LEA^{2}P

The Linux Energy Attribution and Accounting Platform

Diploma Thesis of Sebastian Ryffel[†]

Advisor: Dr. Thanos Stathopoulos[‡]

Supervisors: Prof. Dr. Lothar Thiele[†] and Prof. Dr. William J. Kaiser[‡]

[†]Swiss Federal Institute of Technology (ETH) Zurich, Switzerland

[‡]UCLA, Department of Electrical Engineering

July 21, 2008 - January 18, 2009

Abstract

This thesis presents a novel energy attribution and accounting architecture for multi-core systems that can provide accurate, per-process energy information of individual hardware components. I 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. I also introduce a performance counter based behavioral model that provides indirect information on the proportional energy consumption of concurrently executing processes in the system. I 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 I show that my energy apportioning system achieves an accuracy of at least 96% while impacting CPU performance by less than 0.6%.

Contents

Chapter 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 thesis introduces the Linux Energy Attribution and Accounting Platform (LEA^{2}P). Using a combination of detailed, hardware-assisted energy measurements and indirect energy measurement models, LEA^{2}P provides the first (to the best of my knowledge) runtime, low-overhead, integrated energy monitoring of *individual processes* executing concurrently on a *multi-core platform*. My thesis 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, I 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. I argue that direct energy measurements such as those provided by RTDEMS are *necessary*, albeit *insufficient* to determine per-process energy attribution. I 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 the LEA^{2}P kernel-space software system, I demonstrate through experimentation that I 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 thesis 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 (Chapter 2).
- An experimental analysis of the energy apportioning problem in multi-core systems, using RTDEMS-obtained data (Chapter 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 (Chapter 4).
- The LEA^{2}P, 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 (Chapter 5 and 6).

I have used my system to attribute energy consumption to several applications on a multi-core platform and present my results in Chapter 7. I present related work in Chapter 8 and conclude the thesis in Chapter 9.

Chapter 2

The Runtime Direct Energy Measurement System

In this Section, I provide an overview of direct energy measuring techniques and also describe my detailed real-time direct energy measurement system. Using empirical information and experimental results, I 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 my overall architecture, as it provides the necessary information regarding energy consumption. As my 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.

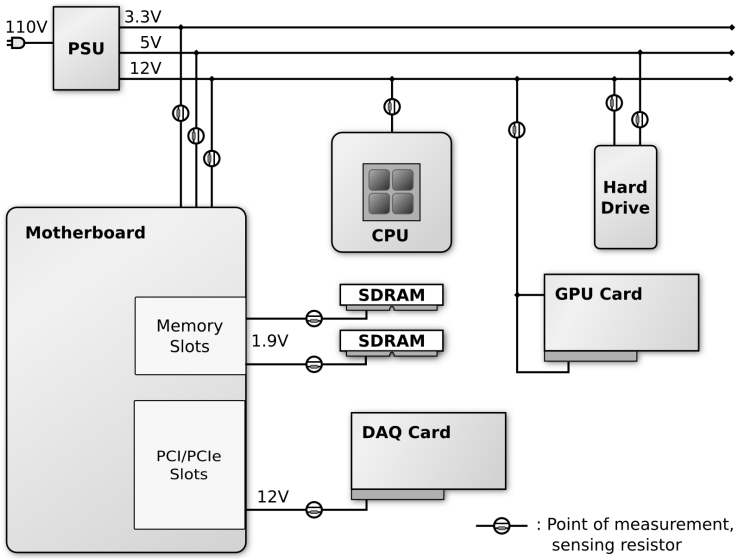

Figure 2.1: The RTDEMS measurement system hardware diagram.

In order to attain the resolution, cost and integration goals, I 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 my experiments is hosted on an Intel® Core™ 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, 0.01Ω sensing resistors were inserted 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 user space

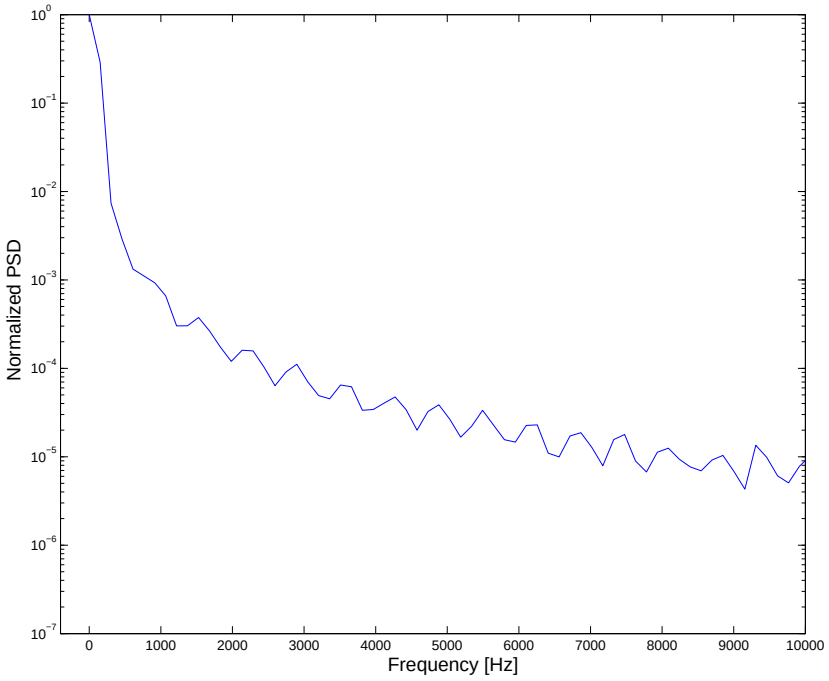

Figure 2.2: Power spectral density of the CPU current signal.

applications. Figure 2.1 presents a summary of the RTDEMS energy measurement system.

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 my 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 is of interest is 300Hz, a sampling frequency of at least 600Hz is required, 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. I 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 2.2. 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.

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 sam-

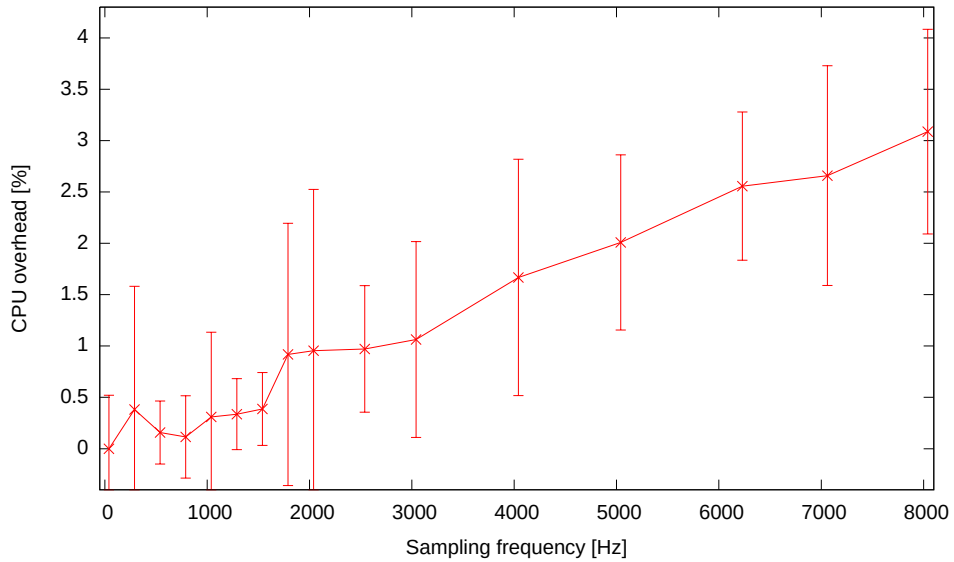

Figure 2.3: CPU overhead of RTDEMS as a function of sampling frequency, when sampling 11 channels concurrently with 95% confidence interval.

ples 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 2.3 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%. When only sampling the computational subsystem—CPU and SDRAM—instead of all eleven of RTDEMS' channels, the overhead diminishes and becomes not measurable anymore. In conclusion, this experiment shows that for all practical purposes the overhead is negligible when sampling CPU and SDRAM energy only. 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, I have chosen 1kHz as the operational sampling frequency for RTDEMS since it meets both the resolution and overhead requirements.

AC power consumption: The following experiment assesses the impact of the RTDEMS measured per-component energy on the whole system's AC power consumption. This experiment generates different load patterns for the CPU and SDRAM, while also changing the CPU frequency, which can be either $2.4GHz$ or $1.6GHz$. Figure 2.4 shows the correlation between component energy—CPU and SDRAM—and PSU AC power of the system. Although the AC power is very noisy, it is obviously proportional to a smoothed linear combination of CPU and

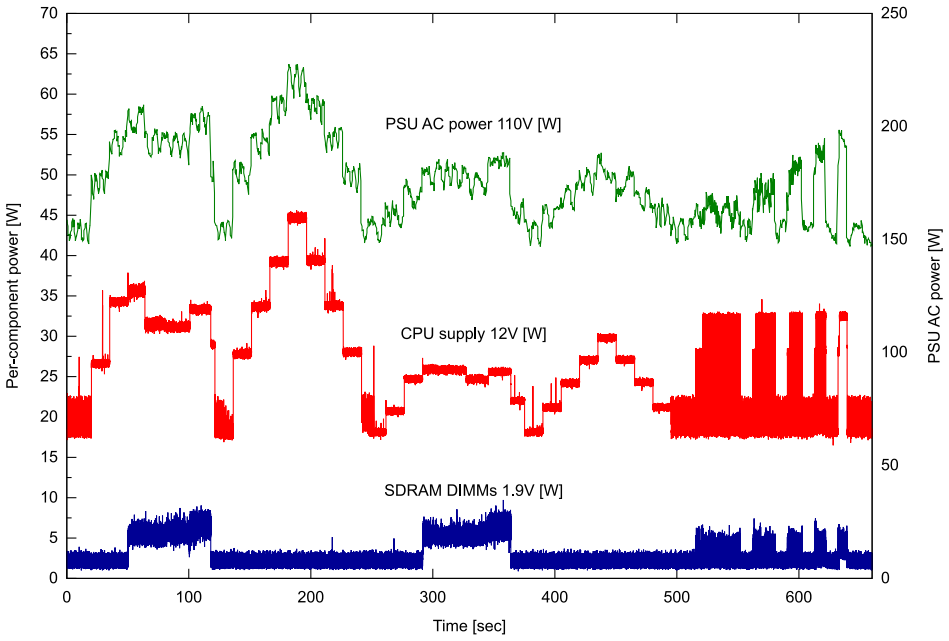

Figure 2.4: Correlation between component energy—CPU and SDRAM—and PSU AC power consumption.

