Per-Task Energy Metering and Accounting in the Multicore Era

Qixiao Liu

Advisor
Miquel Moretó Planas
Co-Advisors
Jaume Abella Ferrer
Francisco J. Cazorla Almeida
Mateo Valero Cortés

Memòria del Projecte de Tesi
Programa de Doctorat d'Arquitectura i Tecnologia de Computadors
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

March 2016

Abstract

Chip multi-core processors (CMPs) are the preferred processing platform across different domains such as data centers, real-time systems and mobile devices. In all those domains, energy is arguably the most expensive resource in a computing system, in particular with fastest growth. Therefore, measuring the energy usage draws vast attention. Current studies mostly focus on obtaining finer-granularity energy measurement, such as measuring power in smaller time intervals, distributing energy to hardware components or software components. Such studies focus on scenarios where system energy is measured, and under the assumption that only one program is running in the system. So far, there is no hardware-level mechanism proposed to distribute the system energy to multiple running programs in a resource sharing multi-core system in an exact way.

In this thesis, for the first time, we formalize the need for per-task energy measurement in multicore by establishing a two-fold concept: per-task energy metering and sensible energy accounting. The former, for a task running in a multi-core system, provides estimates on the actual energy consumption corresponding to its resource usage. The latter provides estimates on the energy the task would have consumed running in isolation with a given fraction of the shared resources.

Accurately determining the energy consumed by each task in a system will become of prominent importance in future multi-core based systems as it offers several benefits including (i) better application energy/performance optimizations, (ii) improved energy-aware task scheduling and (iii) energy-aware billing in data centers.

We have shown how these two concepts can be applied to the main components of a computing system: the processor and the memory system. In each, we have proposed models to ideally meter and account the energy. And by trading off the hardware cost with the estimation accuracy, we have obtained implementable and affordable mechanisms with high accuracy. We have also shown how these techniques can be applied in different scenarios, such as, to detect significant energy usage variations for any particular task and to develop more energy efficient scheduling policy for the multi-core system.

Acknowledgment

First and foremost, I want to thank my advisers, Dr. Miquel Moretó, Dr. Jaume Abella, Dr. Francisco J. Cazorla and Professor Mateo Valero. They have taught me how high-quality research in the computer architecture field is done. It has been a great honor to be supervised by them and I sincerely appreciate their contributions of time, ideas and guidance to make my Ph.D experience stimulating and enjoyable. Their joys and enthusiasms for the research are contagious and motivational for me, especially during the tough times of the study. They have also provided me with excellent examples as successful researchers and mentors.

The members of the Computer Architecture and Operating System interface (CAOS) group have contributed immensely to my research and life here at Barcelona. The group has been a source of friendships as well as good advices. I'm especially grateful for Dr. Carlos Luque, whose work is one of the cornerstone of my thesis: the performance accounting. He vigorously shared his visions and experiences with me, and helped me to get over difficulties. I would like to thank Dr. Petar Radojkovic, from whom I have learned the successful and encouraging researching experiences.

I want to thank Pradip Bose, Alper Buyuktosunoglu, Ramon Bertran and Victor Jimenez from IBM T.J Watson in New York, where I have spent five months for an internship, for their immense help in the research and life. It has been incentive to work with them with joy. They have taught me numerous knowledges and experiences on research with excellent mentoring skills. I am also grateful for Professor Daniel A. Jimenez from Texas A&M university. He has played an important role in the work we have collaborated together, I'm greatly influenced by his meticulous attitude and profound knowledge. In regards to the work here in Barcelona Supercomputing Center, I thank Francesc Subirada. With Mateo, they have spent numerous efforts in building this world class supercomputer and a fabulous workplace for researchers. It has been an great honor to stand by them exhibiting the MareNostrum, the stunning artifact to the visitors from my home country, China.

In particular, this work has been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557; the HiPEAC Network of Excellence, by the European Research Council under the European Union's 7th FP, ERC Grant Agreement n. 321253; and by a joint study agreement between IBM and BSC-CNS (number W1361154). I have also been funded by the Chinese Scholarship Council under grant 2010608015.

