UCLA
Technical Reports

Title
Accurate Energy Attribution and Accounting for Multi-core Systems

Permalink
https://escholarship.org/uc/item/81s2s0t2

Authors
Ryffel, Sebi
Stathopoulos, Thanos
McIntire, Dustin
et al.

Publication Date
2009-01-09

eScholarship.org Powered by the California Digital Library
University of California

Accurate Energy Attribution and Accounting for Multi-core Systems

Sebi Ryffel[†], Thanos Stathopoulos[‡], Dustin McIntire[‡], William J. Kaiser[‡], and Lothar Thiele[†]

[†]Swiss Federal Institute of Technology (ETH) Zurich, Switzerland
sryffel@ee.ethz.ch, thiele@tik.ee.ethz.ch
[‡]UCLA, Department of Electrical Engineering
thanos@cs.ucla.edu, dustin@seas.ucla.edu, kaiser@ee.ucla.edu

Abstract

This paper presents a novel energy attribution and accounting architecture for multi-core systems that can provide accurate, per-process energy information of individual hardware components. We introduce a hardware-assisted direct energy measurement system that integrates seamlessly with the host platform and provides detailed energy information of multiple hardware elements at millisecond-scale time resolution. We also introduce a performance counter based behavioral model that provides indirect information on the proportional energy consumption of concurrently executing processes in the system. We fuse the direct and indirect measurement information into a low-overhead kernel-based energy apportion and accounting software system that provides unprecedented visibility of per-process CPU and RAM energy consumption information on multi-core systems. Through experimentation we show that our energy apportioning system achieves an accuracy of at least 96% while impacting CPU performance by less than 0.6%.

1 Introduction

The ever-increasing energy requirements of modern computing devices, from mobile and embedded systems to large data centers, present significant research and technical challenges. In data centers in particular, rising energy costs have resulted in hardware replacement cycles of two years or less, as it is more cost effective to acquire newer, more energy efficient systems than maintaining older ones [23]. According to recent studies [16], electricity use associated with servers has doubled between 2000 and 2005 and is expected to rise by up to 76% by 2010. It is therefore paramount that computing platforms across all application domains consider energy efficiency as a primary design objective.

In addition to advances in hardware and low-power CMOS technology, a critical step in achieving higher energy efficiency is the development of a deep understanding of the *runtime energy consumption of individual system entities*, including hardware and software components. By obtaining detailed, runtime information about energy consumption of system entities and by determining the energy consumption contribution of individual entities further operating system and application energy optimizations can be achieved. Detailed runtime energy profiling of applications can be used to identify suboptimal behaviors and thereby improve the energy usage. Moreover, runtime application energy information can be used for auditing and accounting purposes. For instance, a service provider could potentially charge clients by energy usage, in addition to computational resource usage and network bandwidth usage. Therefore, the goal is to develop an *energy measurement and accounting* system that can accurately determine the contribution of *individual processes* to the energy consumption of *individual hardware components* such as CPU or main memory.

This paper introduces an energy attribution and accounting architecture for *multi-core systems*. Using a combination of detailed, hardware-assisted energy measurements and indirect energy measurement models, our system provides the first (to the best of our knowledge) runtime, low-overhead, integrated energy monitoring of *individual processes* executing concurrently on a multi-core platform. Our work focuses on the computational subsystem, including CPUs and main memory that together can account for 30-50% of a server's total energy consumption [12, 23]. In addition to its paramount importance in the overall server functionality, the computational subsystem indirectly influences the power required for cooling, planar, and other components that make up the remaining energy consumption of a server.

Several different mechanisms exist that attempt to determine a system's energy consumption. Options include ACPI battery state [1] for mobile systems, external measurements [8] or energy estimation [5, 20]. In contrast to prior work, we introduce the *RunTime Direct Energy*

Measurement System (RTDEMS), a high-resolution energy measurement system that provides energy values for *individual* hardware components such as CPU, SDRAM, motherboard, video card, hard drive and so on. RTDEMS is *integrated* with the host platform, thereby allowing the host platform's operating system immediate and direct access to energy data. We argue that direct energy measurements such as those provided by RTDEMS are *necessary*, albeit *insufficient* to determine per-process energy attribution. We consequently introduce an *indirect energy measurement model* that is based on performance counters to determine the proportional contribution of individual processes to the total energy consumption of a hardware component. By asynchronously combining data from RTDEMS and the indirect energy measurement model into a kernel-space software system, we demonstrate through experimentation that we can attribute energy consumption to concurrently executing processes with at least 96% accuracy, while inducing less than 0.6% of CPU overhead.

The primary contribution of this paper is the introduction and experimental verification of a novel per-process energy attribution and accounting architecture for multi-core platforms. Additional key contributions include:

- The introduction of the runtime direct energy measurement system that provides accurate high-resolution energy information on a per-hardware component basis with negligible overhead (Section 2).
- An experimental analysis of the energy apportioning problem in multi-core systems, using RTDEMS-obtained data (Section 3).
- A performance-counter based indirect energy measurement model approach as a proposed solution to the energy apportioning problem that includes experimental data for several applications (Section 4).
- A low-overhead kernel-based software system that combines data from RTDEMS and the performance counter behavioral model to provide per-process energy information for arbitrary processes (Section 5).

We have used our system to attribute energy consumption to several applications on a multi-core platform and present our results in Section 6. We present related work in Section 7 and conclude the paper in Section 8.

2 The Runtime Direct Energy Measurement System

In this Section, we provide an overview of direct energy measuring techniques and also describe our detailed real-time direct energy measurement system. Using empirical information and experimental results, we argue that a detailed direct energy measurement system is *necessary* in order to attribute proportional energy consumption to hardware and software activities.

2.1 Direct energy measurement system overview

The *direct* energy measurement system is a critical element of our overall architecture, as it provides the necessary information regarding energy consumption. As our goal is to attribute energy consumption to individual hardware and software entities at *runtime*, the direct energy measurement system needs to satisfy the following requirements:

- *Resolution*. To resolve energy consumption of individual entities, the measurement system must provide high resolution information in both the spatial (for hardware) and temporal (for software) domains.
- *Cost*. The measurement system needs to operate with the lowest possible overhead—in terms of energy and resource consumption—as it is intended to be used in production systems.
- *Integration*. The measurement system needs to be integrated with the system-under-measurement so as to provide the necessary information in the fastest and most resource-efficient way possible.

Traditional energy measurement solutions in mobile, desktop as well as server class systems rely on external measurements, such as oscilloscope sampling or other data acquisition systems [3, 2, 10, 11, 8]. For battery powered systems internal devices such as commercial "fuel gauge" or simpler voltage monitoring solutions are common [6, 21, 18]. However, none of those devices in either category satisfies all three aforementioned requirements. External devices typically satisfy the resolution requirement but do not meet either the cost or integration requirements, while internal devices meet the latter but do not meet the resolution requirement.

In order to attain the resolution, cost and integration goals, we implemented the Runtime Direct Energy Measurement System (RTDEMS). RTDEMS is the adaption of the embedded low power energy-aware processing (LEAP) project [19, 24] to desktop and server-class systems. RTDEMS differs from previous desktop-class energy measurement approaches such as PowerScope [11] in that it provides *both* real-time power consumption information *and* a standard application execution environment on the *same* platform. As a result, RTDEMS eliminates the need for synchronization between the device under test and an external power measurement unit. Moreover, RTDEMS provides power information of individual subsystems, such as CPU, GPU and RAM, through *direct measurement*, thereby enabling accurate assessments of software and hardware effects on the power behavior of individual components.