SDRAM power. In conclusion, CPU and SDRAM load caused by running tasks not only have a high impact on per-component energy, but also on the whole machine's energy consumption. However, measuring CPU and SDRAM as opposed to AC power power consumption has many advantages. For instance, the additional per-component energy usage information can be used for scheduling or application profiling and energy optimization [24]. More importantly, as opposed to AC energy, which is too smoothed out to reflect power consumption changes at the rate at which tasks can be switched, CPU and SDRAM energy consumption can be attributed to the running processes. An example for this can be observed in the last phase of the experiment shown in Figure 2.4.

Chapter 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 Chapter 1 I 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]. I 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 3.1 presents an example of energy attribution in a single-core machine. For this example, I 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, I executed the test program only on one of the four CPU cores of the test machine—CPU1. In addition to power information on the CPU, SDRAM and motherboard, Figure 3.1 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, I 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 multi-

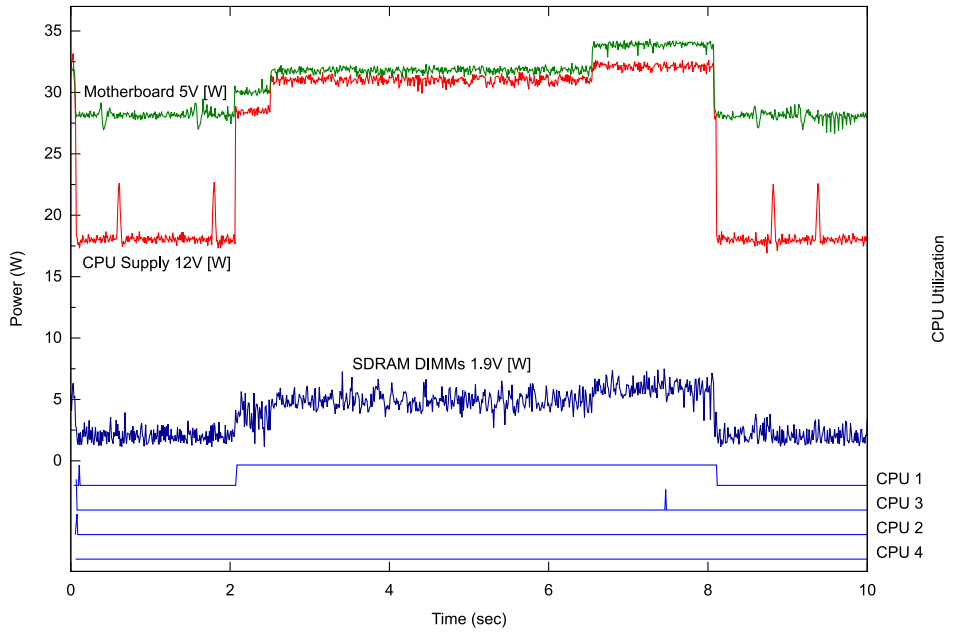

Figure 3.1: CPU, SDRAM and motherboard power over time for a single 512MB memory read & write test.

ple 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.

I 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.

The following example illustrates why such an approach is not always correct. I conducted three experiments using my 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 3.2 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.

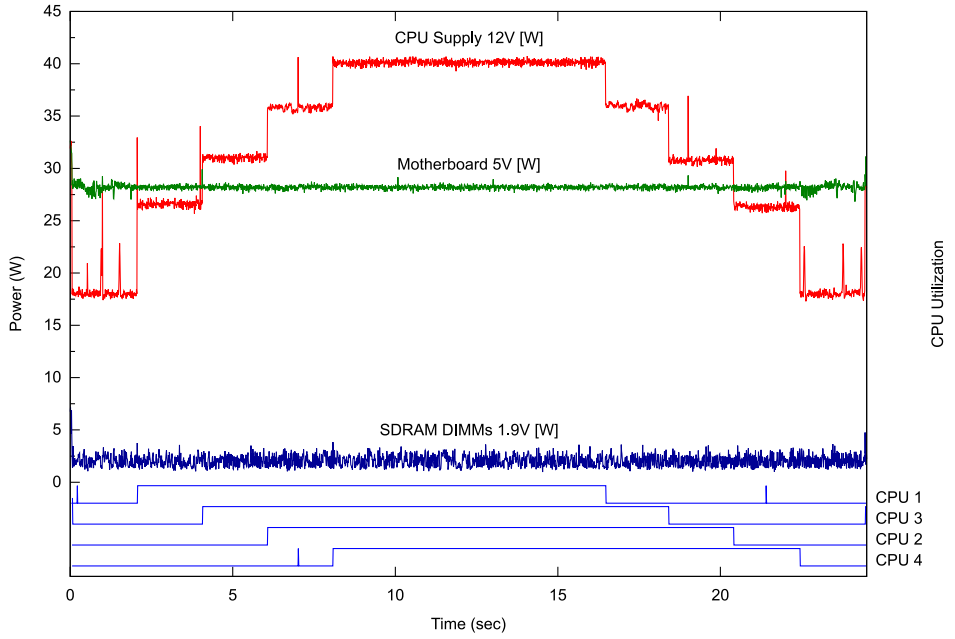

Figure 3.2: CPU, SDRAM and motherboard power over time for four 32KB memory read & write tests.

In the second experiment, shown in Figure 3.3 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 3.2)—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 3.4, 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.

The aforementioned examples showcase that a simple energy apportion solution that

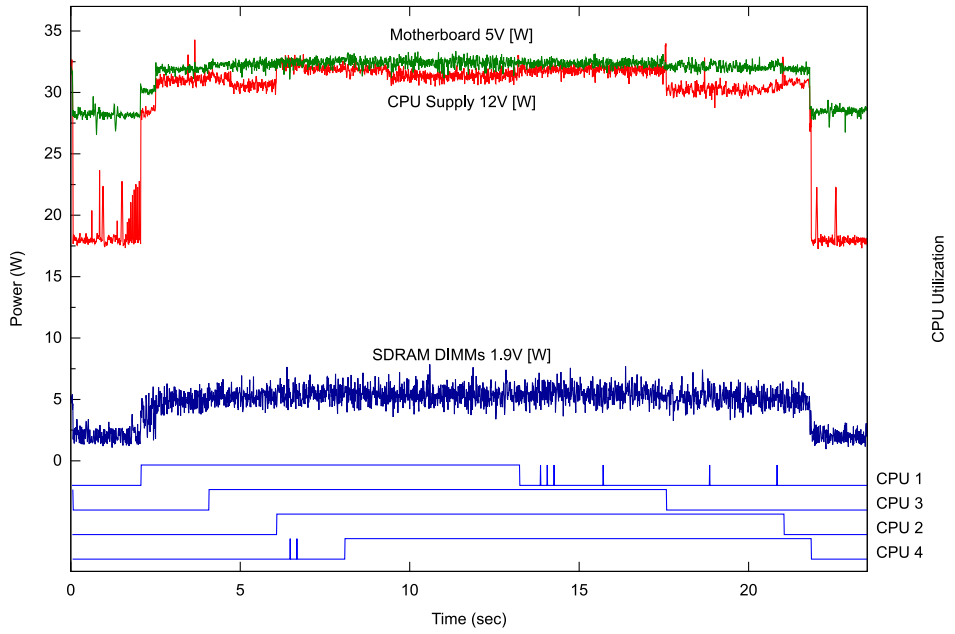

Figure 3.3: CPU, SDRAM and motherboard power over time for four 512MB memory read &write tests.

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 3.4 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 sophisticated indirect measurement methodology is needed.

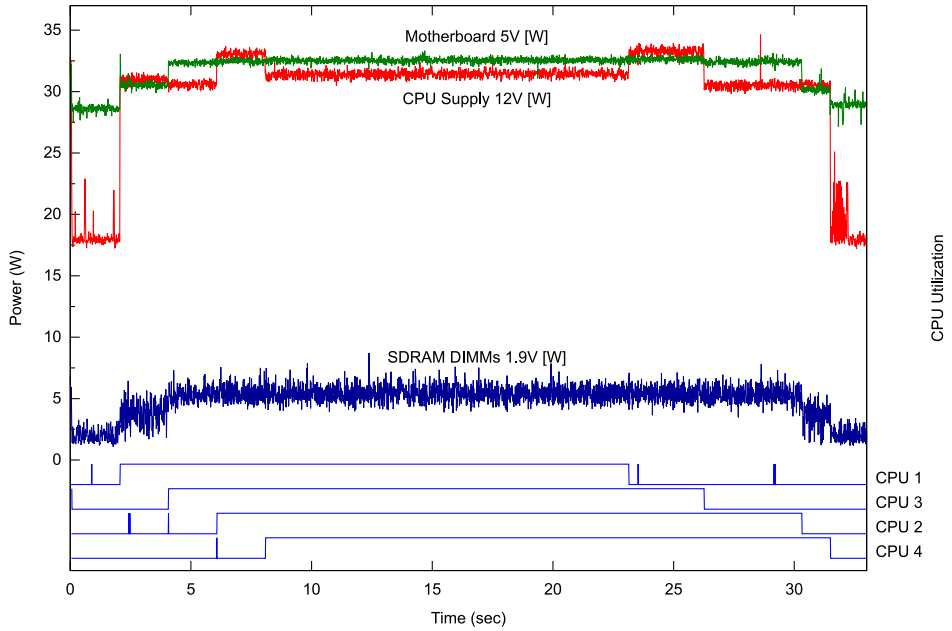

Figure 3.4: CPU, SDRAM and motherboard power over time for four 4MB memory read & write tests.