At last, I would like to thank my parents for all their love, encouragements and supports. With all of that, I can pursue this Ph.D.

Qixiao Liu
March, 2016

Contents

1 Introduction

Energy is becoming one of the most, if not the most, expensive resource in computing systems. This trend will continue as the price of energy continues to rise, increasing in recent years by up to 70% in several European countries [28].

- In a large-scale computing facility (LSCF), energy for computing already accounts for 20% of the total cost of ownership [8,39]. In addition, the Power Usage Effectiveness (PUE) is still above 2.0 in most LSCFs in the year 2015 [24,38]. This metric compares the energy used on the computing facilities with the total energy consumed including other facilities such as power delivery and cooling system: $PUE = \frac{total\ energy}{computing\ energy}$. Thus, the energy cost doubles if we consider all the facilities in LSCFs, implying that the total energy-related cost is already in the same order of magnitude as the hardware-related cost (servers), which dominates the cost of ownership. Additionally, while server cost has remained almost constant over successive generations, energy cost is expected to rise [8]. In fact, in terms of power, current facilities consume several megawatts, enough to power small towns [6]. Meanwhile, in terms of energy, worldwide energy consumption attributable to servers and data centers is estimated to be above 200 billion kWh annually in 2010 [63].

- Energy demand is also an issue for home computers. A typical desktop computer may use in the order of 100–200 Watts (the particular figure depends on the type of computer and peripherals) whereas laptops fit in a lower range (60–100 Watts). The energy cost of running a computer can be computed as $\frac{Watts \times Hours\ Of\ Use}{1000} \times Cost\ per\ kWh$. Assuming that a computer runs for 15,000 hours during its lifetime (around 28 months nonstop) with a cost of 22.1 cents per kWh (household), the energy cost of a 150W desktop is \$497. This figure already represents a significant fraction of the purchase cost of a computer.

- Energy is also critical for the mobile embedded systems, as the computing power of hardware keeps growing whereas the energy densities of the battery technology comparatively slowly grows. Estimating the battery duration of the device with a set of applications running, based on the energy delivered by the battery for a given size and weight is essential for device design.

The so called *power wall* and Instruction Level Parallelism (ILP) wall have been shown to be the major obstacles to maintain the historical rate of performance growth in computing systems [15, 41, 79, 85, 112]. In this line, multi-core and many-core design paradigms have enabled the growth of throughput performance despite the dramatic slowdown in clock speed growth. Multi-core designs offer improved performance per Watt – for similar single-core solutions – for workloads that can make use of multiple cores. However, its establishment as the *de facto* hardware paradigm across most computing domains, together with increasing core counts in each new generation, makes energy consumption in such complex system difficult to be measured at a fine granularity (e.g., per task). Thus, in the current energy-sensitive environment, accurate attribution of energy contribution needs more sensible understanding and study.

Take the scenario of LSCFs where energy already dominates the operational cost for example: In the age of non-virtualized systems, service providers normally charge users based on the time they have used the facility. In this case, as stated in [53], *once a user instance received some physical resources, no other user would be able to share those resources. In such a situation, time is indeed money; so, even if the user instance isn't using the allocated resources, it would make sense to charge the user a flat, per-hour rental rate, because once a set of resources is tied up, the owner can't make rental income out of those resources from any other waiting customer.*

Today's LSCFs providers, cloud-computing for example, serve the customers with *services* based on different models, such as the Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). While in the basic IaaS model, the physical machines are still offered along with virtual machines (more often). The trend is to provide customers highly virtualized online service instead of direct hardware resources, such as PaaS and SaaS. Current cloud-computing providers such as ElasticHost [26] and CloudSigma [21] provide computing power in the form of IaaS. They use similar pricing models, which has been concluded in [74] as follows:

$$P_{vm} = P_{base} + P_{CPU}\left(\frac{f_{CPU} - f_{CPU_{base}}}{f_{CPU_{base}}}\right) + P_{RAM}\frac{RAM_{size}}{RAM_{size_{base}}} \tag{1.1}$$

In this model, customers are offered with the flexibility to choose a specific CPU frequency f_{CPU} which stands for the demanding computing power of the processor, and the memory size RAM_{size}. they are For other resources, they can be priced using the same methodology, but they are ignored in this formula to simplify the discussion. Based on the customer's selection, the corresponding price is calculated with fixed rates P_{base}, P_{CPU} and P_{RAM}. P_{base} is the basic price when the minimum CPU capacity f_{base} and $RAM_{size_{base}}$ are used. With the extra demand on computing capacity and memory size, the price P_{vm} that customer has to pay also rises.