2.2 RTDEMS Design

The RTDEMS implementation used in our experiments is hosted on an Intel® Core^{TM} 2 Quad CPU Q6600 2.4GHz with 2×4MB of shared L2 cache and 4GB of 1066MHz DDR2 SDRAM. Data acquisition and sampling is performed by a NI PCI-6225 data acquisition (DAQ) card capable of acquiring 250kSamples/s at 16-bit resolution. In order to measure the energy consumption of individual subsystems, we inserted 0.01Ω sensing resistors in all the DC outputs of the power supply—3.3, 5 and 12V rails. Components that are powered through the motherboard such as SDRAM DIMMs are placed on riser cards in order to gain access to the voltage pins. Power measurements are obtained by first deriving the current flowing over the sensing resistors through voltage measurements across the resistors and then multiplying with the measured voltage on the DC power connector. The DAQ card autonomously samples the voltages at the specified frequency and stores them in its buffer. A Linux driver initiates an interrupt-based DMA transfer of the buffer's content to main (kernel) memory. A Linux kernel module was implemented in order to convert the measured voltage values to energy and export them through the `proc` filesystem, thereby enabling integration with both kernel- and userspace applications. Figure 1 presents a summary of the RTDEMS energy measurement system.

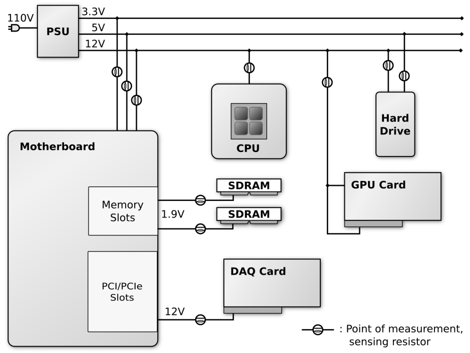

Figure 1: The RTDEMS measurement system hardware diagram.

Resolution: RTDEMS requires sufficient measurement resolution—sampling frequency—to capture the energy used within each scheduler tick in order to resolve per-process energy information. Modern multitasking systems run several processes pseudo-concurrently, by executing one runnable process after the other for a short time slice. For our Linux system this time slice is 3.3ms; thereby, the currently executing task is usually changed at the end of such a time slice. Therefore, assuming that the maximum frequency of energy information that we are interested in is 300Hz, we require a sampling frequency of at least 600Hz, based on the Nyquist criterion.

In addition to measuring the energy used within each scheduler tick, measurement resolution must be sufficient to accurately capture the power dissipation profile of the CPU as well as SDRAM. Because the CPU supply voltage is constantly $12V$, the frequency spectrum of the power dissipation profile is defined by the current signal. We used an oscilloscope to measure the current of the CPU and SDRAM channels at high frequency—5MSa/s. The power spectral density of the CPU current signal is shown in Figure 2. 99% of the CPU energy signal is contained within the first 500Hz—therefore a sampling frequency of at least 1KHz can recreate the signal and thus adequately meet both resolution requirements.

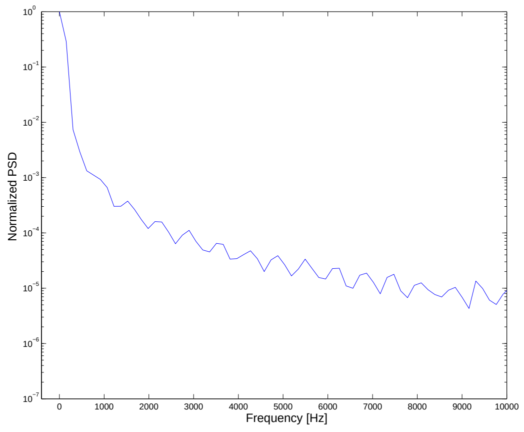

Figure 2: Power spectral density of the CPU current signal.

Overhead: The RTDEMS energy measurement system utilizes the main CPU to process power information. The process of data acquisition, conversion and storage can adversely affect the CPU performance and thus violate the *cost* requirement. The performance overhead is directly related to the sampling rate, as more samples result in larger amounts of data that need to be transferred to the CPU and processed. At the same time, a very low sampling rate will violate the *resolution* requirement, since it will provide insufficient information on the temporal domain. A set of experiments was thereby conducted to ascertain the overhead-resolution trade-off. The impact of sampling rate on CPU performance was determined by having all CPUs execute a constant workload and subsequently measuring the completion time. As Figure 3 shows, the minimum sampling frequency of

1KHz that is required to meet the resolution requirement results in a CPU overhead of less than 0.7%. It must be noted that the data acquisition overhead depends on the CPU speed—a faster CPU will result in less overhead, thereby allowing for higher sampling rates. Ultimately however, the best approach to practically eliminate the resource overhead would be to integrate the data acquisition system on the motherboard—thereby eliminating the PCI bus transactions—and perform the data conversions and energy accumulation in hardware—thereby eliminating any dependence on the main CPU. In the embedded systems space, the LEAP2 energy-aware system [24] adopted a similar approach.

As a result, we have chosen 1kHz as the operational sampling frequency for RTDEMS since it meets both the resolution and overhead requirements.

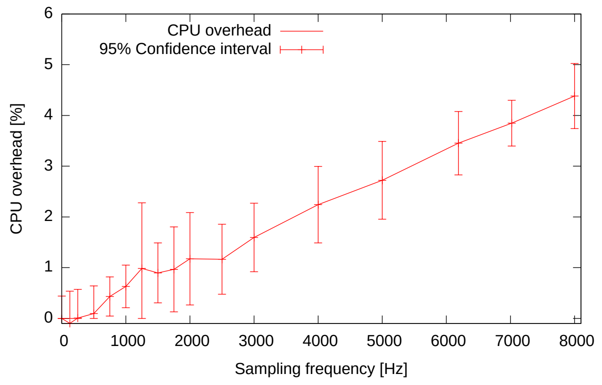

Figure 3: CPU overhead of RTDEMS as a function of sampling frequency, when sampling 11 channels concurrently.

3 Per-Process Energy Apportioning

The RTDEMS energy measurement system can accurately measure the energy expenditure on individual hardware components, such as the CPU, SDRAM, motherboard, hard drives and video card. It is reasonable then to consider whether an energy measurement system with high spatial and temporal fidelity is *sufficient* to attribute energy consumption to individual software entities such as *processes*. As mentioned in Section 1 we will focus on the energy attribution of the computational subsystem components, i.e. the CPU and SDRAM.

In a *single-core system*, only one process is executing in the CPU at any point in time. With a sampling resolution higher than the scheduler tick—so as to determine which process was executing at any point in time—attributing energy for synchronous operations (i.e. CPU and SDRAM energy) is trivial; all the energy is charged to the currently running process, which could be the idle thread or the kernel itself [24]. We therefore argue that in a single-core architecture, the ability to measure energy consumption of individual hardware components coupled with a sampling rate that is higher than the process time slice is indeed sufficient for energy attribution.

