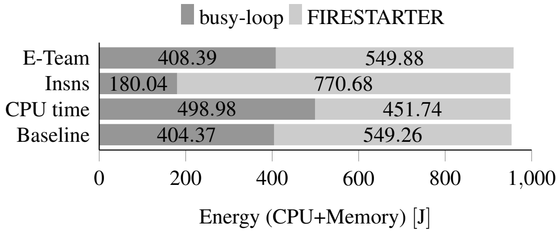


Figure 1: Energy attribution based on instructions retired (Insns) and CPU time, compared to E-Team

ergy consumption and apportion it based on CPU time and based on instructions retired. Both methods are incapable of correctly attributing energy consumption. We also give a short glimpse of the result of our solution, E-Team, which is able to accurately capture the energy consumption for both applications.

1.2 Contributions

We present the design and implementation of an operating system service for accurate and efficient measurement of per-application CPU energy use on multi-core systems using RAPL. Our key contributions are:

- A scheduler design to circumvent the limitations of RAPL using team scheduling (Section 3.1) and a Linux implementation (Section 4).
- A user-accessible interface to start and stop energy accounting of individual thread groups (Section 4.2).
- A scheduler integration of RAPL for short code paths (Section 2.3).
- An evaluation using standard benchmarks (NPB [1]) and real-world scenarios with multiple individually measured applications running in parallel (Section 5).
- A validation of our implementation's accuracy using a precise external measurement setup (Section 6).

2 RAPL for Energy Measurements

Starting with the Sandy Bridge generation, Intel CPUs provide the *Running Average Power Limit* technology (RAPL) [33]. As the name implies, RAPL is intended for power-limiting, but also provides energy counters. Due to the widespread availability and the fact that it requires no additional instrumentation, RAPL is used extensively for power and energy estimation [12, 16, 11, 39].

2.1 Basic RAPL Operation

RAPL provides energy measurements for four domains:

Package (PKG) the whole processor package,

Cores (PP0) aggregate of all cores in a package,

Graphics (PP1) the CPU-integrated graphics processing unit (not available on server platforms), and

Memory (DRAM) memory. Although officially only supported on server platforms [20], this domain is also available on desktop processors since Haswell.

The initial implementation of RAPL was based on a model using micro-architectural events to estimate energy consumption [8]. Hackenberg et al. [13] have revealed systematic errors in the RAPL energy counters, e.g. bias towards certain workloads and contradictory results when using Hyper-Threading. For Haswell generation processors, RAPL has been demonstrated to provide accurate measurements without systematic errors [15], hence the results presented in Section 5 and 6 were produced on Haswell desktop and server systems.

2.2 Limitations of Basic RAPL

Contrary to performance counters, RAPL counters are exposed exclusively through Model-Specific Registers (MSRs) that are only readable from kernel space. A number of methods exist in Linux to read the MSR in the kernel and make the values accessible to applications: Performance monitoring libraries such as PAPI [26] or LIKWID [38], and dedicated third-party drivers [24]. Since Linux 3.12, RAPL is usable as powercap driver. Since Linux 3.14 RAPL is accessible via the *perf* performance monitoring framework as system-wide performance-counter.

Another fundamental difference to conventional performance counters is that RAPL values are updated with an approximate frequency of 1 kHz [20] only, while performance counters are updated continuously. The update is not associated with a timestamp, preventing identification of stale values. The discrete updates make it difficult to measure code paths running shorter than or close to the counter's 1 ms update interval. The number of updates cannot be accurately determined: considering, for example, a piece of code running for 2.5 ms, it makes a large difference in terms of attributed energy whether there were two or three updates during that time.

This is especially visible in time-shared systems where switching between programs happens frequently. In these systems traditional performance counters are multiplexed by saving and restoring their values on every context switch. Such a technique cannot be trivially applied for RAPL because of the aforementioned update behavior. Counter values may be outdated at the point of context switching leading to significant measurement errors.

Similar to other measurement-based methods, mentioned in Section 1.1, the RAPL design cannot measure energy for individual cores or applications. Instead RAPL accounts the combined energy for all cores in a socket. This makes apportioning energy to an application executing in a multi-processor system with multiple,

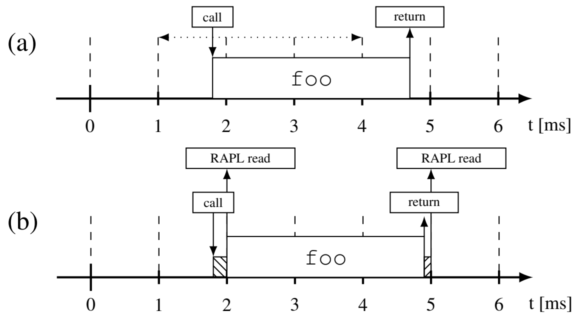


Figure 2: Synchronized measurement for short code

concurrently running applications non-trivial. In Section 3 we present the design of our scheduler-based measurement service. It ensures that, at any point in time, the cores of one socket are assigned exclusively to programs that should be measured together.

2.3 Measuring Short Code Paths

To address the problem of RAPL's fixed update intervals, we use a method from our previous work on measuring short code-paths [16]. When measuring a short code path, both the start and the end of the measurement may fall between the update points of the RAPL energy counter as illustrated in Figure 2 (a). The shorter the measured code path (here less than 3 ms) the higher the influence of measurement inaccuracies on the result. The measurement window, indicated by the dotted arrow, is offset against the code execution, delineated by *call* and *return*. The offset results in the inclusion of irrelevant code at the start and the omission of relevant code at the end. A solution to this problem is to synchronize the measurement time to counter updates. For the start of the measurement, this is achieved by repeatedly reading the `ENERGY_STATUS` register until it changes, indicating a RAPL update. Only then the measured code is executed.