Note that in this case, the boundary between the physical machine and the virtual machine is already unclear. For example, given that customer needs 10GHz CPU frequency, and the per-processor computing power in the infrastructure is 3GHz. We can either presume that the demanded 10GHz CPU frequency can be divided into 3 physical processors entirely, and the rest 1GHz falls into a virtual machine to be placed in any shared processor. Besides, the whole demanded CPU frequency is placed into several virtual machines that the operator can smartly schedule in the infrastructure to maximize the actual resource usage and optimize the overall power and energy consumption. In most cases, the latter one is clearly the preferable choice. Providers benefit from the virtualization of the hardware resources, since they can charge multiple users sharing the hardware resources. As claimed in [53], *in this new scenario, the owner has no reason not to move to an energy-aware accounting system based on actual resource usage; [...] A built-in energy-accounting system could guide the workload management system to make scheduling decisions that result in safe, more efficient workload consolidation.*

From the customer side, energy accounting is also beneficial. For example, if such a system is presented, it can help them to demand proper services to satisfy their need and budget. Most importantly, they will receive billing with higher fairness and accuracy for running their applications. Nevertheless, the benefit of being energy-aware is not limited to this LSCF case, and application can be easily found across all computing domains.

As energy already draws attention from the community, there have been abundant energy-oriented studies. In these works, researchers focus on refining the energy measurement in different perspectives, such as hardware and software based energy measurement in each small time interval, energy and power profiling for programs and systems, energy consumption breakdown in hardware components and program blocks, etc. [14, 27, 105].

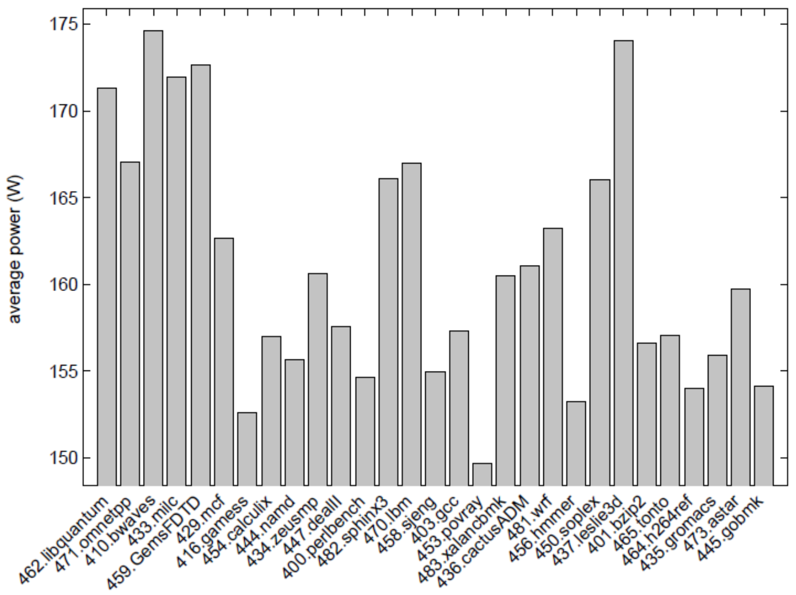


Figure 1.1: Power consumption of SPECCPU 2006 benchmarks on a PS701 system with an IBM POWER7 processor

However, despite the fact that multicore processors have been pervasively used in almost all computing domains in which multiple tasks[1] can simultaneously run, to the best of our knowledge, no mechanism has been proposed to accurately measure the energy consumed by each task in multicore architectures.

Current approaches to measure tasks' energy consumption assume computer system's energy is evenly distributed across all running tasks, as if all of them were using resources homogeneously. However, different applications may easily incur vastly different resource utilization in the shared resources. Such heterogeneous resource utilization translates into heterogeneous power dissipation per application, and therefore, simply dividing energy evenly across running tasks is neither fair nor accurate enough.