Figure 4 presents an example of energy attribution in a single-core machine. For this example, we used a simple memory access benchmark that stored data sequentially to a 512MB array—a large enough size so as to defeat all caches—and subsequently read back the stored data. The test process started at $t = 2sec$ and ended at $t = 8sec$. To simulate a single-core architecture, we executed the test program only on one of the four CPU cores of our test machine—CPU1. In addition to power information on the CPU, SDRAM and motherboard, Figure 4 also plots the CPU utilization of the four cores, as reported by the operating system. Using CPU utilization information, it is clear that the increase in power on all channels can be attributed to CPU1. However, this test also indicates that even though the CPU utilization is at a constant 100%, different components have fundamentally different power levels that also fluctuate over time, depending on their usage. Moreover, we note that even though the power state of the CPU doesn't change, the power consumption is *not constant* but varies by a significant amount, depending on the executing program's functionality. Consequently attributing CPU energy consumption solely based on the CPU utilization can lead to erroneous results [5].

In a multi-core system, per-process energy attribution is not straightforward. As the system includes multiple CPUs and CPU cores, several processes can be executing at the same time. Moreover, main memory access is now shared between multiple CPUs. Resolving per-process CPU energy consumption can be accomplished through augmenting the RTDEMS measurement system with per-CPU core measurement capabilities, assuming that CPU manufacturers can provide interfaces to such information. In the case of main memory (as well as L2 cache) such a technical solution would be infeasible, as it is a *shared resource*. For accurate per-process energy attribution of memory access, one solution would be to create a measurement system that tracks *all memory transactions that occur on the memory bus* and then correlate them with individual processes. Even though such a system is technically feasible, albeit with extensive motherboard modifications, it would generate vast amounts of data and incur very high overhead, as individual memory accesses in modern systems occur in the order of nanoseconds.

We therefore conclude that direct energy measurements, such as those provided by RTDEMS or similar systems, although necessary, are by themselves *insufficient* to resolve the per-process energy attribution problem in *multi-core systems* and that additional, *indirect* energy information is required. One obvious approach to the attribution problem is to use a utilization metric, such as CPU utilization as the indirect measurement and then attribute energy in proportion to the utilization metric, which is essentially process execution time.

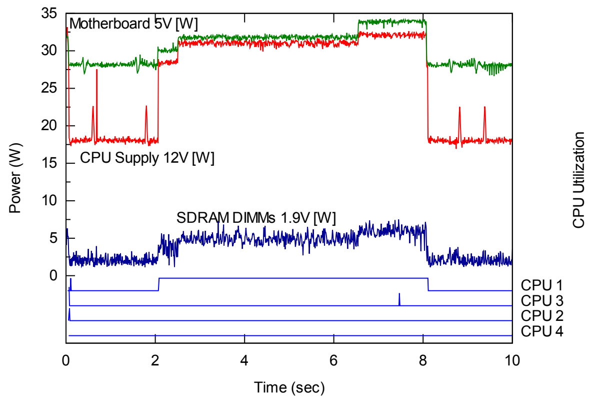

Figure 4: CPU, SDRAM and motherboard power over time for a single 512MB memory read & write test.

The following example illustrates why such an approach is not always correct. We conducted three experiments using our memory access benchmark with array sizes of 32KB for the first, 512MB for the second and 4MB for the third experiment. In each experiment, four identical memory access benchmarks were started in sequence (one per core), with a 2-second delay between each instantiation. In the first experiment, shown in Figure 5 the processes access their CPU's L1 cache only, as an array size of 32KB fits into the L1 cache. Considering that the four processes run independently of each other and on different cores but are executing the same code, attributing energy based on CPU utilization and dividing the total energy provided by RTDEMS equally is a reasonable apportion method.

In the second experiment, shown in Figure 6 all caches are defeated, since the array size is 512MB. When the first task is started it quickly fills the L2 cache and after that point the CPU is primarily stalled waiting for memory (maximum write performance of Intel Core2 Quad is less than 2bytes per CPU cycle). When the second task is started, power consumption *does not increase* (compared to Figure 5)—the two cores are now waiting for the same shared resource. Power consumption increases when the third task is started since that task, unlike the first two, is executed on the previously idle second dual-core chip. This example showcases that even though all tasks perform exactly the same operations, they have different runtimes, indicating that they execute at a different rate. It is therefore not clear if they use the same amount of energy. The apportion problem has no obvious solution unlike the first experiment.

In the third experiment, shown in Figure 7, the array size of the memory benchmark is 4MB—equal to the size of the L2 cache. When the first task is started, its memory space fits in L2 cache. Power consumption of SDRAM increases, which indicates that the CPU proactively write cache lines to main memory. As soon as the second task is started, the combined memory footprint of both tasks' data does not fit into L2 cache anymore, as L2 is shared between the two cores of a CPU. Increased access to main memory is indicated by a power increase in the SDRAM channel. The third task's data fits into the L2 cache on the second chip. Therefore it executes much faster than the two previous tasks that are constrained by main memory access. The net result is an increase in total CPU power until the fourth task is started, which forces the third task to main memory. In this experiment, the addition of a new task can lead to either an increase or a *decrease* in the total CPU power consumption. Even though all cores execute the same code, each individual task's behavior is different and dependent on all other running tasks. As a result, in this example, CPU-utilization-based apportion leads to erroneous results.

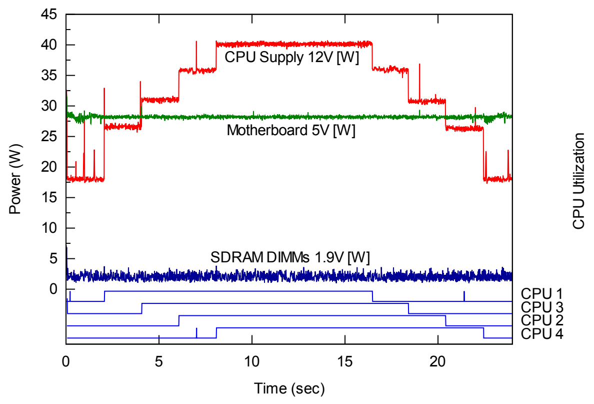

Figure 5: CPU, SDRAM and motherboard power over time for four 32KB memory read & write tests.

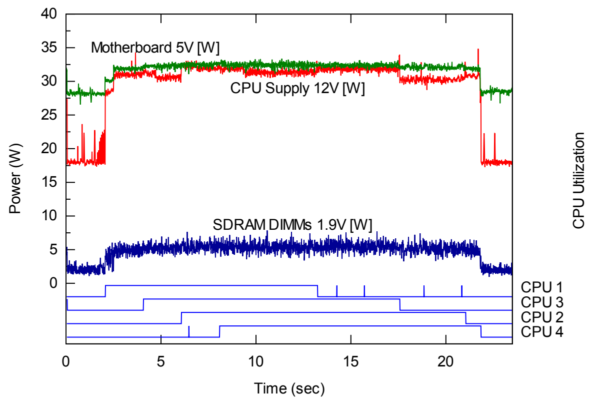

Figure 6: CPU, SDRAM and motherboard power over time for four 512MB memory read &write tests.