Synchronizing the end of the function is not as simple. Just waiting for the next update will skew the measurement as the result would include the energy consumed while waiting. In our previous work, we propose to fill the time until the next update with a workload of fixed and known energy consumption [16]. Since polling the counter is needed to detect the update, using the polling loop as this defined workload elegantly solves the problem. Listing 1 shows pseudo-code for the algorithm executed when the function of interest terminates. The value of `ePerClock` is determined in a one-time calibration step performed by measuring the cost of repeatedly reading the RAPL counter over an interval of several updates (less than one second in total). Subtracting the known cost of the loop ensures that the returned value only contains energy consumed by measured code, thus effectively increasing the RAPL resolution and removing the constraint imposed by the update interval. Figure 2 (b) shows how this solution synchronizes the measurement with RAPL updates. The striped part before the measured code may have arbitrary energy consumption, the part at the end is the described polling loop. We call this mechanism *short-time RAPL.*

```
finish_measurement() {
    cycles = rdtscp();
    e_start = RAPLRead();
    while (RAPLRead() == e_start) {}
    cycles = rdtscp() - cycles;
    return RAPLRead() - ePerClock * cycles;
}
```

Listing 1: Algorithm for leaving measured code

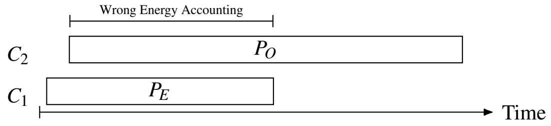


Figure 3: Miss-accounted energy in normal scheduling

2.4 Accounting Individual Processes

Figure 3 illustrates the challenge in accounting energy for individual processes using socket-local measurements. When using RAPL directly to measure process P_E running on core C_1, while an unrelated process P_O executes in parallel on core C_2, the final energy consumption will be incorrect. During the time marked in the graph the measurement will include the energy consumed by both programs. The example in Section 1.1 shows, that time-proportional accounting does not suffice to circumvent this problem as instruction types and access patterns are not taken into account. More sophisticated performance-counter-based models require extensive calibration leading to a significant deployment cost. Accordingly, we propose a different approach.

3 Energy Measurement as a Kernel Service

Our solution, E-Team, enables energy accounting on a per-application basis in multi-core systems through specialized scheduling. The main challenge of introducing a measurement infrastructure into a production system is to minimize the accompanying performance impact. The system should run as usual if there are no active measurements and hardware modifications should not be necessary. However, some overhead may be acceptable during troubleshooting or application development. Even in development scenarios isolating individual applications that are part of a complex system may be worthwhile to pinpoint energy bugs. Naïvely restricting the system to a single core or running the application of interest exclusively in the system solves the energy accounting problem but does not resemble production system behavior.

For the remainder of this paper, we refer to individually executing programs as *process*es. Processes may be comprised of many execution contexts called *thread*s. Each thread is scheduled as a *task* by the scheduler and has an assigned task structure in the kernel.

We introduce the concept of *teams*. A team is an arbitrary group of tasks whose energy is accounted together. Teams get exclusive access to their assigned CPU socket to prevent tasks of different teams from running on the same socket in parallel. This enables the use of the socket-wide energy measurements of RAPL to account the team's energy consumption. Tasks of a team are scheduled on the team's socket according to any scheduling scheme. Thereby energy characteristics caused by interaction between measured tasks are largely preserved and performance degradation is limited.

We call this approach *team scheduling*. For the remainder of this paper we refer to a team that is measured as a *measured team*. There exists exactly one team containing all tasks that should not be measured (the *non-measured team*). To enforce the team-scheduling policy, we added a new scheduler to Linux.

3.1 Team Scheduling

The design of the E-Team scheduler guarantees that no tasks belonging to different teams run on the same socket at the same time. We want to enforce the following properties in our energy measurement service:

Property 1 (Team Interactivity)**:** *Teams are interruptible to enable interaction between different teams and maintain system responsiveness.*

Property 2 (Task Interactivity)**:** *Tasks of a team share the team's cores fairly to enable task interaction and preserve the team's energy characteristics.*

Property 3 (Accuracy)**:** *The scheduler limits switches between teams to curtail measurement errors due to multiplexing and uses short-time RAPL as required.*

Property 4 (Non-invasiveness)**:** *In the absence of measurements, the system behaves like an unmodified system.*

Property 5 (Usability)**:** *Starting and stopping measurements is possible at any point in time, either initiated by the user or the program itself. Teams grow and shrink when tasks are created, destroyed, added, or removed.*

To account energy for individual processes we propose to assign sockets exclusively to teams (measured teams or non-measured) in a time-multiplexed fashion. This leads to the main invariant of our scheduler:

Invariant 1: *On any socket only tasks of the same team can run concurrently at any point in time.*

Property 2 requires cores to be time-multiplexed between tasks of a team. The *team scheduler* controls which tasks can run on which socket at any given time based on policy and team-membership. A *task scheduler* distributes the tasks assigned to a socket between its cores. The team scheduler makes no assumptions about the task scheduler policy and the policy can be set per team. This allows to use the Completely Fair Scheduler (CFS) as task scheduler.

The team scheduler manages a list of teams (*team runqueue*), whereas each team consists of tasks called *team members*. One item in the team runqueue is the non-measured team. Teams are activated and deactivated by the team scheduler only as a whole. To activate a team, the team scheduler first deactivates the currently running team. It then dequeues the new team, affinitizes its tasks to the socket and notifies the responsible task scheduler to reschedule. Deactivation of a team entails removing all its tasks from the socket and adding the team back to the team runqueue. This design enforces Invariant 1. Only tasks of one team are available to the socket-local task scheduler to be scheduled.

When all tasks in the system belong to the same team, team scheduling is reduced to a no-op. This is the case when no measurements are taken as the non-measured team then contains all tasks. Together with the possibility to run arbitrary scheduling schemes within the task schedulers, this enforces Property 4.

While not implemented by us the extension of the proposed scheme to multiple sockets is straightforward. A running team can occupy multiple sockets at the same time. The team scheduler may remove or add sockets to a team as necessary. The maximum number of simultaneously active teams is limited by the number of sockets. Having multiple teams active on different sockets does not affect accounting accuracy, as each socket has its own RAPL domains.

The architecture of E-Team can be thought of as core-local scheduling with socket-level coordination.