Chapter 4

Indirect Energy Measurement Model

In Chapter 3, I 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, I 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.

I 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 4.1.

$$E_{\text{cost},a} = \frac{(E_{\text{total}} - E_{\bar{a}})}{\sum_j \left(E_{\text{total}} - E_{\bar{j}}\right)} * E_{\text{total}} \tag{4.1}$$

As a consequence, a's energy cost depends on all other tasks executed concurrently, that might or might not be under a's control. I 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 Chapter 3). Also, application developers should be encouraged to write efficient code and not be rewarded for inefficient or suboptimal multi-threaded programming.

In this Section, I 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 4.2.

$$E_{\text{est}} = \sum_i c_i * e_i \tag{4.2}$$

I 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. Therefore, performance counters can measure per-core behavior, thus providing the additional per-core visibility that the direct measurement system lacks. Finally, previous work has shown that performance events can be related to energy consumption [5, 20, 17, 13], which makes them good *indicators* for per-core *energy* behavior. I note that, unlike Bellosa et al., I do not use performance counters to *estimate* total energy consumption as the RTDEMS measurement system provides direct and accurate measurements. Rather, after acquiring the total energy consumption through RTDEMS, I use performance counters to solve the *energy apportion problem*. I 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, the Intel® Core™ 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. I 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 4.1.

$$E_{\text{cost},a} = \frac{E_{\text{est},a}}{\sum_j E_{\text{est},j}} * E_{\text{measured}} \tag{4.3}$$

Equation 4.3 shows how I 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.

I defined E_{measured} as the energy cost *caused* by the running tasks. For that reason, I subtract the dc component from all energy and power measurements. I think, this is a *fair* policy, because it charges tasks only for the *additional energy consumption* they are directly responsible for.

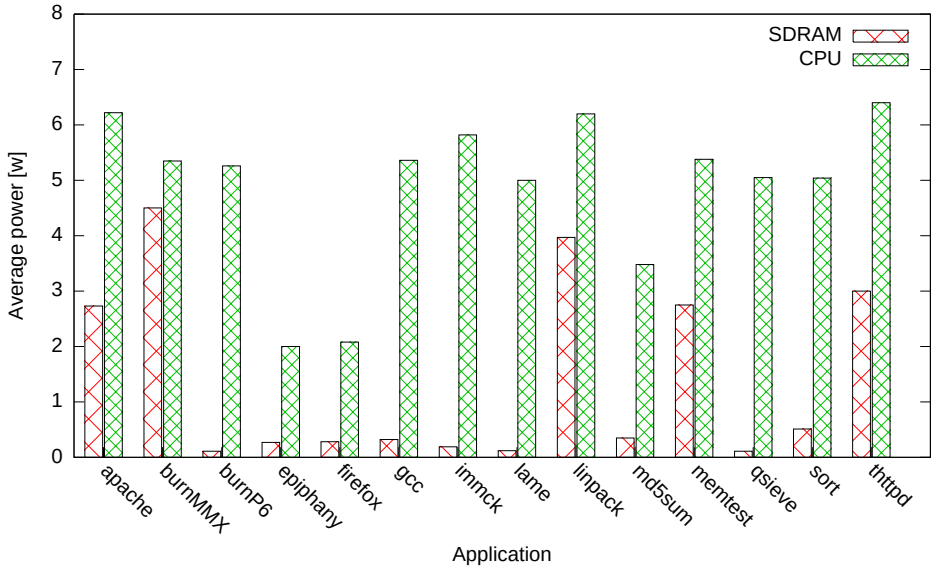

Figure 4.1: Average CPU and SDRAM power usage of different applications.

4.2 Model learning

In order to learn and test the models for CPU and SDRAM, I 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, memtest, my own memory access test capable of accessing memory in various ways (Chapter 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 and the boost library using gcc, the web browsers firefox and epiphany and the web servers apache and thttpd. As shown in Figures 4.1 and 4.2 these applications have very different characteristics, both in terms of power and in terms of performance events.

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™ 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, two events have to be selected that, together with the three predefined events can constitute a reasonable set of indirect measurement indicators for energy consumption (Equation 4.4).

$$E_{\text{est}} = \underbrace{\sum_{i=1}^{2} c_i * e_i}_{\text{choosable}} + \underbrace{\sum_{j=1}^{3} c_{fix,j} * e_{fix,j}}_{\text{predefined}} \tag{4.4}$$

One approach to this selection problem assumes, that energy consumption is best modeled by the *total model*, using all available counters as shown in Equation 4.5. Consequently, the event selection problem is solved, by choosing the two events

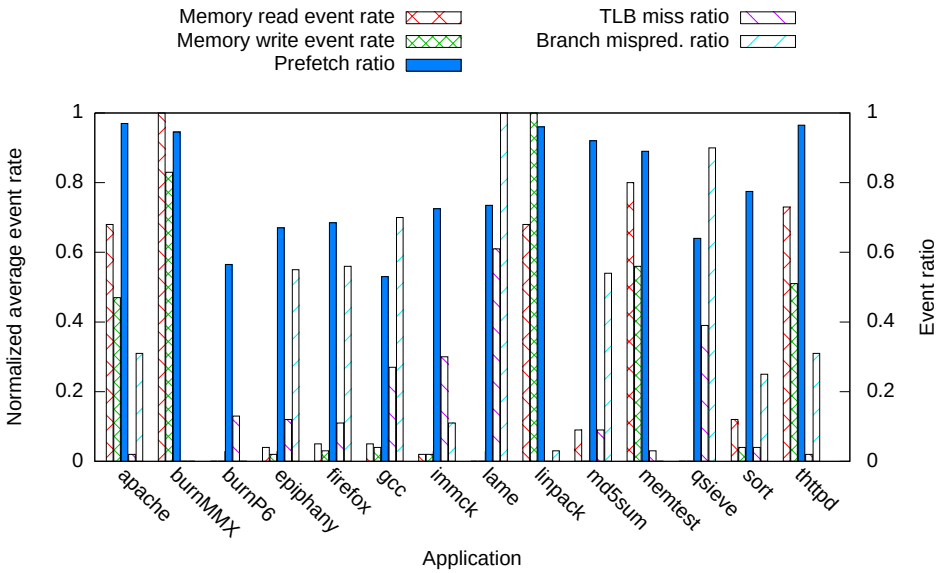

Figure 4.2: Normalized performance event rates and event ratios of different applications.

with the highest energy contribution $e_i * c_i$ to the total model, which results in Equation 4.6.

$$E_{\text{measured}} \approx \sum_{i=1}^{n} c_i * e_i \tag{4.5}$$

$$E_{\text{est}} = \underbrace{\sum_{i=1}^{5} c_i * e_i}_{\text{chosen counters}} + \underbrace{O\left(\sum_{i=6}^{n} c_i * e_i\right)}_{\text{error}} \tag{4.6}$$

The event energies e_i can be measured using micro benchmarks that cause a limited set of PMU events, thus facilitating the calculation of event energies for all possible events. However, this approach is not optimal as event energies are *not constant* but depend on both the running application and the model itself. 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. For these reasons, the energy contributions of events in the total model is not a good means for event selection.

My event selection methodology consists of the following steps. First, an expert prunes the search space of events that are not likely to have an impact on power consumption. For example, the event "transitions from floating point to MMX instructions" is not expected to occur often enough to have a measurable impact. Also, events like "load instructions retired" are not correlated to energy usage, because the load can result in a read from memory, L2, or L1 cache, which leads to funda-

Predefined events	
INSTRUCTIONS_RETIRED	Instructions retired.
UNHALTED_CORE_CYCLES	CPU cycles during which core was unhalted.
UNHALTED_REFERENCE_CYCLES	Front side bus cycles during which core was unhalted.
Selectable events	
L2_M_LINES_OUT:ANY	Modified lines evicted from the L2 cache. All of these lines are written to memory.
L2_M_LINES_OUT:PREFETCH	Only prefetched lines.
L2_M_LINES_OUT:SELF	Only not prefetched lines.
L2_LINES_IN:ANY	Lines read into the L2 cache. Due to the inclusive nature of the cache of the Intel® Core™ 2, all of these lines are read from main memory.
L2_LINES_IN:PREFETCH	Only prefetched lines.
L2_LINES_IN:SELF	Only not prefetched lines.
L2_M_LINES_IN:SELF	Modified lines written from the L1 into L2 cache.
DTLB_MISSES	TLB misses.
RESOURCE_STALLS	Cycles with a resource related stall.
MEMORY_DISAMBIGUATION:RESET	Number of times an instruction had to be re-executed due to a disambiguation error.
BUS_TRANS_MEM:SELF	Completed memory transactions.
MACRO_INSTS:DECODED	Instructions decoded, but not necessarily executed or retired.
MACRO_INSTS:CISC_DECODED	Complex instructions decoded.
UOPS_RETIRED:ANY	Micro-operations retired.
BR_INST_RETIRED:ANY	Branch instructions retired.
BR_INST_RETIRED_MISPRED	Mispredicted branch instructions.
CYCLES_DIV_BUSY	Cycles during which divider is busy.
X87_OPS_RETIRED:ANY	FPU floating-point operations retired.
SIMD_INSTR_RETIRED	Retired streaming Single Instruction Multiple Data (SIMD) instructions.

Table 4.1: Performance events considered for energy estimation. All of these events are counted on a per-core basis.

mentally different energy costs. Table 4.1 shows the performance events remaining after pruning.