The aforementioned examples showcase that a simple

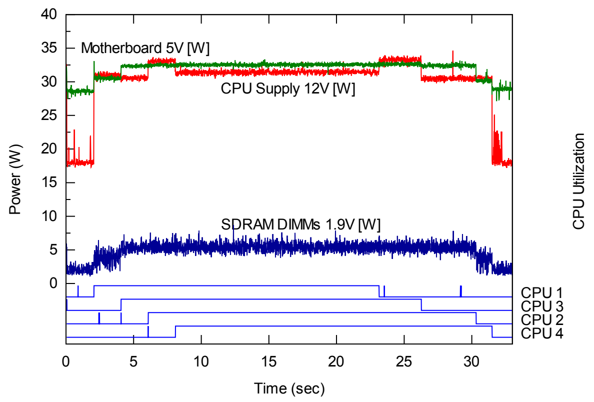

Figure 7: CPU, SDRAM and motherboard power over time for four 4MB memory red & write tests.

energy apportion solution that attributes proportional energy consumption based on CPU utilization an execution time can lead to significant errors in multi-core systems. Using such a method, the third task in Figure 7 would be charged for main memory access, while in fact it does not access main memory. Another potential solution would be to profile each application individually. However, as illustrated in the examples above, the energy consumption of a task depends on the behavior of *all* other running tasks in the system. Therefore, a more complicated indirect measurement methodology is needed.

4 Indirect Energy Measurement Model

In Section 3, we argued that direct energy measurements are a necessary but *insufficient* condition for determining the energy used by individual processes and tasks in multi-core platforms. For this purpose, we introduced the concept of indirect energy measurements. The indirect energy measurement system needs to meet the following requirements:

- The values measured should reflect a task's energy behavior.
- Appropriate models have to be defined, which allow behavioral comparisons of different tasks. Consequently, variables and models have to be found for *all* tasks running on a system.
- The comparison of different tasks should result in a "fair" energy apportion scheme.

We define a *fair* energy apportion as one that apportions the total energy in proportion to the amount of energy that would have been saved if the task would not have been executed. Given a set of tasks that uses total energy E_{total}, if task a is removed, the remaining tasks use energy $E_{\bar{a}}$. A fair apportion method would charge the energy cost $E_{\text{cost},a}$ to a task a as shown in Equation 1.

$$E_{\text{cost},a} = \frac{(E_{\text{total}} - E_{\bar{a}})}{\sum_j (E_{\text{total}} - E_{\bar{j}})} * E_{\text{total}} \qquad (1)$$

As a consequence, a's energy cost depends on all other tasks executed concurrently, that might or might not be under a's control. We argue that this is *fair* for the following reasons. First, tasks should be charged for the energy consumption they cause, which depends on the other tasks and can result in either energy benefits or savings (see Section 3). Also, application developers should be encouraged to write efficient code and not be rewarded for inefficient and suboptimal multi-threaded programming.

In this Section, we will present an indirect energy measurement model which is well suited for the apportion of SDRAM as well as CPU energy.

4.1 Performance counter behavioral model

Most modern processors, whether embedded, desktop or server-class, contain a *performance measurement unit* (PMU) which is capable of counting a variety of different processor related events. Performance counters are typically used to profile applications and optimize their performance. In the energy estimation domain, Bellosa et al. used performance counters to predict CPU temperature for dynamic thermal management [5]. They propose a linear, event-based model to estimate the energy consumed by a single-core CPU. They show that the CPU's energy consumption E can be modeled as a sum of event counts c_i multiplied by event energy e_i, as in Equation 2.

$$E_{\text{est}} = \sum_i c_i * e_i \qquad (2)$$

We argue that performance counters are suitable indirect indicators for energy apportion. Performance counters are available on most modern processors and can be accessed without incurring significant overhead. Additionally, performance events can be counted for each core separately and can therefore measure the behavior of each core individually, thus providing the additional visibility that our direct measurement system lacks. Finally, previous work has shown performance events to be good *indicators* for energy usage [5, 20, 17, 13]. We note that, unlike Bellosa et al., we do not use performance counters to *estimate* total energy consumption as our RTDEMS measurement system provides us with direct and accurate measurements. Rather, after acquiring the total energy consumption through RTDEMS, we use performance counters to solve the *energy apportion problem*. We also extend prior work by using performance counters as indicators for SDRAM energy consumption, in addition to CPU energy consumption.

Both CPU and SDRAM are complex systems that contain numerous subsystems. Several of those subsystems can be shared among running processes at any point in time. For example, our Intel® Core^{TM} 2 Quad CPU consists of two separate dual-core CPUs with 4MB of L2 cache each. Therefore, depending on which core two processes are executing, they either have access to a shared 4MB L2 cache, or to two individual 4MB L2 caches. We use models to estimate the tasks' energy behavior relative to each other and do not try to model absolute CPU energy consumption, which would require to model all subsystems and inter-task dependencies. It is sufficient to learn the event model for the single-core case as this provides an apt approximation of $E_{\text{total}} - E_{\bar{a}}$ for Equation 1.

$$E_{\text{cost},a} = \frac{E_{\text{est},a}}{\sum_j E_{\text{est},j}} * E_{\text{measured}} \tag{3}$$

Equation 3 shows how we apply performance event based models for *fair* multi-core energy apportion. The total measured energy E_{measured} is apportioned among a set of tasks. The energy cost $E_{\text{cost},a}$ charged to a task a can be calculated by dividing total energy E_{measured} proportional to $E_{\text{est},a}$, which is the energy the task's behavior would have cost in single-core operation. As a consequence, additional costs as well as energy savings resulting from running tasks on a multi-core system, are split equally among the tasks.

We defined E_{measured} as the energy cost caused by the running tasks. For that reason, we subtract the dc part from all energy and power measurements. We think, this is a "fair" policy.

4.2 Model learning

In order to learn and test our model, we use a variety of different microbenchmarks and actual applications typically found on desktop or server systems. Microbenchmarks include burnMMX and burnP6, a memory and CPU stress test from the cpuburn package, mem, our own memory access test capable of accessing memory in various ways (Section 3), qsieve integer factorization and linpackc. Applications include sort, md5sum, multimedia encoders (mm), lame (mp3) and oggenc (ogg vorbis), imagemagick (immck), compilation of the Linux kernel using gcc, the web browsers firefox and epiphany and the webservers apache and thttpd. As shown in Figures 8 and 9 these applications have very different characteristics, both in terms of power and in terms of performance events.

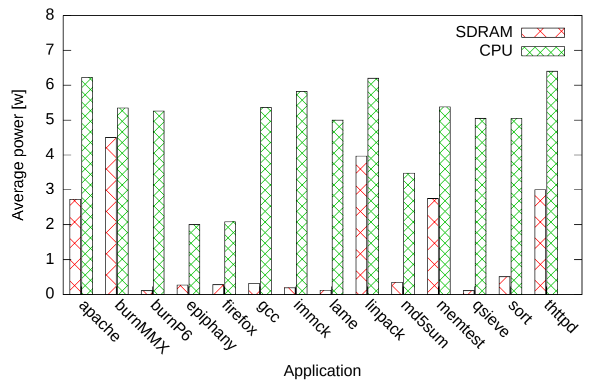

Figure 8: Average CPU and SDRAM power usage of different applications.

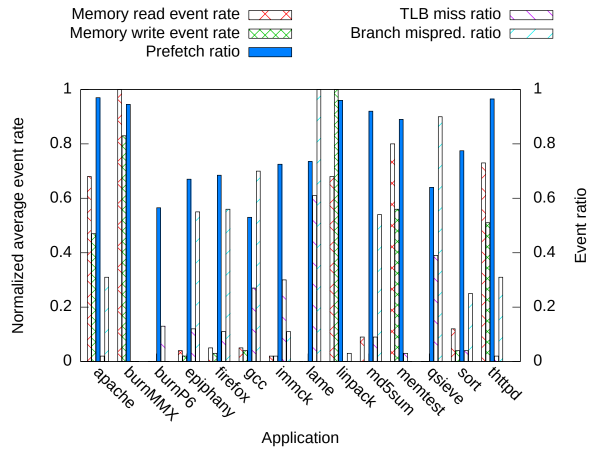

Figure 9: Normalized performance event rates and event ratios of different applications.