3.2 Fine-Grained Context Switching

Property 3 is the hardest to enforce. Although the team scheduling approach guarantees that energy is only accounted for measured tasks, we still need to ensure that processes which execute for short times due to blocking are accounted accurately. To minimize overhead we try to switch teams only every 100 ms or more. This results in an error of about 1 % due to the fixed RAPL counter update intervals. The team scheduling frequency is tun-

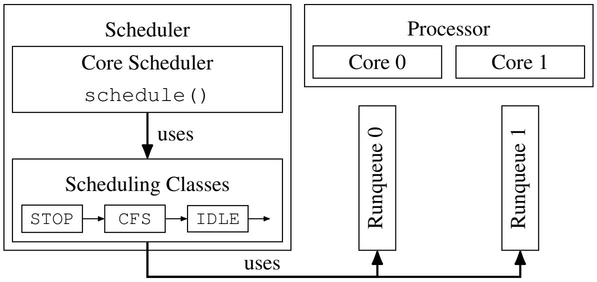


Figure 4: Linux scheduler architecture

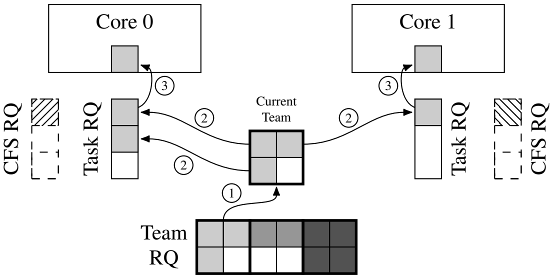


Figure 5: Scheduling a team for energy measurement

able, allowing the system operator to trade overhead against interactivity. More frequent team switches lead to higher system responsiveness at the expense of more overhead of the E-Team mechanism. Irrespective of the configured time-slice length, the measurement has to be stopped when all tasks within a measured team yield the CPU, which is regularly the case with I/O-bound workloads. To avoid measurement errors due to short executions, we employ the short-time RAPL technique introduced in Section 2.3 if less than 50 ms of the time slice are used. Otherwise we read the RAPL counters directly. This allows us to guarantee the accuracy property for all workloads, even interactive and I/O-heavy ones, while limiting the performance impact for compute-intensive workloads and execution phases.

4 Implementation

We implemented our energy measurement service E-Team as a scheduling class in the Linux kernel. The kernel patch and user tools are available on GitHub[1].

4.1 Scheduler Implementation

The general architecture of the Linux scheduling framework is illustrated in Figure 4. It consists of a core scheduler, which invokes the scheduling classes implementing the actual policies. Scheduling classes are sorted by priority. The highest priority is given to the *STOP* class, the lowest to the *IDLE* class. Schedulers maintain per-core runqueues, enabling them to make core-local scheduling decisions, which removes one of the bottlenecks in many-core systems. Usually, tasks in Linux are scheduled by the Completely Fair Scheduler (CFS). We prioritize the team scheduler above CFS.

The team scheduler must ensure that only tasks belonging to the same team are assigned to the same socket. The team scheduler maintains a list of teams (the team runqueue) where a team is a pointer to a list of the tasks that comprise the team. We illustrate the team scheduling process in Figure 5. As soon as the team scheduler decides — based on its team scheduling policy — to switch the team running on a socket it clears the core-local task runqueues . The team scheduler then picks the new team (step 1) and distributes its tasks to the cores of the socket (step 2) by enqueuing them in the task runqueues of the cores. CPU affinity is respected during this step. The core-local task schedulers are then triggered to reschedule the tasks in their task runqueue according to their scheduling policy (step 3). If there are not enough tasks in a team to occupy all cores of the assigned sockets, the idle task is scheduled on the remaining cores. This causes these cores to enter energy-saving states. When all the tasks in a team have terminated or the team's time slice is exhausted, the next team is scheduled.

Non-measured tasks are treated as an implicit team which is not managed by the team scheduler. In our implementation, they are not implemented as an actual team but as tasks kept in the core-local runqueues of CFS. These tasks are scheduled in between measured teams by yielding to CFS in order to enforce the interactivity property. How frequently E-Team yields to CFS depends on the number of non-measured tasks and the number of tasks in the measured teams, allowing fairness properties similar to CFS to be enforced.

Time-sliced round-robin with a base time-slice length of 100 ms was chosen as the team-scheduling policy. The base time-slice length is configurable. The actual length of the time slice depends on the number of ready tasks in the teams (i.e. the load). A team with more tasks waiting in its runqueues will get proportionally more time than a less-loaded team. This leads to fair multiplexing of CPU time between the team scheduler and the regular tasks in the system scheduled by CFS. We found a 100 ms base time-slice length to be a good compromise between overhead, accounting accuracy, and system-responsiveness. CFS chooses a similar base time-slice length for a system with CPU intensive load [6, Table 7-2]. Shortening the default time slice can improve responsiveness at the cost of higher overhead for frequent switching and more frequent use of the short-time RAPL mechanism. Please note that the default time-slice length is independent of the timer frequency. For all our experiments the timer

[1] https://github.com/TUD-OS

still ticked with a frequency of 1 kHz, thus invoking the scheduler every millisecond. The default time slice is a scheduler parameter that determines the *default* amount of time each process gets before it is rescheduled. The actual time may be less.

Tasks in the task runqueue are scheduled using time-sliced round-robin, but it would also be possible to use CFS as the task scheduling policy. Especially when measuring large teams containing many tasks with different priorities, CFS would better preserve the execution characteristics of the unmodified system.

When the E-Team scheduler does not schedule a measured team, it will yield to CFS, which then schedules the non-measured tasks as it normally would. This is an advantage of the implementation of the non-measured team using the normal CFS runqueues. Accordingly the system performs exactly as if it was unmodified whenever no tasks are in measured teams.

4.2 User-Level Tooling

Teams are formed by assigning a process to the E-Team scheduler. E-Team then automatically adds threads created by the process to the process's team. Although this simplified grouping was sufficient for our evaluation, it would also be possible to move individual threads to a team, disable the automatic addition of newly created threads, or combine several processes in one team.

The decision to use a specialized scheduler supports the usability property (see Section 3.1). Scheduler assignment is performed by starting the measured program through a tool such as `schedtool`. Alternatively our own tool `energy` can be used with the added benefit of outputting the energy consumption after program termination (like the Unix `time` utility does for time). Applications can start and stop measurement at arbitrary points in time by calling `sched_setscheduler` to move the process between the E-Team scheduler and CFS.

Applications can read their energy consumption from a file in their `procfs` subdirectory. Procfs provides runtime parameters and statistics of each process. We added two entries, `energystat` and `loopstat`, which provide access to the energy consumption and statistics about the scheduler's operation (number of time slices executed, short-time RAPL statistics, etc.), respectively. Listing 2 shows example output for the energy data.