Second, I gather model learning and test data. Because running a test multiple times always results in the same total event count, it is possible to run each test $\frac{1}{2}n$ times as required to measure the total counts for n events. As opposed to total values, using time series of event counts and energy provides more data points and also makes many counts linearly independent. For instance, for a program that reads and writes the same amount of data, the total counts for `Memory Reads` and `Memory Writes` are equal. This makes it impossible to attribute event energies using total counts, while it would be possible using time series. On the other hand, using time series each test has to be executed separately for each possible event combination, $\binom{n}{2}$ times in total. Because this is extremely time consuming, I decided to use total counts. To provide valuable model learning data tests have to be designed carefully.

Finally, I systematically build models using linear regression for SDRAM and CPU energy and compare their performances. For my system, I found through exhaustive search that the events `L2_LINES_IN:ANY`, `L2_M_LINES_OUT:ANY` and `INSTRUCTIONS_RETIRED` are well suited to model both SDRAM and CPU

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

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 (see Table 4.2).

For event energy learning I use time series. By performing linear regression on all of the learning data, I obtain a *generic model*. While learning application specific models, overlearning can become an issue if an application's counts for certain events are too low to have a measurable impact on energy, or if two counters are highly correlated. To avoid overlearning and ensure results that are generalizable beyond the learning data, I calculate the application specific event energies with a *weighted* least squares linear regression. By assigning higher weights to a specific application's learning data points, the generic model is shifted towards values that correspond to this application's event energies. Table 4.2 shows the event energies estimated with this method. I implemented a system, that fully automates the event energy learning step, which makes it easy to support new applications.

Figure 4.3 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.

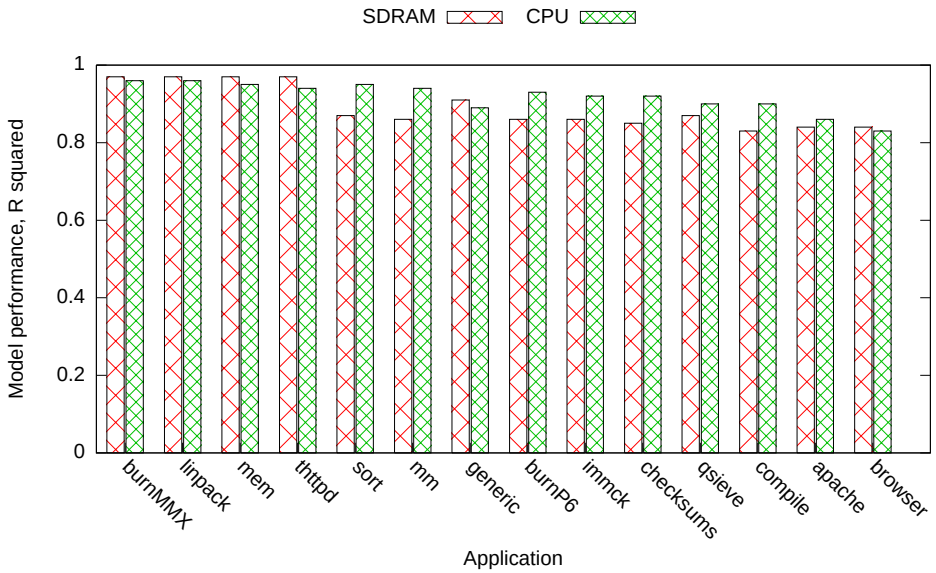

Figure 4.3: Single-core energy estimation performance.

Chapter 5

Architecture

The purpose of LEA^{2}P is to accurately account for the energy used by processes within a computer system. LEA^{2}P also provides runtime per-process energy usage information to the operating system and to user space programs. LEA^{2}P builds on the RTDEMS real-time energy measurement capability. Consequently, LEA^{2}P should meet the same requirements regarding resolution, cost, and integration stated in Chapter 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 the software architecture.

Parallelism. The software architecture must be optimized for multi-core operation.

Figure 5.1 shows an architectural diagram of LEA^{2}P. 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*, my 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 Chapter 4.

5.1 Resource containers

As energy accounting data structures I 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 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 my system. When an an accounting entity is active, its energy consumption is charged to the respective container. I utilize dynamic resource binding to allow binding of any resource container to a

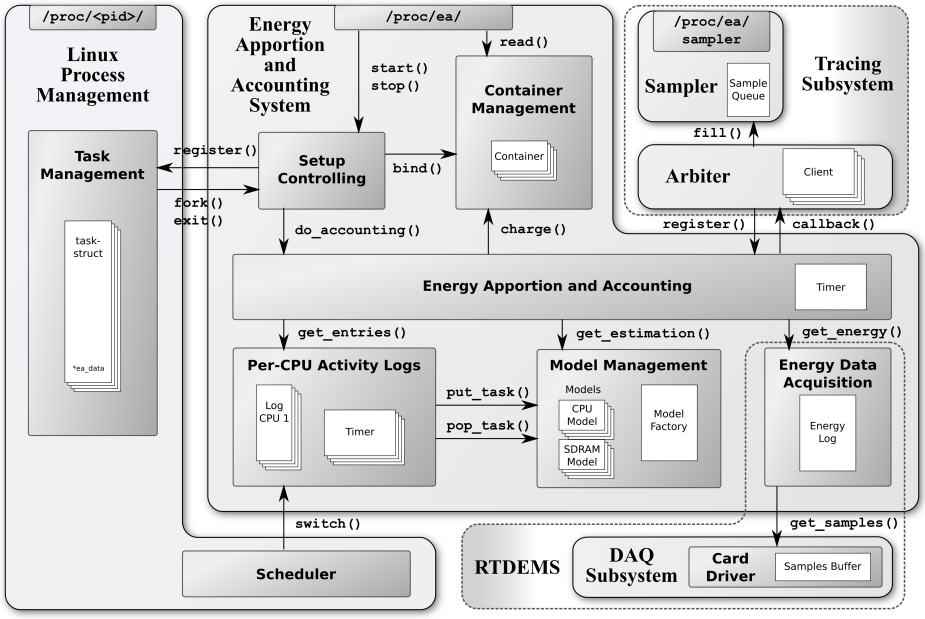

Figure 5.1: Architectural diagram of LEA²P.

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 introduced in Chapter 4 apportions energy among up to n tasks that run *concurrently*. For that reason, RTDEMS' energy measurement data has to be divided 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 to the tasks running concurrently on the CPUs, through the energy apportion algorithm. Figure 5.2 shows two CPUs running three tasks a, b, and c, their combined energy consumption E_{measured} as measured by RTDEMS and the resulting energy apportion time slices Δt_i. For Δt_1, used total energy e_1 is divided among tasks a and b, in Δt_2 e_2 is divided among a and c, in Δt_3 all energy e_3 is charged to c, and so on.

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

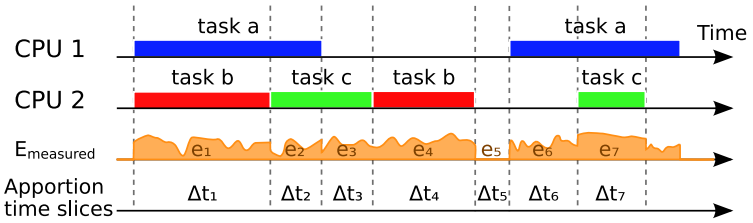

Figure 5.2: Energy accounting time slicing for three tasks a, b and c running on two CPUs.

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 my architecture depicted in Figure 5.1: 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, I 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.

My 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 task's energy behavior E_{est} needed by the apportion algorithm introduced in Chapter 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. Because the computation of E_{est} might be expensive, it is also deferred. Instead models utilize the activity log's memory regions to store model specific values to compute E_{est}. For example, the PMU model stores the event counts c_i.

Model management: LEA^{2}P 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 reconfigure the PMU to count different events that are better suited to determine the energy behavior of this particular 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, I 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 LEA^{2}P 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.

Besides augmenting the process data structure `task_struct`, I created an interface to Linux's process management that allows 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, I defined an interface, through which the energy accounting system can register callbacks for the following events:

`fork()` and `exit()` On the creation and termination of a task create or delete `ea_data`.

`exec()` When a task loads a new executable, reevaluate the task's models and possibly choose new application specific models. This is done using the model management's *model factory*.

`switch()` When the scheduler is about to switch to another task insert new entry into the per-CPU's 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: My 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. This thread processes the CPU's activity logs and charges the energy measured by RTDEMS to the resource containers using the application specific models.

5.3 Tracing subsystem

In addition to the energy accounting system, I 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 my 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 processed 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, I designed the tracing subsystem as optional kernel modules, which should only be activated when their functionality is needed. The sampler module exports the tracing interface and the samples to user space using the process file system.

Chapter 6

Implementation

This Chapter documents the most important features and trade-offs of my implementation of LEA^{2}P described in the previous Chapters. After explaining the event energy learning system, I describe the implementation of LEA^{2}P, and conclude this Chapter with a list of possible future improvements.

All of my source code is provided on the CD included with this thesis report. Table 6.1 shows the organization of the CD's content.

6.1 Behavioral model learning system

In this section I describe the implementation of the model learning system introduced in Chapter 4. I designed the standalone RTDEMS sampling module to facilitate training data acquisition for model learning. My behavioral model learning system consists of the steps event selection and event energy learning.

6.1.1 RTDEMS sampling module

The RTDEMS[1] sampling module is a data acquisition module that builds on RTDEMS' energy sampling capability described in Section 2.2. Figure 6.1 shows

[1] RTDEMS was originally called LEAP-Server. However, time did not allow to change the implementation to reflect the name change. Therefore, in this Chapter the two names are used interchangeably of each other.