4.2.1 Performance event selection

Performance measuring units can track dozens of different events, albeit only few at the same time. For instance, the Intel® Core^{TM} 2 CPU used on the RTDEMS is capable of counting well over 100 different events. However, for each core, only five different events can be measured simultaneously and three of those are predefined and cannot be changed. As a result, we need to select two events that, together with the three predefined events can constitute a reasonable set of indirect measurement indicators for energy consumption.

One approach to this selection problem is to write micro benchmarks that cause a limited set of PMU events, thus facilitating the calculation of event energies e_i for all possible events. Then, events with the highest energy contribution $e_i * c_i$ (Equation 2) can be chosen. However this approach is not optimal as event energies are not constant but depend on both the running application and the model. In general, a PMU event can be caused by a set of different hardware events (subevents) with different energy costs. For example, subevents for the PMU event `Memory Read` include random reads within the same row, random reads that induce a row change, and burst reads. The relative frequencies of those subevents and subsequently the average cost of a PMU event depend on the application. Similarly, many PMU events are not independent of each other. For example, translation lookaside buffer (TLB) misses are for most applications highly correlated with memory reads. A model not counting TLB misses will likely include their cost into the cost of memory reads.

Our event selection methodology consists of the following steps. First, we manually prune the search space of events that are not likely to have an impact on power consumption. Second, we gather model learning and test data. Because running a test multiple times always results in the same total event count, it is sufficient to run each test as many times as required to get the total counts for all events and total energy. Finally, we systematically build models for SDRAM and CPU energy and compare their performance.

For our system, we found through exhaustive search of the reduced search space that the events for 'lines read into the last-level cache', 'modified lines evicted from the last-level cache' and 'instructions retired' are well suited to model SDRAM as well as CPU energy.

4.2.2 Event energy learning

After having selected the appropriate PMU events, their event energies e_i must be learned. As mentioned in Section 4.2.1, event energies vary slightly between applications. For example, when reading an 512MB buffer sequentially the energy cost of reading a cache line is 52nJ. When reading the same buffer randomly, the cost rises to 69nJ.

Using time series of energy and event data as opposed to total values for event energy learning provides more data points and makes several variables independent even if there is only limited training data available per application. It is therefore possible to learn the event weights for an application with few tests. Overlearning becomes an issue when an application's counts for certain events are too low to have a measurable impact on power or when two counters are highly correlated. For these events, the application's costs are set to those from the generic model. We thus ensure that each application's model is valid even if the application changes its behavior in the future. Table 1 shows the event energies estimated with this method.

Figure 10 shows the R^2 value of the single-core energy estimation for both SDRAM and CPU using three event counters only. The performance of the model is very good for all test applications, even for very complex programs like Firefox or a complete Linux kernel compilation. An increase in the number of PMU counters and energy-relevant performance events would lead to improved model performance.

Application	*SDRAM Model*		*CPU Model*		
	Memory Reads	*Memory Writes*	*Instr. Retired*	*Memory Reads*	*Memory Writes*
generic	56	63	2.1	121	273
apache	66	67	3.1	241	266
browsers	59	63	2.5	128	252
burnMMX	55	59	2.1	120	264
burnP6	55	64	2.0	124	277
gcc	57	64	3.2	98	296
immck	56	63	1.9	151	264
lame	57	62	2.6	95	265
linpack	55	63	1.9	134	233
md5sum	56	63	2.3	116	269
memtest*	52	61	1.6	113	265
memtest†	69	85	2.0	185	325
qsieve	56	64	2.6	98	272
sort	68	55	2.4	159	215
thttpd	63	72	2.6	162	387

Table 1: Event energies in nJ for different applications.
* sequential read/write, † random read/write

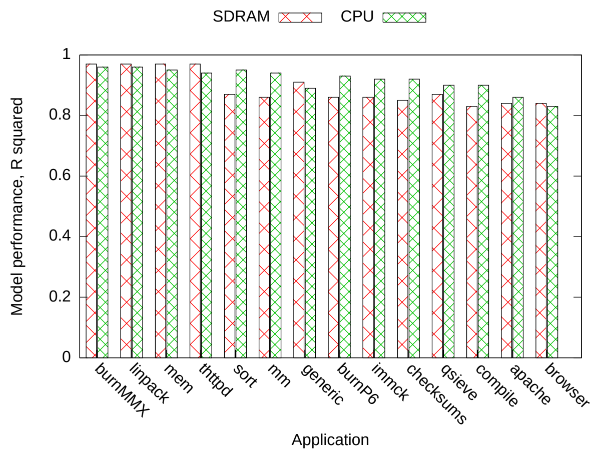

Figure 10: Single-core energy estimation performance.

5 The Energy Apportion & Accounting System

The purpose of the energy accounting system is to accurately account for the energy used by processes within a computer system. The system also provides runtime per-process energy usage information to the operating system and to user space programs. The energy accounting system builds on the RTDEMS real-time energy measurement capability. Consequently, the accounting system should meet the same requirements regarding resolution, cost, and integration stated in Section 2 as well as the following:

- *Modularity*. The accounting system and its subsystems should be modular components of the operating system.

- *Latency*. Per-process accounting requires tight integration with critical code paths of the operating system. Low latency on these critical paths is paramount for the operating system and therefore a crucial requirement for our software architecture.
- *Parallelism.* The software architecture must be optimized for multi-core operation.

Figure 11 shows an architectural diagram of the energy accounting system. It is comprised of several Linux kernel modules together with a small kernel patch that adds energy information to Linux's process management. The system uses real-time energy samples acquired by RTDEMS. Energy is ultimately charged to *resource containers*, our energy cost accumulation data structures [4, 25]. The core of the system is the energy apportion and accounting component, which asynchronously processes the CPU's activity lists in order to apportion the energy measured using the performance counter attribution method introduced in Section 4.

5.1 Resource containers

As our energy accounting data structures we use resource containers, a well-known OS abstraction that is used for accounting usage costs of several shared OS resources. Resource containers separate the concept of a resource consuming entity from processes, which allows more fine-grained accounting. Resource containers accumulate energy values for all hardware components individually. Each process and thread is associated with a resource container by means of resource binding [4]. Processes and threads constitute the *accounting entities* of our system. When an an accounting entity is active, its energy consumption is charged to the respective container. We utilize dynamic resource binding to allow binding of any resource container to a process or thread at any time. This makes it possible to implement systems other than per-process accounting, such as per-activity accounting [4, 25].

5.2 Per-process accounting subsystem

On a single-core machine, each process would be charged the energy E_{measured} measured during the time slice the process ran uninterruptedly.

On a multi-core machine however, tasks are executed on all n cores independently. The energy apportion algorithm apportions energy among up to n tasks that run *concurrently*. For that reason, we have to divide the uninterrupted execution time of a process into segments (time slices) that do not include any task switches on any CPU. The energy used during those time slices is proportionally attributed to concurrently running tasks, through the energy apportion algorithm. Figure 12 shows two CPUs running three tasks a, b, and c and the resulting energy apportion time slices Δt_i. For Δt_1, total energy is divided among tasks a and b, in Δt_2 among a and c, in Δt_3 all energy is charged to c, and so on.