Both files can be read during program execution to get regularly updated energy and statistics values. The content of the files will not change when the process is not measured and retains the values from when the process left the E-Team scheduling class. The files will retain their final values when the process stops being measured by leaving the E-Team scheduling class.

```
root@measure$ cat /proc/100/energystat
package (uJ)        : 1052792342
dram (uJ)           : 54277983
core (uJ)           : 842365434
gpu (uJ)            : 89234
updates             : 303
avg_loop_time (us)  : 180
```

Listing 2: Example data provided by E-Team

5 Evaluation

To deliver on our promise of accurate energy accounting, we evaluate our system by first establishing a baseline using an unmodified system and then analyzing the energy and time overhead in three scenarios. We start by measuring a single application running alone on a Linux system. We then add background load and execute two applications in parallel, measuring them individually. Finally, we investigate the influence of short scheduling intervals and the effects of short-time RAPL.

Measurements were performed on a single-socket quad-core Intel® Haswell Core™ i7-4770 machine with 3.4 GHz nominal frequency and 2×4 GiB of DDR3 CL9 RAM clocked at 1333 MHz. We disabled Hyper-Threading and Turbo Boost, to make the individual measurements more deterministic and maintain comparability between single-application and multi-application runs. These options could otherwise lead to different behavior based on thread assignment and the decisions of Turbo Boost. We used a Linux 4.2.3 kernel in our experiments. Although we measured energy for all available RAPL domains, we only present PKG energy for brevity. The other domains showed comparable results.

5.1 Baseline and Overhead

Our first measurements establish the baseline for the rest of our evaluation. Baseline measurements were performed on a Linux system stripped down to the minimum necessary to run the benchmarks: We ran the system from an initrd with no system services interfering with execution. We believe the measured energy to conform to the energy consumed by the benchmarks. We use the NAS Parallel Benchmarks (NPB) [1], version 3.1, as benchmarks. Presented data is averaged over 20 consecutive runs. Error bars are not given in graphs if the standard deviation is below 1 %. Figure 6 shows the end-to-end measurement of the benchmarks for wall-clock time, cpu time and package energy (measured by the PKG counter). The benchmarks were scheduled using CFS. No parts of our kernel modification were active during the runs. Time was determined using the `time` command, while energy was measured by reading the RAPL MSR at the start and end of the benchmark. We measured each benchmark running with one to four threads.

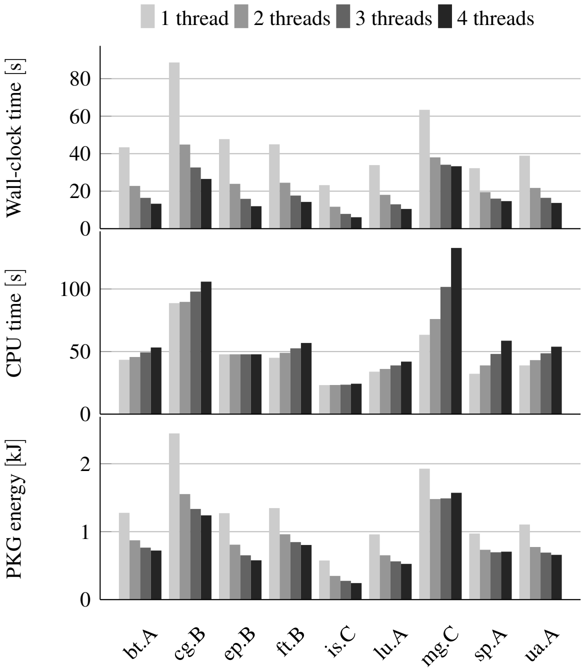


Figure 6: Baselines for wall-clock time, CPU time and PKG energy for different NPB kernels

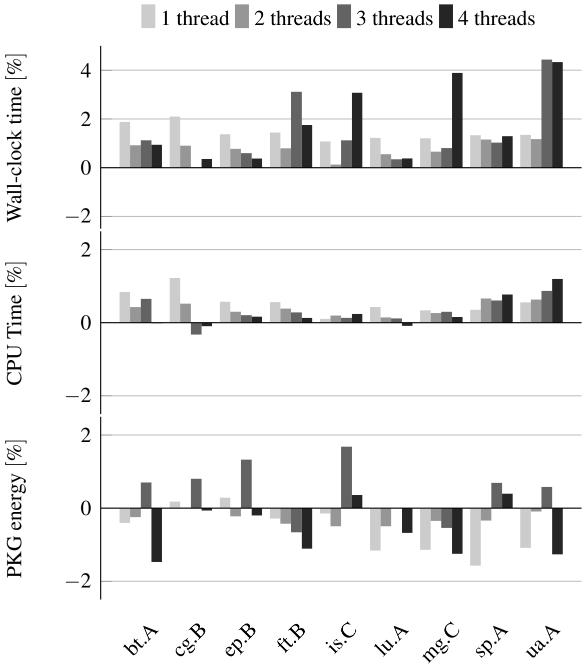


Figure 7: Wall-clock time, CPU time, and PKG energy measured using team scheduling compared to baseline.

We do not include the DC benchmark in our measurements because it mixes computation with extensive I/O. We found that its CPU time deviates significantly (>10 %) from wall-clock time when run as single application with one thread. The effect increases with the number of threads. This makes an end-to-end measurement meaningless as too much of the time is spent outside the benchmark. This is one of the cases that cannot be measured reliably without E-Team. We evaluate similar cases in Section 5.4 and will show a detailed discussion of DC in Section 6, when comparing against external measurements. For the other benchmarks, wall-clock time matched CPU time for the single-core case, resulting in a usable end-to-end baseline for energy.

Next, we repeated the baseline measurement using E-Team. This measurement and all the following in this section were performed on a normal Arch Linux system that was not stripped down. Ideally, the results obtained from E-Team would show the same wall-clock time and CPU time as the baseline. We also expected slightly lower energy consumption than the baseline, since E-Team does not account energy that is consumed by kernel tasks or by other processes in the system. Figure 7 shows the results of our measurements relative to the baseline. Team scheduling increases the wall-clock time of each benchmark. The more threads the program has, the longer it executes compared to the baseline, since background load in the system, even if single-threaded, blocks the whole measured program from running on the CPU. This is expected and the worst-case overhead is approximately 4 %. As we had hoped, CPU time did not increase significantly, which shows that the performance impact of our scheduler is negligible at less than 1 % in most cases. As long as there are enough tasks in all the teams, total performance of the system will not suffer. For package energy, our measurements are in the expected range with a difference relative to the baseline of less than 2 %. For most benchmarks we even measure less consumption due to the exclusion of unrelated work performed by the system. The measurements prove that our method combines low overhead with high precision.