Directory	*Content*
`software/`	Source code
`rtdems/`	RTDEMS system
`module/`	RTDEMS sampling module
`tools/`	Sampling and model learning tools
`tests/`	Application tests used for model learning
`lea2p/`	LEA^{2}P system
`module/`	Kernel modules and kernel patch
`tools/`	User space tools
`docs/`	Documents such as references and manuals
`doku/`	My documentation
`report/`	Thesis report
`usenix09/`	Paper submitted to USENIX'09

Table 6.1: Contents of the CD included with this thesis report.

the software architecture of the module. I implemented the module for model learning and other experiments where per-process accounting is not needed or practical. This standalone module is suitable for these cases, because in addition to energy, it contains a *system state sampler*, which can sample arbitrary per-CPU system variables such as CPU utilization and CPU frequency. More importantly, the system state sampler can measure arbitrary performance counters. Furthermore, the module offers a process filesystem interface that allows to control the sampling process and read the data from user space. The module's source code is located in the CD's `software/rtdems/module/` directory.

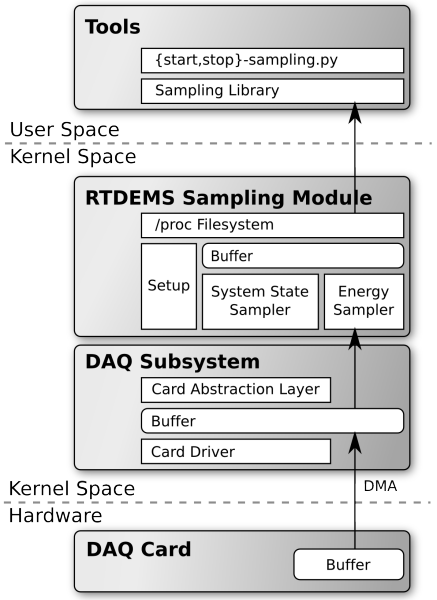

Figure 6.1: Software architecture of the RTDEMS sampling module.

Two user space tools using the RTDEMS sampling module are `start-sampling.py` and `stop-sampling.py`, which are implemented in python and can be found in the CD's `software/rtdems/tools/` directory. These applications use the sampling library `leapserver.py` to setup, start and stop the sampling process as well as to read, convert and export the values. `leapserver.py` interfaces to the sampling module using the `proc` files in `/proc/leapserver/`.

6.1.2 Event selection

The event selection process consists of two steps: training data acquisition and analysis. The implementation of the event selection method can be found in the CD's `software/rtdems/tools/tests/` directory.

To collect training data in a fully automated manner, I have written the `pmutotals.py` script. The script measures *total* performance event counts as well as energy values for a set of over 200 test cases containing all applications for which I want to find behavioral models. The script's arguments are the relevant performance event names, the number of repetitions of each test case, the CPU frequency and more. The test cases are executed in different combinations on different CPUs. To get total counts for all performance events, every test case is executed multiple

times each measuring two different events. This takes a long time. For example, to measure $2k = 20$ performance events for $n = 200$ test cases in $c = 2$ configurations with $r = 4$ repetitions $k*n*c*r = 16000$ tests have to be executed, thus even if all tests are only $30s$ long, this takes over five days. The script writes the the results to a file for further processing.

For analysis, the output of `pmutotals.py` can be processed using statistical software such as SPSS[2] or R[3]. I wrote the script `analyze-pmutotals.py`, which performs the analysis using R's linear regression and model analysis functionality. The script is capable of parsing `pmutotals.py`'s output and generates detailed statistics regarding the performance of the CPU and SDRAM models using a given set of performance events. Furthermore, this script calculates estimations for the per-applications event energies and other per-application statistics. This aids in comparing different models in order to select suitable performance events.

6.1.3 Event energy learning

After selecting events, the application specific event energies must be learned. While the event selection system provides event energies, I implemented a separate event energy learning system for the following reasons. First, the separate event energy learning system uses time series of event counts and energy. As explained in Section 4.2.2, this has many advantages and results in more accurate event energy estimations using fewer tests. Additionally, the event energy learning system does not require the RTDEMS sampling module, but uses the *tracing subsystem* of LEA^{2}P. Therefore, the event energy learning system allows to learn models for new applications on a *running* LEA^{2}P, thus constituting a first step towards online model learning.

Similar to event selection, the scripts `pmu-tests.py` and `pmu-learn.py` gather learning data and calculate event energies. These tools are located in the CD's `software/lea2p/tools` directory.

6.2 Energy apportion and accounting system

The design and implementation of LEA^{2}P are two of the main contributions of this thesis. This Section describes crucial parts of LEA^{2}P which are the data model, the per-process accounting algorithm as well as the interfaces offered to user space applications. I conclude the Section with a list of possible future implementation related improvements.

6.2.1 Data model

LEA^{2}P's design described in Chapter 5 defines the following crucial requirements for the data model: energy as well as per-CPU activity logs enabling deferred energy accounting, dynamic resource container and model bindings for tasks, and support for future extension to per-activity accounting. My data model is shown in Figure 6.5 on page 34, whereas arrows indicate possible pointer links. The following paragraphs describe key features of the data model and individual data structures like the energy log, per-CPU activity logs, task management, behavioral models and resource containers. Afterwards, I explain LEA^{2}P's time measurement and accounting granularity choices.

[2] SPSS: `http://www.spss.com/statistics/`
[3] R: `http://www.r-project.org/`

Energy log: I implemented the energy log as a ring buffer with a `head` and `tail` pointer as well as a `count` field. This data structure allows to concurrently insert samples at the head and read from the tail without the necessity of synchronizing access. Because LEA^{2}P's apportion algorithm reads the energy consumption for accounting time slices of arbitrary length, each of the energy log's entries stores the energy as well as the duration of the sampling period. This allows to partially read energy log entries by segmentation, where energy is divided up in proportion to the duration of the segments.

Per-CPU activity logs: Per-CPU activity logs are the core data structure that allow to defer energy apportion and accounting. I designed the activity logs similar to the energy log as ring buffers. In general, each entry represents a period, during which a task is scheduled uninterruptedly. Each activity log entry stores the following information needed in order to charge energy costs to that task: First, the length of the time slice represented by the entry, second, a pointer to the task's current `ea_data`, which binds the task to its current model, third, the `model_values[]`, that allows each of the models to store model specific values required to compute E_{est}, and finally, the task's current resource container to which the energy consumption of the task is charged. In a per-activity accounting system, a task's resource container binding changes frequently. Therefore, I preserve the correct resource container binding in the activity log. For example, in Figure 6.5 the entries 2 and 4 of the nth CPU's activity log point to the same task, even the same `ea_data`, but not necessarily to the same resource container. Alternatively, the resource container could be referenced using `ea_data`, but then a new `ea_data` would have to be created whenever the resource container binding of a task changes.

Task management: Dynamic resource and model binding of tasks is implemented by the `ea_data` structure. Dynamic allocation of `ea_data` 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 is freed.

Behavioral models: Models are represented by both their interface consisting of the five functions described in Section 5.2, and a model specific configuration. For example, PMU models for different applications are implemented by the same model using different event energies e_i stored in different model configurations. This facilitates generating models for new application by simply allocating a new model configuration.

Resource containers: Besides merely storing energy, resource containers implement a few more features. All operations on resource containers are atomic, which allows them to be *shared* by multiple processes. For instance, all idle processes (one per core) are associated with the same idle container. I use `p_refs` as a *reference counter* to indicate how many processes are associated with a container. As a consequence of deferring energy accounting, the resource container values 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 in `wl_refs`, that are not yet accounted for. If the count is zero, a resource container's energy values are up-to-date. As soon as both the process as well as the activity log reference counters are zero, the container is not in use anymore and can be deleted. Additionally, every resource container has an unique id, that allows to identify a particular container.

Time: LEA²P uses three different measures of time, which are: *a)* The operating system's time as returned by `gettimeofday()`, *b)* the DAQ card's internal oscillator from which the energy sampling frequency is derived, and *c)* the CPU's time stamp counter (tsc) used in the activity logs. For accounting, I have decided to use the tsc for time measurements, because of its nanosecond resolution and stability among the CPU cores. It is crucial, that these three time measures are always *synchronized*. For example, if the DAQ card's time and the per-CPU activity log's time drift apart, samples read from the energy log are not actually from the same time as the entries read from the activity logs. Therefore, the energy is attributed to the wrong processes.

The most important means to prevent time drift is to make sure, that no entries in the energy and the activity logs are lost. Consequently, it is paramount that the system runs uninterruptedly and the log queues never become full, such that new entries can be added at all times. I assume that both the CPU's time stamp counter as well as the DAQ card's oscillator increase at a steady rate, such that the time sync problem can be solved by knowing the respective rates very exactly in order to allow a perfect conversion. By further choosing the DAQ card's sampling period as an integer multiple of the tsc period, the length of an energy log entry can perfectly be expressed in tsc cycles. My experiments confirm, that these measures work well for runtimes of a couple of minutes. In order to guarantee time sync for hours or even days, an active time synchronization system, such as the one I describe in Section 6.3, must be implemented.

Accounting granularity: For performance reasons, floating point registers are not saved on context switches within the kernel, thus making it unsafe to use floating point operations in kernel space. Therefore, all of LEA²P's calculations use integer arithmetic and require to carefully choose the granularity of the units. On one hand, the accounting unit's granularity must be fine enough, to accurately represent small energy values, for example for very small accounting time slices. On the other hand, large values for example energy accumulated over a long time, or intermediate calculation results, must fit into 64bit variables. Because energy values are stored in signed integer variables, their absolute value should not become bigger than $2^{63}-1$.