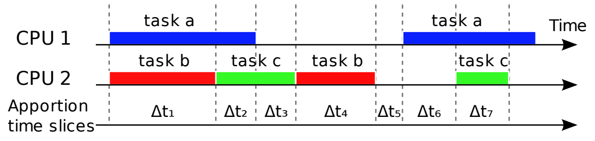

Figure 12: Energy accounting time slicing for three tasks a, b and c running on two CPUs.

The *energy apportion time slice* is in general smaller than the *scheduler time slice* since tasks can block or be interrupted by a high priority task before their scheduler time slice expires. Additionally, while a task a is running uninterruptedly, a task switch might happen on another CPU, leading to segmentation.

For each time the scheduler selects a new task on any of the cores, the apportion algorithm needs to be executed, and energy charged accordingly. It is not practically feasible to do the energy apportioning immediately when a task switch is scheduled. The computational load of the algorithm would lead to a significantly increased latency in the scheduler. More importantly, the accounting would have to be synchronized among the CPUs, which would result in a blocking scheduler and therefore a momentous latency overhead.

The following paragraphs describe the following main components of our architecture depicted in Figure 11: Per-CPU activity logs, model management, task management and the energy apportion and accounting algorithm.

Per-CPU activity logs: To minimize the latency in the scheduler and to solve the synchronization issue, we introduced *per-CPU activity logs*. The logs keep track of the scheduling of tasks, on a per CPU basis. Using per-CPU log information, the expensive apportion computation can be deferred and need not be executed on every context switch. In addition, since information is recorded per CPU, log access need not be synchronized.

Our energy apportion algorithm needs the following data as input: *a)* the energy E_{measured} as measured by the RTDEMS, *b)* the concurrently running tasks among which E_{measured} is divided, and *c)* the model values c_i and e_i needed by the apportion algorithm introduced in Section 4.

While E_{measured} is provided by the RTDEMS' energy log, all other data must be stored within the activity logs. Consequently, log entries contain a time stamp, a pointer to the task's current resource container, a pointer to the task's current model information and a memory region for each of the task's energy models. The models utilize those memory regions to store values c_i required in order

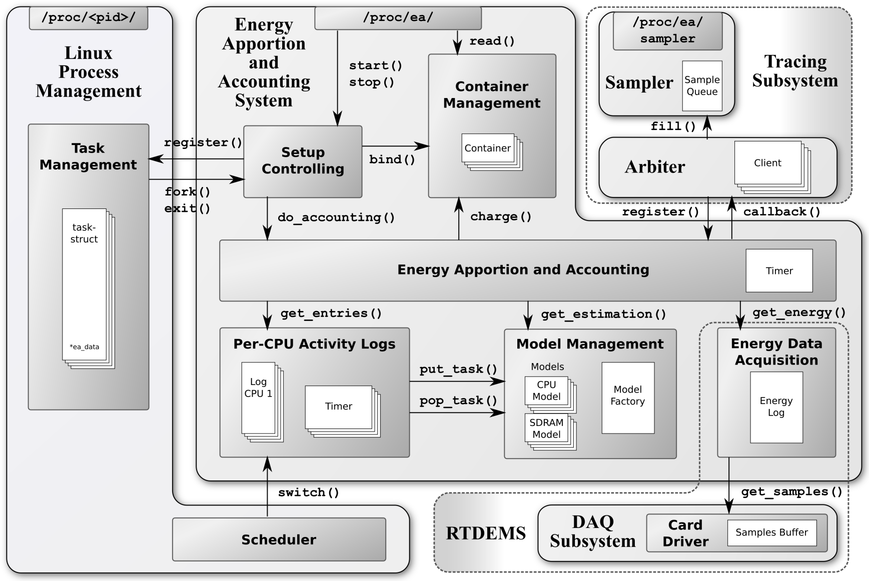

Figure 11: Architectural diagram of the energy accounting software system.

to determine the task's energy behavior during the time period since the last entry.

Model management: Our system uses application- and device-specific energy models to assess the processes' respective energy behavior. Each model is defined by an interface consisting of the following five functions:

- `put_task()` This function initializes the model's memory area for a task that has just been selected by the scheduler. For instance, the PMU model uses this function to store the current, initial performance counter values in the memory area. The model could also configure the PMU to count different events that are better suited to determine the energy behavior of this application.
- `pop_task()` This function finalizes an activity log entry for a task that is removed from the CPU. For example, the PMU model reads the new performance counter values, subtracts the initial values, and saves the difference in the memory area.
- `estimate()` This function calculates E_{est} of a given activity log entry. For example, the PMU model calculates E_{est} using its application specific event energies and the performance event counts stored in the entries' model memory area.
- `init(), exit()` These functions are called upon loading and exiting the energy accounting system in order to initialize and terminate the model.

Task management: In order to enable both per-process accounting and application specific models, we extended Linux's process data structure `task_struct` to include a pointer to a dynamically allocated data structure `ea_data`. This structure contains task specific information needed by the energy accounting system such as a pointer to the task's current resource container and models—one for CPU and one for SDRAM energy.

Dynamic allocation of the energy accounting data structure enables dynamic binding of the task's model and resource container. This feature allows the accounting system to change the model at any point without data loss. Because activity log entries contain a pointer to the task's currently valid `ea_data`, entries are always processed using the right model, even if the task's model binding has changed by the time the accounting thread becomes active.

Dynamic allocation necessitates the inclusion of a reference counter, which indicates if `ea_data` is still being referenced by either a task or an activity log entry. As soon as the reference counter becomes zero, the data structure can be freed.

Besides augmenting the process data structure `task_struct`, we created an interface to Linux's process management that allows us to manage per-process energy data. When a process or thread is created, changed or exits, its `ea_data` structure must be updated. In order to not violate the modularity requirement by including this functionality into Linux's process management, we defined an interface, through which the energy accounting system can register callbacks for the following events:

- `fork()` and `exit()`: Create or delete `ea_data`.
- `exec()`: Reevaluate the task's model and possibly choose new application specific models.
- `switch()`: The scheduler is about to switch to another task. Insert new entry into activity log.

In addition, the process management makes per-process energy information accessible from user space using the process file system by reading the file `/proc/<pid>/ea`.

Energy apportion and accounting: Our design allows to defer the apportion and accounting algorithm and execute it in a dedicated kernel thread. This thread runs periodically or on demand on any CPU, preferably an idle one. The apportion and accounting algorithm works as follows:

1. Get activity log entries for the next time slice Δt from each CPU.
2. For each resource (CPU, SDRAM)
   (a) Get E_{measured} for Δt from the RTDEMS' energy log.
   (b) Get E_{est} of each entry by executing the `estimate()` function of the model which is referenced by the entries' `ea_data` reference.
   (c) Charge each entry's resource container according to the apportion rule introduced in Section 3.

Since all CPUs create activity log entries independently, the entries in general do not correspond to the same time slice. To apportion the energy of a time slice Δt, the entries have to be of the same length. Therefore, the length of the time slice is defined in step 1 by the shortest entry. All longer entries are split into an entry of size Δt plus the remainder of their own time slice. We assume that the behavior of a task is approximately uniform during a time slice, such that model values can be divided up linearly among the parts. Because activity log entries created by the scheduler have an arbitrary length, a timer inserts new entries at a frequency for which the above assumption is reasonable—20Hz in our implementation.

As a consequence of deferring energy accounting, the resource containers are updated with a delay. The maximum delay is limited by the period at which the accounting thread is run. Each resource container counts the number of its activity log entries, that are not yet accounted for. If the count is zero, a resource container's energy values are up-to-date.