5.2 Surveying Individual Groups of Tasks

After demonstrating that E-Team performs as good as the end-to-end measurements, we will show that our measurements stay accurate even in the presence of other tasks that are scheduled by the system. We introduce background load by running a single-threaded busy loop concurrently to the NPB suite. An empty busy loop does not touch any data and thus avoids any cache interference that could lead to changes in energy consumption.

Figure 8 shows the results of this experiment. We omit wall-clock time, as it is not a useful metric to compare against in this case. Wall-clock time will increase compared to the baseline in any case due to the intro-

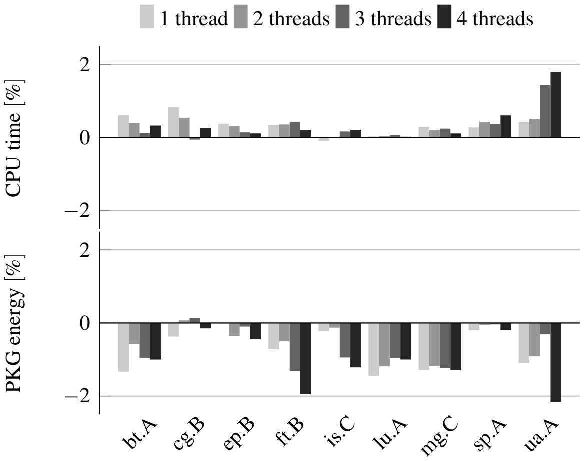


Figure 8: Per-application CPU time and PKG energy (relative to the baseline) as determined by E-Team with a background application running in parallel.

duced background load. For CPU time, we see an overhead of at most 2 %, while energy measurements are slightly below the baseline. We suspect the encountered energy reduction to be an artifact of precision limits of the RAPL counters. The busy-loop consumes significantly less energy than the benchmarks and we speculate that internal RAPL state influenced by this low-power activity bleeds into the results for the much more energy-consuming benchmarks. To test this hypothesis, we replaced the busy-loop with FIRESTARTER, which consumes more energy than the benchmarks. In this experiment, energy consumption increased relative to the baseline (e.g., 1.6 % for *ft.B*). This result indicates that inaccuracies within RAPL caused the measurement errors we observed. RAPL counters for DRAM proved less susceptible to this effect.

5.3 Multiple Measurements

One feature of our scheduler is that we can extract and measure a single application out of a number of applications running in parallel on the system. To demonstrate this feature we executed all application-pairs of the NPB suite (except for DC due to the lack of a meaningful baseline), measuring only one application of the pair. The results can be seen in Figure 9. We used scheduling slices of 100 ms to limit interference between the benchmarks. We ran this benchmark for one to four threads and compared the results against baseline.

As a guide to read Figure 9, consider the row with *ft.B* in the rightmost pane showing four threads in Figure 9b: Selecting the column *is.C* shows that the measured energy consumption of *ft.B*, when running concurrently with *is.C*, is 4 % below the baseline. No statement is made about *is.C* in this cell. Our worst-case error is 6 % for the DRAM energy (not shown) when running *ua.A* concurrently with itself. This may be attributed to either measurement errors introduced by our scheduler, errors in the RAPL model (i.e. incorrect energy values), or interference between the programs, despite the long scheduling interval. We will discuss the cause of this divergence in Section 6. Even a 6 % error is still on par with model-based estimation techniques [32, 37, 5].

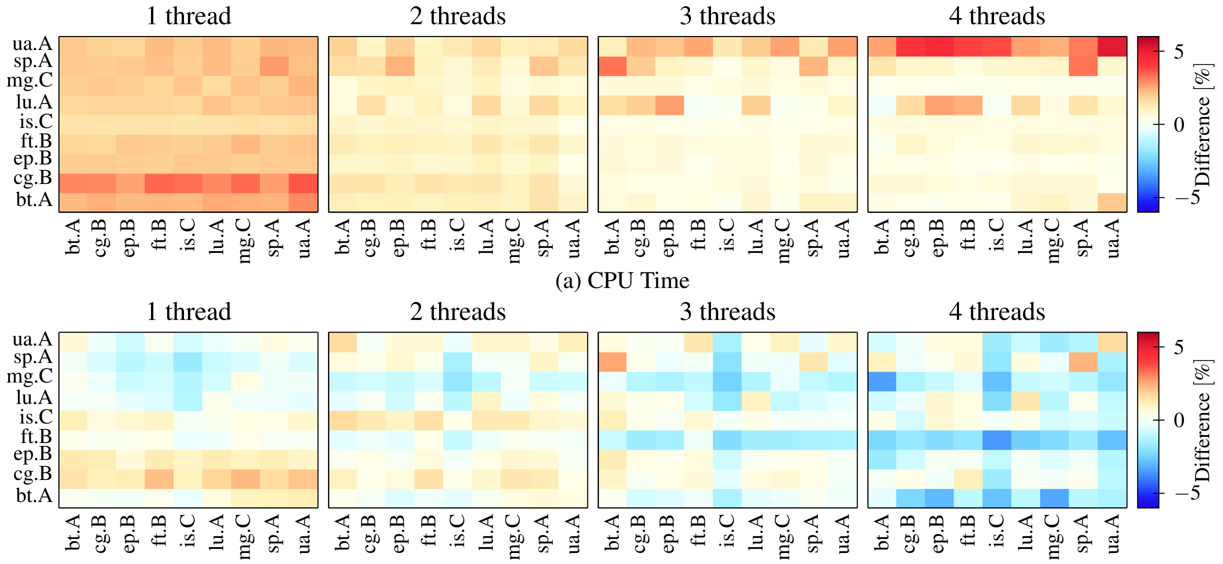
5.4 Short Scheduling Intervals

Particularly challenging for E-Team are applications that execute in short bursts, blocking in-between execution phases. Interactive GUI or multimedia applications as well as I/O-bound applications are examples that exhibit such behavior. They require rescheduling more often than our default time slice of 100 ms by yielding the CPU. Every time all the threads in the currently running team yield the CPU, we must switch to another team. If the last switch was not at least 50 ms ago, we need to perform short-time RAPL (refer to Section 2.3), to avoid inaccuracies introduced by the time-discrete updates of the RAPL counters. To evaluate the benefits of short-time RAPL for scheduling, we implemented a synthetic, interactive load that executes a busy loop for 5.4 ms, subsequently blocks for 1 s and then repeats the procedure 50 times. We compare short-time RAPL and naïve, update-oblivious multiplexing of the counter. As baseline we measure the busy loop that occupies the CPU as long as our synthetic workload (270 ms), but runs uninterrupted. Figure 10 shows the results. The energy measured by E-Team matches the baseline. When using naïve, update-oblivious multiplexing our measurements exhibit an error of up to 10 %. In contrast, short-time measurements only exhibit an error of 0.2 %. We conclude, that E-Team can reliably measure interactive and I/O-intensive tasks that yield the CPU frequently.