I have identified the smallest physical values used by LEA²P, which are the pmu model's event costs e_i. As Table 4.2 shows, event costs are of the order of a few nanojoules and should preferably be represented with pico-joule granularity. LEA²P's largest numbers occur within RTDEMS' energy calculation as in Equation 6.1 and the energy apportion algorithm as in Equation 6.2. Variables with a tilde denote an entity in accounting units and u represents one physical unit expressed in accounting units, such as 10^6 for micro-joule accounting granularity. Therefore, $\widetilde{x} = x * u$, for example using micro-joule granularity $x = 10.5\text{mJ}$ is represented by $\widetilde{x} = 10,500$.

$$\widetilde{E} = \frac{\overbrace{\widetilde{v} * \widetilde{i}}^{<(2^{63}-1)}}{u} * \Delta t \quad \Longrightarrow \quad \log_2 u < 31 - \frac{1}{2}\log_2 \max(P) \tag{6.1}$$

$$\widetilde{E}_{\text{cost},i} = \frac{\overbrace{\widetilde{E}_{\text{est},i} * \widetilde{E}_{\text{measured}}}^{<(2^{63}-1)}}{\sum_j \widetilde{E}_{\text{est},j}} \quad \Longrightarrow \quad \log_2 u < 31 - \log_2 \max(E) \tag{6.2}$$

Since the accounting time slice and therefore the energies used by the apportion algorithm are typically very small, Equation 6.1 is more critical. As a result, I have chosen 10^{-8} as my accounting unit, which allows to measure power consumptions

of up to 128 watts and apportion energies of up to 11 joules per accounting time slice. These constraints force me, to use different units within the pmu model's calculations, where I use pico-joules and ultimately convert the result to accounting units.

6.2.2 Per-process accounting

In order to apportion and account energy, my system logs per-CPU activity and energy measurements. In a second step, the logs are processed and energy is charged. The following sections describe both steps and the algorithms used in more detail.

Per-CPU activity logging

Each CPU inserts entries in its own activity log, which are later used by the accounting thread to charge energy costs to tasks. Whenever the scheduler selects a new task, the LEA^{2}P is informed and inserts a new entry in the CPU's activity log. For example, Figure 6.2a) shows two CPUs running three tasks a, b, and c. Task a is scheduled on CPU 1 at $t = 0$ms, and a new activity log entry is created. For each of a's models, the `put_task()` function is called to gives the model the opportunity to prepare to monitor a's energy behavior. Later at $t = 11.4$ms a is removed from the CPU, the task's model's `pop_task()` functions are called and the activity log entry is finalized. In this example in Figure 6.2b), the models store the values `[100, 200]` and `[200, 100]`, respectively. At this point, the entry contains all information needed by the energy accounting thread. CPU 2 performs the same actions for tasks b and c.

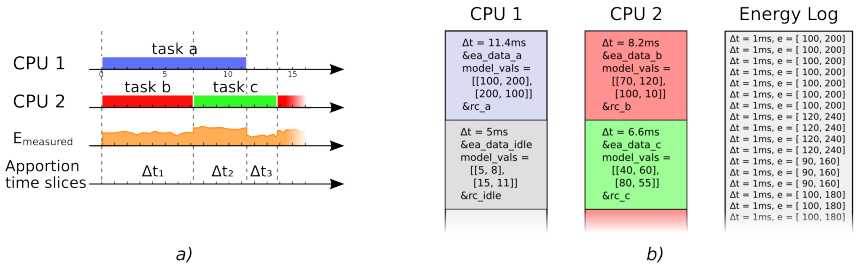

Figure 6.2: Per-CPU activity logging on two CPUs running the tasks a, b, and c. *a)* Scheduling of the tasks. *b)* Corresponding activity and energy log entries.

Apportion and accounting algorithm

The apportion and accounting algorithm works as follows:

1. Get entries for the next accounting time slice Δt from all per-CPU activity logs.
2. For each resource (CPU, SDRAM)
   (a) Get E_{measured} for Δt from 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.

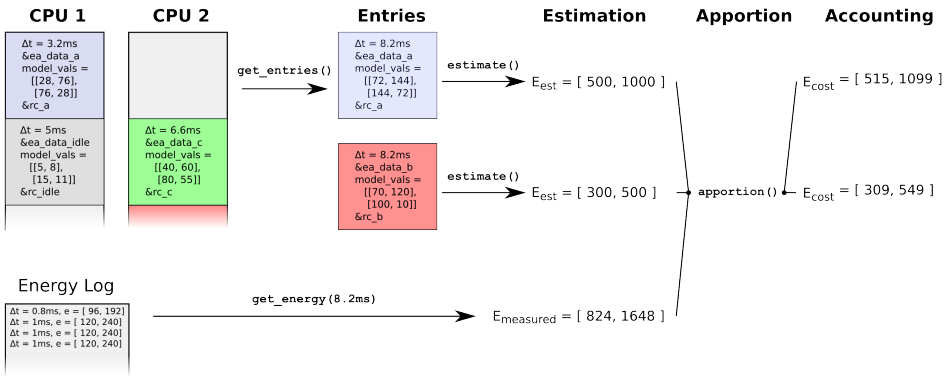

Figure 6.3: Steps taken by the activity log processing algorithm in order to account for the schedule of tasks shown in Figure 6.2.

(c) Charge each entry's resource container according to the apportion rule introduced in Chapter 3.

For example, Figure 6.3 shows the steps necessary to process the energy and activity log entries created by the schedule of tasks shown in Figure 6.2.

First, the entries of the activity logs are read. Since all CPUs create activity log entries independently, the log entries do not correspond to the same time slice in general. However, to apportion the energy of a time slice Δt, the entries have to be of the *same length*. The length of the accounting time slice is defined in step 1 by the shortest entry, which is in my example task b's and 8.2ms long. Longer entries are segmented into an entry of length Δt and the remainder. In this example, the 11.4ms entry of CPU 1 is segmented into a parts of 8.2ms and 3.2ms. All model values are divided up in proportion to the length of the segments. For example, when splitting up the value 100 the 8.2ms segment gets $100 * 8.2/11.4 = 77$ and the second 3.2ms segment remaining in the activity log gets 28.

Second, the energy E_{measured} used during those 8.2ms is retrieved from RTDEMS' energy log. In my example this is 824 for SDRAM and 1648 for CPU energy.

Third, each of the task's behavioral energy models is called to provide an energy estimation for the log entry:

```
sdram_est = entry->ea_data->sdram_model->estimate(entry)
```

For this estimation, the model relies on the model values it saved in the `put_task()` and `pop_task()` calls.

Forth, E_{measured} is apportioned according to my fair apportion system introduced in Chapter 4 and finally, this energy is charged to the task's resource container referenced in the activity log entry.

I assume that the behavior of a task is approximately *uniform* during an activity log time slice, such that when an entry is segmented, model values can be divided up linearly among the segments. The following example shows a case, where an activity log entry with non-uniform behavior leads to wrong apportion of energy. Assume that task a and task c shown in Figure 6.4 perform memory accesses, but b does not. Furthermore, assume that a only accesses memory, when b but not when c is active. In order to account SDRAM energy e_1 of Δt_1, a's activity log entry is segmented, and the model values are divided up proportionally among the segments, even though a did not access memory during Δt_2. Because b did not access memory during Δt_1, all of e_1 is charged to a, which is correct. However, when the system apportions e_2 for Δt_2, a's model values for Δt_t indicate wrongly, that a accessed

File	*Functionality*
`state`	Displays LEA^{2}P's current state and allows to start and stop the system.
`cont`	Allows to read the values of a particular resource container, which can be selected by writing the id to the file prior to reading.
`conts`	Lists the values of all resource containers.
`stats`	Provides information such as queue fill statistics, overhead caused by the system, current sampling rates and more.

Table 6.2: LEA^{2}P `proc` interface files in `/proc/ea/`.

memory. Therefore, e_2 is split among a and c, which is obviously wrong because only c accessed memory during Δt_2. In conclusion, in order to segment activity log entries the behavior of a task must be uniform during the entries time slice. 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 of uniform behavior is reasonable—20Hz in my implementation.

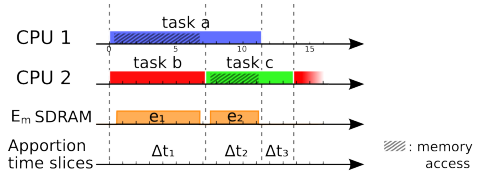

Figure 6.4: Application example, where an activity log entry of non-uniform behavior leads to a wrong apportion result.

6.2.3 User space interfaces

LEA^{2}P provides a number of different interfaces, that enable user space applications to control and monitor the system as well as acquire runtime energy information. Along with three `proc` filesystem interfaces for LEA^{2}P controlling, per-process energy information and for the tracing subsystem, I implemented a set of python libraries that facilitate interaction with the interfaces.

LEA^{2}P controlling and monitoring

The LEA^{2}P controlling and monitoring interface is located in the `/proc/ea/` directory. The functionality of the files is described in Table 6.2. The library interfacing to these files is called `ea.py` and included in the CD's `software/lea2p/tools/` directory.

Per-process energy information

Similar to other per-process information offered by Linux, LEA^{2}P adds a file to each process' `proc` directory showing the process' energy usage in `/proc/<pid>/ea`. A process has to have sufficient privileges in order to access this file. An application like the well-known `top`, which provides a dynamic runtime view of tasks running on a system, could easily be extended to display this per-process energy information [24].

File	*Functionality*
`state`	Displays the current state and allows to start and stop tracing.
`ctrl`	Allows to choose the values which are traced, for instance E_{measured}, E_{est}, per-CPU energy consumption, model values and more.
`cont`	Allows to choose the resource container that is traced. Choose -1 to trace the whole system.
`samples`	Provides the tracing results.
`power`	Displays the whole system's current power consumption.

Table 6.3: Tracing subsystem `proc` interface files in `/proc/ea/sampler/`.

Tracing subsystem

The tracing subsystem's interface is implemented using files in `/proc/ea/sampler/`. Table 6.3 describes the interface. However, I recommend to use the library `ea.py` from the CD's `software/lea2p/tools/` directory for sampling. An example of how this can be implemented can be seen in `trace-app.py`.

6.3 Future improvements

My implementation described in this Chapter is complete and working. Now that all components are finished and successfully meet the requirements specified in Chapter 5, the following five improvements could be implemented.