5.3 Tracing subsystem

In addition to the energy accounting system, we also implemented a tracing subsystem that provides an interface for monitoring and accurate energy measuring of tasks. This subsystem acquires values such as measured energy E_{measured}, energy used by individual CPUs, model values c_i, and model data e_i and makes them available as time series data. Values can be traced continuously or only when a given process is active. The tracing subsystem is used by our model learning tools and also enables application developers to analyze the energy consumption of a single application.

The tracing subsystem consists of two modules: the arbiter module and the sampler module. The arbiter module acquires and aggregates data that is subsequently provided to multiple clients. Whenever the accounting algorithm processes a set of activity log entries, it informs the arbiter which aggregates and distributes the samples to its clients' sample queues. Because this causes an additional overhead for the accounting algorithm, we designed the tracing subsystem as optional kernel modules, which should only be activated when their functionality is necessary. The sampler module exports the tracing interface and the samples to user space using the process file system.

6 Evaluation

In this section, we present an experimental evaluation of the accuracy of the apportion algorithm and demonstrate the functionality of our system using typical desktop applications. In addition, we investigate the impact of our system on CPU resources.

6.1 Per-process energy apportion accuracy

The main part of the energy apportion algorithm is the set of performance event models introduced in Section 4. In order to ascertain the accuracy of these models, we would need to compare the results to those obtained by an a priori accurate measurement system. However, as mentioned in Section 3, without extensive motherboard and CPU modifications, we cannot measure the energy consumption of individual CPU cores or the percentage of SDRAM energy usage caused by a particular process. As a result, the correct solution of the apportion problem (i.e. ground truth) is in the general case not known.

In order to test the accuracy of the apportion algorithm we therefore designed experiments for which we are able to assert a particular apportion. We then compare the asserted values with the solution found by our online algorithm. We chose a sequential memory access benchmark as our test program, as it allows us to test SDRAM as well as CPU energy apportioning, by controlling the number of memory accesses. We also chose a memory buffer of 512MB in order to minimize the impact of the CPU's cache management, which is beyond our control.

By executing two instances of the memory benchmark, A and B concurrently on two different cores and by controlling the number of accesses over the memory buffer, we assert the energy apportioning to be proportional to the number of memory accesses performed by the two processes. For example, if process A accesses the memory buffer once while process B accesses it twice, we assert that the correct memory energy attribution would be 33% for process A and 66% for process B. We note that this assertion holds for our benchmark because the *type* and *locality* of memory accesses is the same for both processes, as opposed to arbitrary processes and tasks, where neither type nor locality can be known in advance.

Figure 13 depicts the result of our experiments for both CPU and SDRAM energy attribution. The x-axis depicts the asserted value of process A as a percentage of the total energy value of $A + B$, i.e. $\frac{E_A}{E_A+E_B}$, while the y-axis shows the equivalent measured result from our energy apportioning system. An ideal energy apportioning system would have a y-axis value that would match the corresponding x-axis value. As seen in Figure 13, our energy apportioning system is very accurate, with a maximum deviation of up to 4% of the equivalent asserted values. The accuracy achieved in this experiment makes it reasonable to assume that our system provides the correct apportion for arbitrary applications.

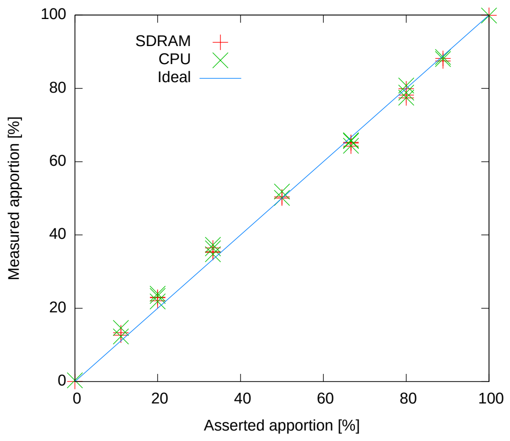

Figure 13: Asserted and measured CPU and SDRAM energy apportion of two tasks a and b.

6.2 CPU Overhead

As mentioned in Section 2 and 5, our energy accounting system needs to operate with the lowest possible overhead. In Section 2 we investigated the RTDEMS overhead in terms of CPU resources. However RTDEMS-induced overhead is only part of the energy accounting system's overhead. In addition, overhead is caused by the insertion of entries into the activity logs and subsequent processing by the apportion and accounting thread. As a consequence, the overhead is expected to depend on the scheduler's task switching activity. Whenever the tracing subsystem described in Section 5.3 is active, the accounting thread is required to supply the arbiter with time series data, thus creating additional overhead.

In order to determine the overhead of our system on CPU resources, we measured the CPU time spent within the energy accounting system. This was determined using the processor's time stamp counter which provides nanosecond time resolution with minimal CPU impact. To quantify the impact of scheduling activity, we implemented a microbenchmark that periodically performs a CPU-bound computation and then causes a task switch by yielding the processor. The task switching frequency can thus be controlled by modifying the duration of the CPU-bound computation. Each task switch leads to an additional entry in the per-CPU activity log as described in Section 5.2. The CPU overhead depends on the *rate* of modifications (insertions and deletions) on the per-CPU activity log. Therefore, increased scheduling activity (task switching frequency) is expected to incur higher overhead.

We conducted experiments using two tasks per CPU, variable task activity periods, and by enabling and disabling the tracing subsystem. Figure 14 shows the CPU overhead of the energy apportion system. We note that even at very high task switching frequencies of 300Hz the impact on the CPU is less than 0.45% and is reduced to less than 0.2% at switching frequencies of 10Hz or less. On the other hand, the activation of the tracing subsystem results in a relatively substantial overhead of 0.6% at high task switching frequencies. Acquisition of time series data is a relatively expensive operation thereby justifying our design decision to implement the tracing subsystem as an optional module.

6.3 Application apportioning

Table 2 shows the result of the apportion system for four concurrently running applications. Our application set includes the apache web server, a gcc compilation of parts of the boost library, sorting a 128MB file of 100-byte length random integers and an image blurring process. The applications have different runtimes and—as expected—different energy footprints which are reflected in the resulting energy apportion.

Our experimental results demonstrate that our energy attribution and accounting system has achieved its design goals of providing integrated and accurate—96%

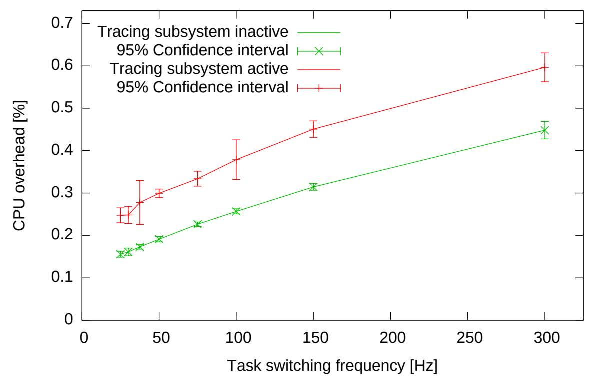

Figure 14: CPU overhead of the energy apportioning and accounting system as a function of task switching frequency.

Application	*CPU [J]*	*SDRAM [J]*	*Runtime [sec]*
apache	66.8	2.8	14.9
gcc	105.0	4.3	19.2
sort, 128MB	169.8	5.5	20.0
image blur	97.2	0.6	10.3

Table 2: CPU and SDRAM energy apportion for four concurrently running applications.

of optimal—per-process energy accounting of individual hardware components, incurring only up to 0.6% of CPU overhead.