5.5 Practical Scenarios

Virtual machines We used `qemu-kvm` to run two VMs with Debian Jessie 8.4 64-bit, each given one core and 2 GiB of RAM. One VM was serving files over HTTP, the other was a malicious VM wasting CPU cycles by executing FIRESTARTER. We ran both VMs in parallel on Arch Linux using E-Team. Each VM received 300 s CPU time. The fileserver used 1034.1 J while the malicious VM used 7013.8 J. Based on this information a data-center operator could use appropriate billing or reduce the CPU time allocated to the malicious VM.

Single-Core Sampling We show the effectiveness of sampling to reduce overhead for single-threaded work-



(a) CPU Time

(b) Package Energy

Figure 9: Benchmarks in the rows are measured while running concurrently with the benchmarks in the columns. Shown is the relative difference to the baseline. We repeated this experiment for thread counts from one to four.

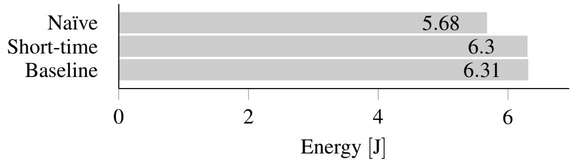


Figure 10: Short-time RAPL and update-oblivious measurements compared to baseline

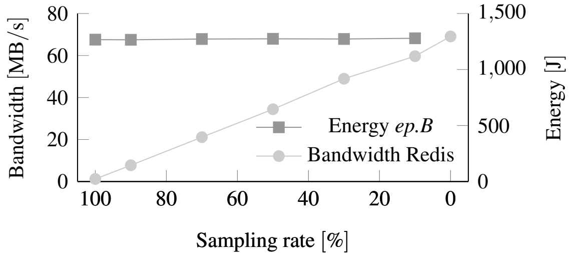


Figure 11: Redis throughput vs. *ep.B* measurement accuracy at various sampling rates

loads. Our setup consists of a single-threaded measured instance of *ep.B* running in parallel with unmeasured Redis, a popular in-memory database. To reduce the performance impact of isolating *ep.B* we employ random sampling. Figure 11 shows the throughput of memtier_benchmark at default configuration. At 10 % sampling we achive 86.4 % of the baseline performance of Redis, while maintaining 99 % measurement accuracy for *ep.B*. Like DC from NPB, Redis' I/O-intensive nature complicates determining an energy baseline. We thus omit its energy values.

6 External Validation

When running multiple teams in parallel, as done in Section 5.3, we do not know the cause of any aberrations from the baseline. Causes may be interference between threads, RAPL inaccuracies, or accounting errors in E-Team. To rule out the latter, we verify E-Team results using a secondary, external measurement infrastructure.

6.1 Measurement Setup

To verify the accuracy of our results, we use a sophisticated high-resolution power measurement infrastructure, which has been thoroughly verified [19]. It has been adapted to a Haswell-EP with two-socket Xeon E5-2690 v3 and a total of 256 GiB DDR4-2133 ECC RAM that we use as evaluation platform in this section.

We compare the results of RAPL against direct current (DC) measurements at inputs of each socket's voltage regulators. Both sockets are measured at a sampling rate of 500 kSa/s, to track power consumption between scheduling events. Data obtained from the external measurement infrastructure correspond to the sum of PKG and DRAM consumption according to RAPL. Because we measure at the input of the voltage regulators, the external measurements cover some components on the mainboard that are not measured by RAPL. Therefore RAPL reports less power consumption than the external measurements, even if both are perfectly accurate in their own power domain.

The verification is focused on identifying potential systematic inaccuracies introduced by our novel tech-

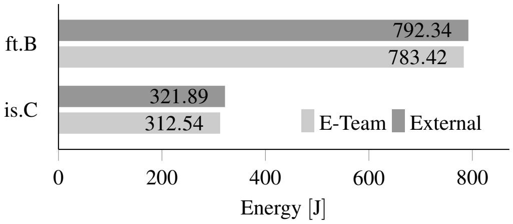


Figure 12: *ft.B* and *is.C* PKG energy using E-Team and external measurement with 12 threads per application

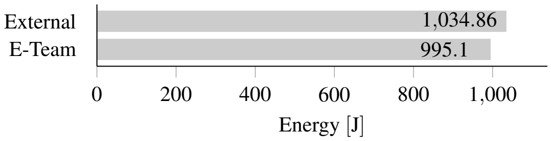


Figure 13: *dc.W* measured with E-Team and externally

nique. To compare the measured reference against the power domain of RAPL, we apply a model to map between the two. The model is trained on measurements of different workload kernels executed at various thread-counts and configurations as described in [4]. Training is performed on a non-modified Linux system using continuous RAPL and reference measurements. Linear regression provides the final slopes and intercepts separately for each socket with $R^2 > 0.999$.

Since the external measurement traces not only contain the power usage of the measured program but also of other tasks executed in parallel, a post-processing step was necessary to identify the regions in the traces during which the program of interest actually executed. For this purpose, we used an additional trace, generated by the E-Team scheduler, which indicates when each program was scheduled on the processor. We had to synchronize the traces, because they have timestamps from different clocks. We generated a special energy pattern before and after every measurement to correlate the measurement and scheduler traces.

6.2 Results

To validate our measurements from Section 5, we execute selected benchmarks on the instrumented hardware. We present the case of *ft.B* running together with *is.C*, which we already used in Section 5.3, as they exhibit significantly different power usage of 110 W and 80 W per socket, respectively. The results in Figure 12 show that our measurement is very accurate with an error of 1.1 % for *ft.B* and 2.9 % for *is.C*. The 12-core configuration used for the figure represents the worst case for this benchmark. The error decreased with fewer threads. We also examined the DC benchmark, which we were not able to evaluate in Section 5. We measured *dc.W* running with two threads and compared the external measurement to the E-Team result. Figure 13 shows that even for this I/O-intensive benchmark E-Team's error is only 3.5 %. Over 20 consecutive runs we observed a standard deviation well below 1 % in all cases.