First, as mentioned in Section 6.2 the system's time sources are not actively synchronized. Because all time sources advance at a reasonably steady rate, this is sufficient for short runtimes of LEA^{2}P. However, for runtimes in the order of hours or even days an active synchronization is necessary. This can be implemented by generating a pulse at a known frequency on a system output that is connected to and measured by the DAQ card. By correlating the DAQ card's measurements with the known times, at which the pulses were sent, time drift can be detected and corrected.

Second, the implementation of application specific models must be finished by implementing the missing `exec()` callback. Furthermore, alternative behavioral models could be studied. For example CPU frequency aware or independent models, non linear models or also models that take the model values of all CPUs into account.

Third, per-user or also per-process accounting, where a process' resource container accumulates values from all threads and child processes, could be implemented. For this, a resource container hierarchy must be introduced, where the energy costs of a container are also charged to its parent container. This could provide valuable energy usage information.

Fourth, the addition of per-activity accounting would be useful for some applications such as web servers [4, 25]. This would not require changes on LEA^{2}P, but extensive modifications of the kernel as well as user space applications.

Finally, energy-aware user space applications such as `etop` could be implemented.

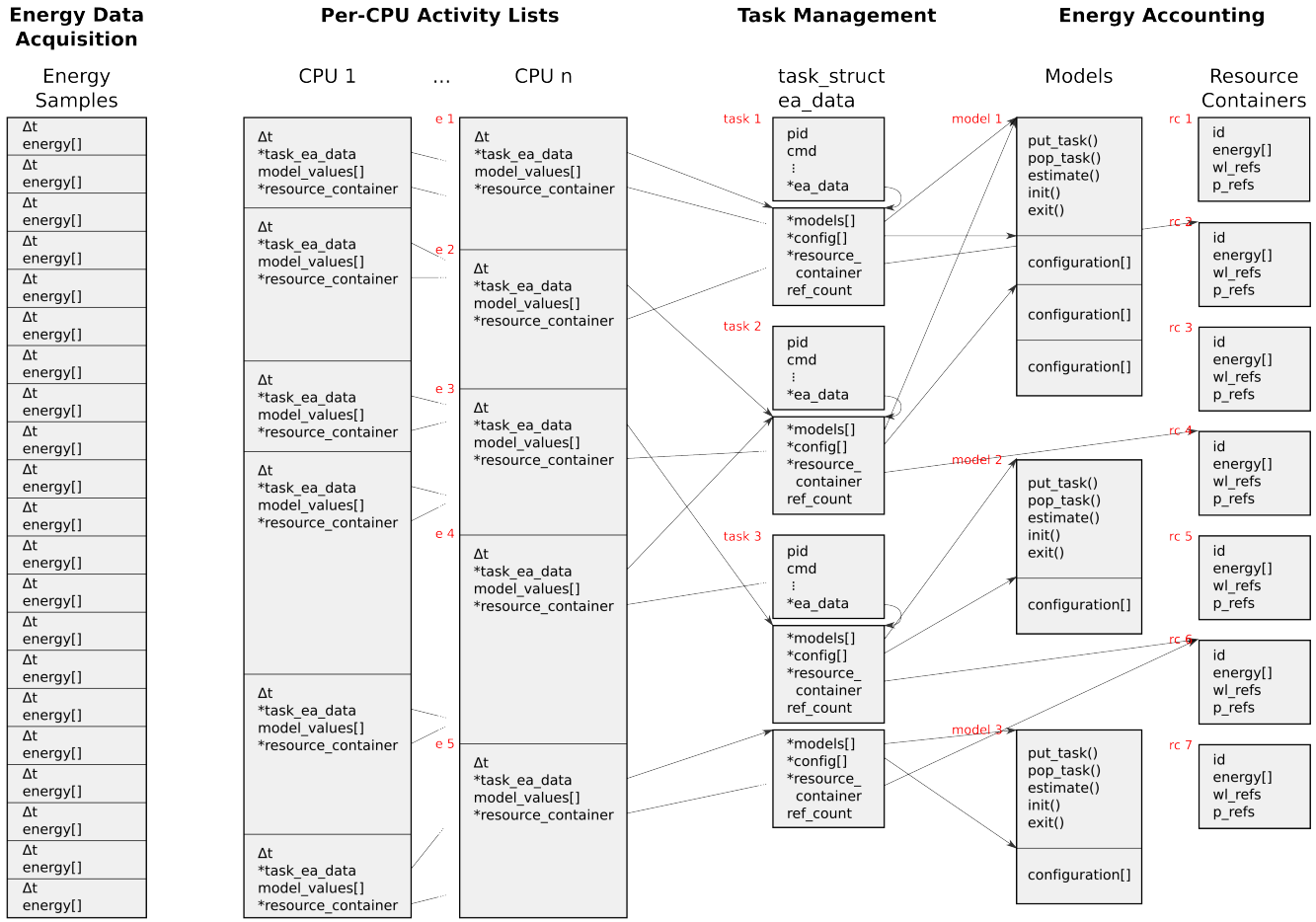

Figure 6.5: Data model of the energy accounting software system.

Chapter 7

Evaluation

In this section, I present an experimental evaluation of the accuracy of the apportion algorithm and demonstrate the functionality of LEA^{2}P using typical desktop applications. In addition, I investigate the impact of my system on CPU resources.

7.1 Per-process energy apportion accuracy

The main part of the energy apportion algorithm is the set of performance event models introduced in Chapter 4. In order to ascertain the accuracy of these models, I would need to compare the results to those obtained by an a priori accurate measurement system. However, as mentioned in Chapter 3, without extensive motherboard and CPU modifications, I 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 I therefore designed experiments for which I am able to assert a particular apportion. I then compare the asserted values with the solution found by my online algorithm. I chose a sequential memory access benchmark as my test program, as it allows to test SDRAM as well as CPU energy apportioning, by controlling the number of memory accesses. I chose a memory buffer of 512MB in order to minimize the impact of the CPU's cache management, which is beyond my 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, I 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, I assert that the correct memory energy attribution would be 33% for process A and 66% for process B. I note that this assertion holds for my 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 7.1 depicts the result of my 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 LEA^{2}P's energy apportioning system. An ideal energy apportioning system would have a y-axis value that would match the corresponding x-axis value. SDRAM (utilization) is the CPU utilization based apportion result for

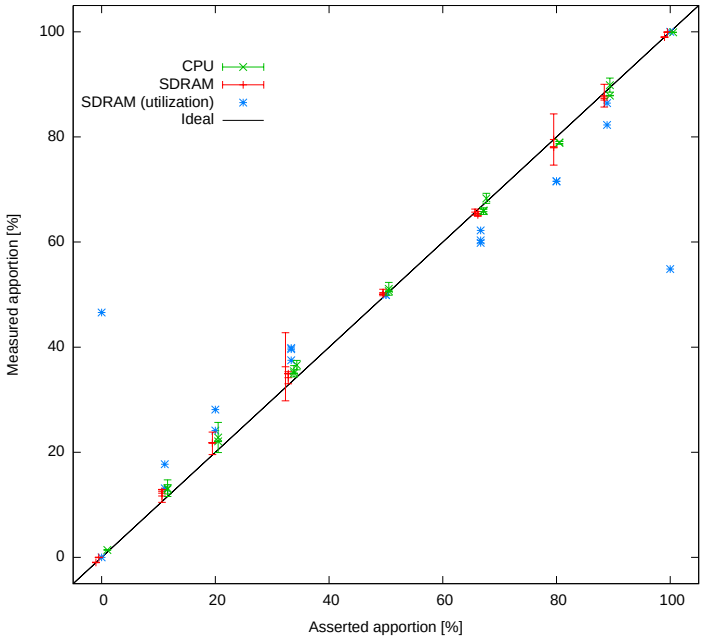

Figure 7.1: Asserted and measured CPU and SDRAM energy apportion of two tasks a and b with 95% confidence interval. "SDRAM (utilization)" is the CPU utilization based apportion result for SDRAM energy.

SDRAM energy. In all cases the utilization based apportion performs worse than LEA^{2}P, especially for the cases where one test does not access memory by using the CPU cache exclusively. As seen in Figure 7.1, my 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 my system provides the correct apportion for arbitrary applications.

7.2 CPU overhead

As mentioned in Chapter 2 and 5, LEA^{2}P needs to operate with the lowest possible overhead. In Chapter 2 I investigated the RTDEMS overhead in terms of CPU resources. However RTDEMS-induced overhead is only part of LEA^{2}P'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 my system on CPU resources, I measured the CPU time spent within LEA^{2}P. 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, I 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.

I conducted experiments using two tasks per CPU, variable task activity periods, and by enabling and disabling the tracing subsystem. Figure 7.2 shows LEA^{2}P's CPU overhead. I 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 my design decision to implement the tracing subsystem as an optional module.

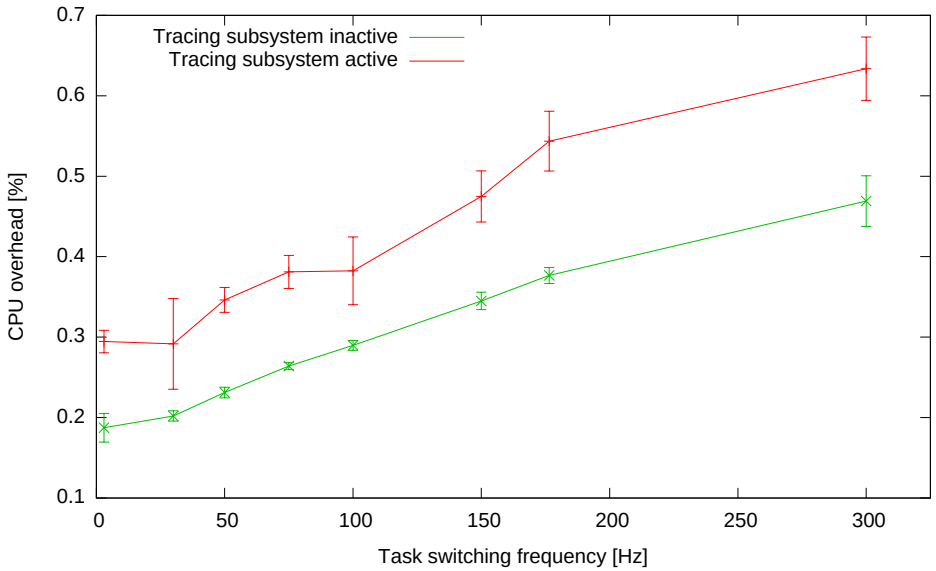

Figure 7.2: CPU overhead with 95% confidence intervals of LEA^{2}P as a function of task switching frequency.