7 Related Work

Prior work on energy measurement for server systems typically focuses on global server power consumption surveys, such as Binachini et al. [7] and Koomey et al. [16]. External power measurements have been used by Chase et al. [8] to optimize the energy consumption of a hosting center by dynamically resizing the server set of a cluster. As explained in Section 2 external measurements are not suitable for per-process accounting as they do not meet the integration and resolution requirements.

An alternative approach to determine a server's energy consumption is through estimation techniques. ECOSystem [27] and Rivoire et al. [22] use a power state based model and associate a fixed power consumption to each state. Kansal et al. [15] propose a similar model for application energy profiling. Using a per-component utilization based approach, Mantis [10] learns and predicts the power consumption of a server system for different workloads. Our work does not rely on estimation techniques, as our RTDEMS system provides us with direct energy measurements at high temporal and spatial resolution.

Bellosa et al. [5, 20] introduced linear performance event based models for CPU energy estimation and utilized them for dynamic thermal management. Our energy behavior models and methods of energy apportioning among tasks are based on this work. Our system extend this previous work by providing application specific models for accurate apportioning and also extends the PMU models to include main memory energy consumption.

Isci et al. [13] used performance counters to estimate the energy consumption of processor subsystems from power external measurements. Similarly, Lewis et al. [17] proposed a method to calculate per-component energy from AC power measurements. Alternatives to performance event based models are SimplePower [26], an instruction level emulator and energy estimator, or regulator switching cycles based energy models as proposed by Dutta et al. [9]. However, those systems do not provide per-task resolution and thus cannot be easily adopted to solve the multi-core energy attribution problem.

Resource containers [4] are a well-known operating system abstraction. They have been proposed and implemented for FreeBSD [4] and for Linux [25]. In addition, Jones et al. [14] designed a modular resource management for the Rialto operating system. Our system builds upon previous work by providing the first implementation of resource containers for multi-core systems and by solving the energy apportion problem.

8 Conclusion

This paper introduces a new energy attribution software architecture that augments the operating system of a *multi-core platform* with runtime per-process energy usage information. Our system utilizes runtime *direct* energy measurements that provide accurate per-component energy usage information at millisecond-scale resolution. We argue that per-process energy accounting on a multi-core or multi-processor platform necessitates the use of *indirect energy measurements*. As a solution to this energy apportion problem we introduce performance counter based energy behavior models. We experimentally demonstrate that our models exhibit high energy estimation accuracy for single-core experiments with both microbenchmarks and actual applications, thereby providing an apt measure for the apportion of both CPU as well as SDRAM energy.

We fuse the direct and indirect portions of our system in a combined energy apportion and accounting software system, designed as a low-overhead modular component of the Linux operating system. Our experiments demonstrate that our energy apportioning system can successfully provide per-process energy consumption with over 96% accuracy, while impacting CPU performance by less than 0.6%.

In the future we plan to extend our system to account energy usage of other components such as hard drives and network cards, which requires the design and implementation of suitable energy models. Furthermore, we aim to replace the initial calibration phase necessary for model learning with an online model learning system. We designed our system with future expansion to alternative accounting systems like per-activity accounting in mind. We also intend to build on our resource container implementation and provide a more powerful interface for container manipulation to userspace applications.

References

[1] Advanced configuration and power interface specification, 2005. `http://www.acpi.info`.

[2] Manish Anand, Edmund B. Nightingale, and Jason Flinn. Ghosts in the machine: interfaces for better power management. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 23–35, New York, NY, USA, 2004. ACM.

[3] Manish Anand, Edmund B. Nightingale, and Jason Flinn. Self-tuning wireless network power management. *Wirel. Netw.*, 11(4):451–469, 2005.

[4] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.

[5] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, New Orleans, LA, September 27 2003.

[6] Luca Benini, Giuliano Castelli, Alberto Macii, Enrico Macii, and Riccardo Scarsi. Battery-driven dynamic power management of portable systems. In *ISSS '00: Proceedings of the 13th international symposium on System synthesis*, pages 25–30, Washington, DC, USA, 2000. IEEE Computer Society.

[7] Ricardo Bianchini and Ram Rajamony. Power and energy management for server systems. *Computer*, 37(11):68–74, 2004.

[8] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 103–116, New York, NY, USA, 2001. ACM.

[9] Prabal Dutta, Mark Feldmeier, Joseph Paradiso, and David Culler. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, pages 283–294, Washington, DC, USA, 2008. IEEE Computer Society.

[10] Dimitris Economou, Suzanne Rivoire, Christos Kozyrakis, and Partha Ranganathan. Full-system power analysis and modeling for server environments. In *Workshop of Modeling, Benchmarking, and Simulation*, 2006.

[11] Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *WMCSA '99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, page 2, Washington, DC, USA, 1999. IEEE Computer Society.

[12] William A Hammond. Efficient power consumption in the modern datacenter. Technical report, Digital Enterprise Group, 2005.

[13] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93, Washington, DC, USA, 2003. IEEE Computer Society.

[14] M. B. Jones, P. J. Leach, R. P. Draves, and . Iii Barrera J. S. Modular real-time resource management in the rialto operating system. In *HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, page 12, Washington, DC, USA, 1995. IEEE Computer Society.

[15] Aman Kansal and Feng Zhao. Fine-grained energy profiling for power-aware application design. *SIGMETRICS Perform. Eval. Rev.*, 36(2):26–31, 2008.

[16] Jonathan G. Koomey. Estimating total power consumption by servers in the u.s. and the world. *Analytics Press*, February 2007.

[17] Adam Lewis, Soumik Ghosh, and N.-F. Tzeng. Run-time energy consumption estimation based on workload in server systems. In *HotPower '08, San Diego*, 2008.

[18] Dimitrios Lymberopoulos, Nissanka B. Priyantha, and Feng Zhao. mplatform: a reconfigurable architecture and efficient data sharing mechanism for modular sensor nodes. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 128–137, New York, NY, USA, 2007. ACM.

[19] D. Mclntire, K. Ho, B. Yip, A. Singh, W. Wu, and W.J. Kaiser. The low power energy aware processing (LEAP) embedded networked sensor system. *Information Processing in Sensor Networks, 2006. IPSN 2006. The Fifth International Conference on*, pages 449–457, April 2006.

[20] Andreas Merkel, Frank Bellosa, and Andreas Weissel. Event-driven thermal management in SMP systems. In *Second Workshop on Temperature-Aware Computer Systems (TACS'05)*, Madison, USA, June 2005.

[21] Dinesh Ramanathan and Rajesh Gupta. System level online power management algorithms. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 606–611, New York, NY, USA, 2000. ACM.

[22] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 365–376, New York, NY, USA, 2007. ACM.

[23] Ishan Sehgal and Michael Patterson. Cool crunching: Understanding green hpc. Technical report, IBM and Intel, 2008.

[24] Thanos Stathopoulos, Dustin McIntire, and William J. Kaiser. The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes. *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, pages 383–394, April 2008.

[25] Martin Waitz. Accounting and control of power consumption in energy-aware operating systems.

[26] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 340–345, New York, NY, USA, 2000. ACM.

[27] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: managing energy as a first class operating system resource. *SIGPLAN Not.*, 37(10):123–132, 2002.