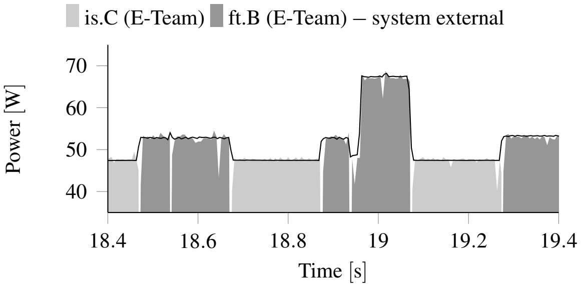


Figure 14: Power characteristics over time

Figure 14 shows that our E-Team implementation accurately tracks energy consumption over time. We ran *is.C* and *ft.B* in parallel, each in its own measured team and read their respective `procfs` entry repeatedly. We used a time slice of 200 ms. The characteristics of the external measurement match those of the internal one. The dips visible in the power consumption reported by E-Team (e.g. at 18.95 s) are caused by switches between teams or scheduling of the non-measured team. Tasks in the non-measured team yielded after very short time leading to short interruptions of the measured teams.

7 Limitations

E-Team provides accurate energy accounting for arbitrary groups of threads using socket-wide energy measurements. But this feature comes at a cost: a team always needs exclusive access to the socket. Accordingly, resources remain unused if teams cannot spread across all cores of the socket. The pathological example for this is a team that consists of a single thread. However, E-Team allows on-the-fly starting and stopping of measurements and thereby supports random sampling. This creates a trade-off space between measurement accuracy and performance overhead. In cases where the performance overhead of E-Team is prohibitive, limiting the measurement duration to short sampling intervals allows for acceptable performance at a slight loss of accuracy.

For I/O-bound workloads short-time RAPL is required more often, incurring additional overhead. In the worst case this translates to 1 ms overhead per team switch. Our experiments with *dc.W* showed only a performance degradation of 4.7 % in the worst case. We further examined a worst-case scenario for I/O-bound workloads by running `grep` recursively over the Linux source tree. We identified the extreme case showing 150 % overhead (155.32 s vs. 62.30 s) when flushing the buffer cache before the run. We also measured Redis running memtier_benchmark and achived 30 % to 80 % of native performance for data sizes of 32 B to 128 kB despite its I/O-intensive nature. To measure such scenarios, we advice the use of random sampling.

8 Related Work

As energy efficiency is a cross-cutting concern, it has been approached from both the hardware and software side. On the hardware side, external measurement methods, such as those proposed by Hönig et al. [18], have improved significantly in sampling speed and accuracy over existing solutions, such as the frequently used Watts-Up power meter [9]. External measurements as data sources integrate well with our method, but introduce the need for additional hardware. Intel's RAPL addresses this problem by providing self-calibrating models [33]. Hackenberg et al. have shown that RAPL produces accurate energy estimates in recent versions [15] and compare various measurement methods [13].

Below the application layer, system architects construct runtimes and scheduling frameworks to model [30], account [29], and control [34] platform energy use. Several methods using performance counter based power models [22, 9, 36, 3, 2] exist. They exhibit relative errors in the range of 5 % to 10 % but can, contrary to RAPL, include other components such as disks. However, models require calibration, which has to be performed for each individual CPU. McCullough et al. found variations between individual CPUs of the exact same type to be too large to calibrate based on CPU model and have shown that linear CPU energy models are intrinsically limited in their accuracy [27].

There are various approaches using performance-counter-based models to apportion energy to VMs or applications. Shen et al. investigate Power Containers, which use model-based apportioning of energy to applications [36]. They use external recalibration during runtime, thus relying on additional hardware. Their methods exhibit relative errors of up to 11 % on Sandy Bridge CPUs. Bertran et al. account energy for VMs using a model-based approach and report 5 % relative error [3].

For high performance computing (HPC) systems, Georgio et al. have shown a SLURM-based job management system, which allows accounting of energy to jobs [12]. Their approach is limited to account energy on a per-node level. While suitable for typical HPC systems, it does not cover cloud or data-center scenarios with multiple simultaneous users per machine.

To schedule groups of tasks Ousterhout introduced co-scheduling [28] and an Feitelson et al. presented gang-scheduling [10]. Our method builds on these approaches.

9 Conclusion & Future Work

We presented the design and implementation of E-Team, a facility that enables accurate measurement of energy consumption for individual threads or groups of threads in a system. We isolate groups of interest using team scheduling. This enables us to use a system-wide measurement method, such as Intel's RAPL, while still being able to apportion energy consumption per thread or group of threads. To address the discrete nature of the RAPL readings, we employ short-time measurements to accommodate for applications that are interactive or yield the CPU often. We are able to isolate arbitrary parts of a system and apportion their energy with an error of at most 3.5 % compared to external measurements. Our methods provide greater accuracy than many existing model-based approaches and our validation shows that E-Team can apportion energy faithfully. To the best of our knowledge, our implementation is the first to allow practical, high-precision, per-application energy attribution in a multi-core system without relying on manual calibration or external measurement equipment.

Our implementation is applicable to a wide range of devices. E-Team does not rely on RAPL but can use other energy measurement techniques such as sensors available on mobile platforms [17] or hand-held devices.

Some ideas of our design are not yet implemented and are left for future work. We did not implement simultaneous execution of different teams on different sockets. The challenge in accounting energy on multiple sockets concurrently is that applications running on one socket can cause energy usage in another socket. Remote memory access is one example for such behavior. We leave the implementation of a cgroup-like interface to future work as well. Such an interface could prove useful to combine threads of multiple applications into one measured team. While we implemented random sampling, a detailed discussion of the performance and accuracy implications is left for future work, due to space constraints.

In summary, our work represents a significant step forward for data-center energy accounting, energy-based billing, and energy profiling of applications in production systems. E-Team provides a cheap, accurate, and easy-to-use solution for on-the-fly energy accounting.

Acknowledgements

This work is supported by the German Research Foundation (DFG) within the CRC 912 - HAEC. The authors would like to thank Mario Bielert for his work on the verification of the external measurement system.

References

[1] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications 5*, 3 (1991), 63–73.

[2] Basmadjian, R., and De Meer, H. Evaluating and modeling power consumption of multi-core processors. In *Future Energy Systems: Where Energy, Computing and Communication Meet (e-Energy), 2012 Third International Conference on* (2012), IEEE, pp. 1–10.