I argue, that LEA^{2}P's overhead depends not only on the scheduling activity, but also on the number or CPUs. Adding a CPU leads to the creation of an additional per-CPU activity log. Not only does the additional CPU have to add entries to the log, the entries also have to be processed. An additional CPU further results in a more complex apportion calculation, because energy has to be apportioned among more CPU entries. Also, since scheduling activity on the added CPU leads to additional segmentation of the activity log entries so that the average accounting time slice becomes smaller, thus resulting in more apportion steps which increases overhead.

It is important to note, that the overhead does not depend on the number of tasks, unless the number of tasks influences the scheduling activity. The overhead to apportion and account for two tasks switching back an forth is the same as for many tasks switching at the same frequency, because in both cases exactly the same amount of activity log entries are created and processed in exactly the same way.

Application	*CPU [J]*	*SDRAM [J]*	*Runtime [sec]*
apache	158.3	1.9	17.5
gcc	159.4	4.0	17.9
sort, 128MB	174.0	5.1	19.2
image blur	96.8	0.5	10.1

Table 7.1: CPU and SDRAM energy apportion for four applications executed sequentially.

Application	*CPU [J]*	*SDRAM [J]*	*Runtime [sec]*
apache	83.9	1.98	18.7
gcc	109.7	5.19	19.6
sort, 128MB	145.92	5.84	20.2
image blur	72.65	0.43	10.3

Table 7.2: CPU and SDRAM energy apportion of the same four applications executed concurrently.

7.3 Application apportioning

Tables 7.1 and 7.2 show the results of the apportion system for four applications executed sequentially and simultaneously, respectively. My 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. Figure 7.3 shows the applications' individual CPU and SDRAM power profiles measured using LEA^{2}P's energy tracing capability. In contrast, Figure 7.4 shows the combined energy consumption when executing all tests concurrently. In addition, the latter Figure depicts the energy apportion as calculated by LEA^{2}P. The application's characteristic energy footprints are also reflected by the apportion result. While each application's runtime increases slightly when executed concurrently with the other applications, total CPU energy consumption decreases, thus all applications are charged less CPU energy. On the other hand, gcc's and sort's SDRAM energy consumption increases. This indicates, that some of their cache lines are evicted by image blur and apache, respectively.

My experimental results demonstrate that LEA^{2}P has achieved its design goals of providing integrated and accurate—96% of optimal—per-process energy accounting of individual hardware components, incurring only up to 0.6% of CPU overhead.

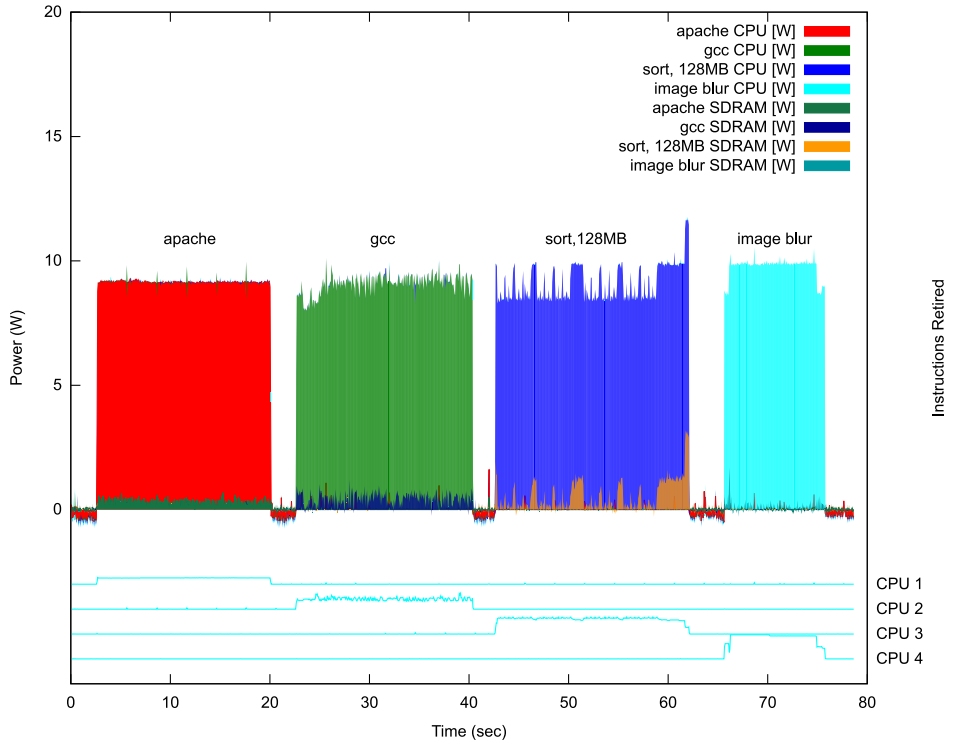

Figure 7.3: Energy consumption of apache, gcc, sort, and image blur executed sequentially.

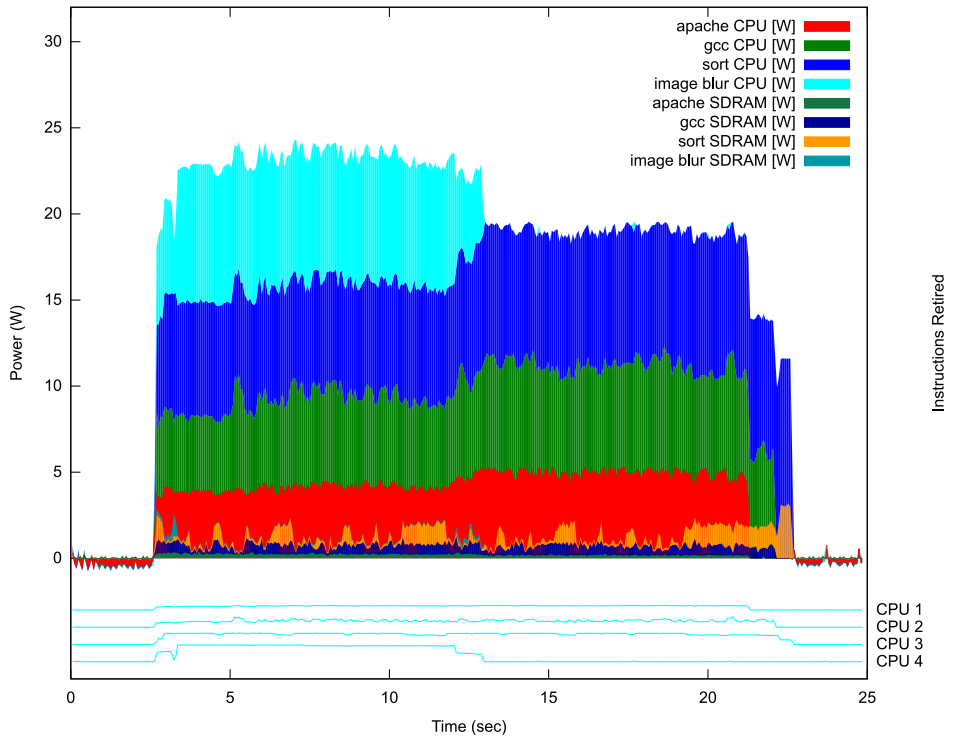

Figure 7.4: Combined energy consumption of apache, gcc, sort, and image blur executed concurrently.

Chapter 8

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 Chapter 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. My work does not rely on estimation techniques, as the RTDEMS system provides 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. My energy behavior models and methods of energy apportioning among tasks are based on this work. LEA^{2}P extends 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. My system builds upon previous work by providing the first implementation of resource containers for multi-core systems and by solving the energy apportion problem.

My work is an adaption of previous work—LEAP2 [24]—for embedded computing. LEAP2 combined real-time energy information measured and accumulated by a dedicated ASIC, and per-process energy accounting. Since this embedded platform

has only one CPU, LEAP2 does neither require indirect energy measurements nor energy apportion.

Chapter 9

Conclusion

This thesis introduces LEA^{2}P, a new energy attribution software architecture that augments the operating system of a *multi-core platform* with runtime per-process energy usage information. LEA^{2}P utilizes runtime *direct* energy measurements that provide accurate per-component energy usage information at millisecond-scale resolution. I 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 I introduce performance counter based energy behavior models. I experimentally demonstrate that my 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.

I fuse the direct and indirect portions of my system in a combined energy apportion and accounting software system—LEA^{2}P—, designed as a low-overhead modular component of the Linux operating system. My experiments demonstrate that my 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 I plan to extend LEA^{2}P 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, I aim to replace the initial calibration phase necessary for model learning with an online model learning system. I designed LEA^{2}P with future expansion to alternative accounting systems like per-activity accounting in mind. I also intend to build on my resource container implementation and provide a more powerful interface for container manipulation to user space applications.

Acknowledgments

I wish to thank Thanos Stathopoulos and Dustin McIntire from UCLA for their invaluable support and advice. I am also very grateful to William J. Kaiser from UCLA and Lothar Thiele from ETH for their guidance and approval that made this thesis possible. Furthermore, I am thankful to all members of the ASCENT Lab for their friendship and helpfulness and to UCLA's Electrical Engineering Department for always providing apt administrative assistance.

Bibliography

[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.