[3] Bertran, R., Becerra, Y., Carrera, D., Beltran, V., Gonzalez, M., Martorell, X., Torres, J., and Ayguade, E. Accurate energy accounting for shared virtualized environments using PMC-based power modeling techniques. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on* (2010), IEEE, pp. 1–8.

[4] Bielert, M. Evaluating power estimation techniques: A methodological approach. Master's thesis, Technische Universität Dresden, 2016.

[5] Bose, P., Martonosi, M., and Brooks, D. Modeling and analyzing CPU power and performance: Metrics, methods, and abstractions. *Tutorial, ACM SIGMETRICS* (2001).

[6] Bovet, D., and Cesati, M. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[7] Colmant, M., Kurpicz, M., Felber, P., Huertas, L., Rouvoy, R., and Sobe, A. Process-level power estimation in VM-based systems. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 14.

[8] David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. RAPL: Memory power estimation and capping. In *Proceedings of the 2010 ACM/IEEE International Symposium on Low-Power Electronics and Design* (2010), ISLPED, IEEE, pp. 189–194.

[9] Do, T., Rawshdeh, S., and Shi, W. ptop: A process-level power profiling tool. In *Proceedings of the 2nd workshop on power aware computing and systems (HotPowerâĂŹ09)* (2009).

[10] Feitelson, D. G., and Rudolph, L. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing 16*, 4 (1992), 306–318.

[11] Gautham, A., Korgaonkar, K., Slpsk, P., Balachandran, S., and Veezhinathan, K. The implications of shared data synchronization techniques on multi-core energy efficiency. In *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems* (2012).

[12] Georgiou, Y., Cadeau, T., Glesser, D., Auble, D., Jette, M., and Hautreux, M. Energy accounting and control with SLURM resource and job management system. In *Distributed Computing and Networking*. Springer, 2014, pp. 96–118.

[13] Hackenberg, D., Ilsche, T., SchÃűne, R., Molka, D., Schmidt, M., and Nagel, W. E. Power measurement techniques on standard compute nodes: A quantitative comparison. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on* (April 2013), pp. 194–204.

[14] Hackenberg, D., Oldenburg, R., Molka, D., and Schone, R. Introducing FIRESTARTER: A processor stress test utility. In *Green Computing Conference (IGCC), 2013 International* (2013), IEEE, pp. 1–9.

[15] Hackenberg, D., Schone, R., Ilsche, T., Molka, D., Schuchart, J., and Geyer, R. An energy efficiency feature survey of the Intel Haswell processor. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International* (2015), IEEE, pp. 896–904.

[16] Hähnel, M., Döbel, B., Völp, M., and Härtig, H. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Perform. Eval. Rev. 40*, 3 (Jan. 2012), 13–17.

[17] Hähnel, M., and Härtig, H. Heterogeneity by the numbers: A study of the ODROID XU+E big.LITTLE platform. In *6th Workshop on Power-Aware Computing and Systems (HotPower 14)* (2014).

[18] Hönig, T., Janker, H., Eibel, C., Mihelic, O., and Kapitza, R. Proactive energy-aware programming with PEEK. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)* (2014).

[19] Ilsche, T., Hackenberg, D., Graul, S., Schuchart, J., and Schöne, R. Power measurements for compute nodes: Improving sampling rates, granularity and accuracy. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)* (Dec. 2015), The sixth international green and sustainable computing conference, pp. 1–8.

[20] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A, 3B, and 3C: System Programming Guide*, Sep 2014. Section 14.9.

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

[22] Kansal, A., Zhao, F., Liu, J., Kothari, N., and Bhattacharya, A. A. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 39–50.

[23] Konstantakos, V., Chatzigeorgiou, A., Nikolaidis, S., and Laopoulos, T. Energy consumption estimation in embedded systems. *Instrumentation and Measurement, IEEE Transactions on 57*, 4 (2008), 797–804.

[24] Krapl: Intel RAPL driver exposing the RAPL interface in sysfs. `https://github.com/TUD-OS/krapl`.

[25] Li, T., and John, L. K. Run-time modeling and estimation of operating system power consumption. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2003), SIGMETRICS '03, ACM, pp. 160–171.

[26] McCraw, H., Ralph, J., Danalis, A., and Dongarra, J. Power monitoring with PAPI for extreme scale architectures and dataflow-based programming models. In *Proceedsings of the 2014 IEEE International Conference on Cluster Computing* (2014), CLUSTER, IEEE, pp. 385–391.

[27] McCullough, J. C., Agarwal, Y., Chandrashekar, J., Kuppuswamy, S., Snoeren, A. C., and Gupta, R. K. Evaluating the effectiveness of model-based power characterization. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 12–12.

[28] Ousterhout, J. K. Scheduling techniques for concurrent systems. In *ICDCS* (1982), vol. 82, pp. 22–30.

[29] 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* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 29–42.

[30] PATHAK, A., HU, Y. C., ZHANG, M., BAHL, P., AND WANG, Y.-M. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the 6th ACM European Conference on Computer Systems* (2011), EuroSys, ACM, pp. 153–168.

[31] 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* (2012), MobiSys, ACM, pp. 267–280.

[32] RIVOIRE, S., RANGANATHAN, P., AND KOZYRAKIS, C. A comparison of high-level full-system power models. *HotPower 8* (2008), 3–3.

[33] ROTEM, E., NAVEH, A., ANANTHAKRISHNAN, A., RAJWAN, D., AND WEISSMANN, E. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro 32*, 2 (2012), 20–27.

[34] 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 6th ACM European Conference on Computer Systems* (2011), EuroSys, ACM, pp. 139–152.

[35] RYFFEL, S. LEA2P – The linux energy attribution and accounting platform. *Master's thesis, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland* (2009).

[36] 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.

[37] SNOWDON, D. C., PETTERS, S. M., AND HEISER, G. Accurate on-line prediction of processor and memory energy usage under voltage scaling. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software* (New York, NY, USA, 2007), EMSOFT '07, ACM, pp. 84–93.

[38] TREIBIG, J., HAGER, G., AND WELLEIN, G. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 39th International Conference on Parallel Processing Workshops* (2010), ICPPW, IEEE, pp. 207–216.

[39] WANG, W., CAVAZOS, J., AND PORTERFIELD, A. Energy autotuning using the polyhedral approach. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques, S. Rajopadhye and S. Verdoolaege, Eds., Vienna, Austria* (2014).