To elaborate on the need of accurate per-task energy measurement, Figure 1.1 shows the average power dissipation when executing all the SPEC CPU2006 benchmarks on a POWER7-based system [58]. As shown, different tasks incur different average power dissipation, with the maximum variation being 16%, between `453.povray` and `410.bwaves`. Hence, if a povray-like and a bwaves-like program execute undisturbed in a computing system for a period of time, they will incur significantly different energy consumptions. However, the same amount of energy would be attributed to each, which sum up to the total energy consumption of the system. Note that workloads in this example are fairly

[1]In this thesis, we use the term task to refer to hardware threads belonging to a single-threaded application. And the term workload refers to a set of co-running tasks.

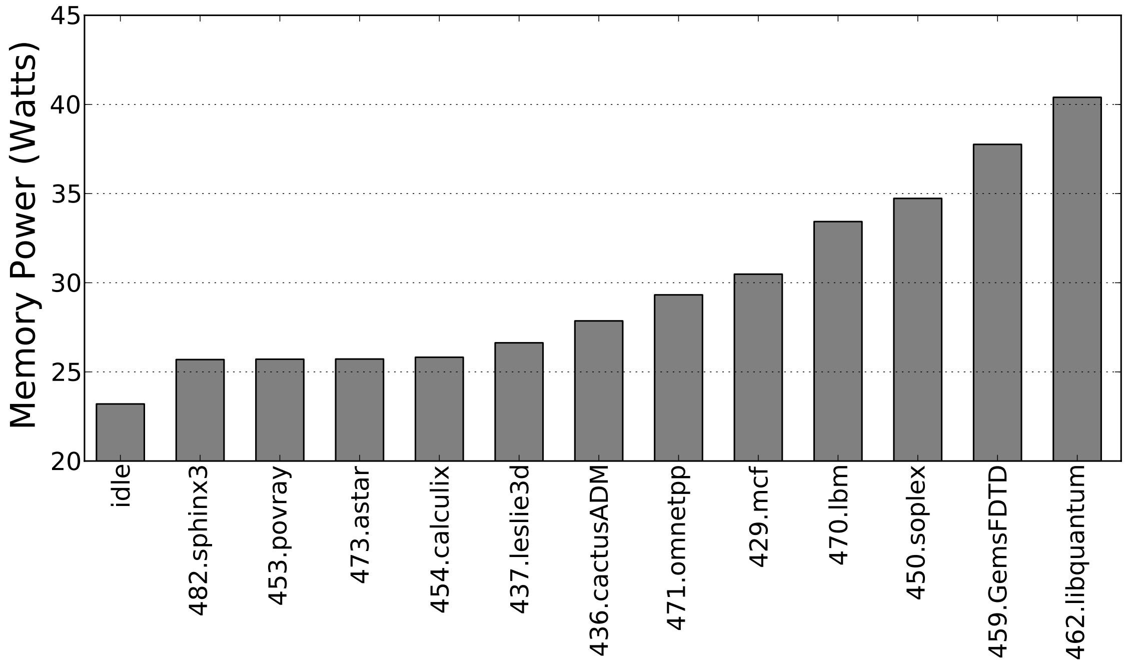


Figure 1.2: Memory power of some SPEC CPU 2006 benchmarks running alone on an Intel Sandy Bridge server, with 8 cores and a 64GB DDR3-1600 memory running at 0.8GHz. Power is obtained using the Running Average Power Limit (RAPL) interfaces [50]. FitPC external multimeter is used to measure wall power. We correlate wall power data with the data collected from the hardware energy counters using time stamps. Representative benchmarks were selected based on previous characterization studies [51, 94].

homogeneous given that they correspond to a single benchmark suite in this case. More heterogeneous workloads including database processing, I/O-intensive applications as well as high-performance ones will exhibit even higher power variations.

Similar trends can be observed on different platforms. We have also performed an experiment with several representative SPEC CPU 2006 benchmarks running on an Intel Sandy Bridge server. In this experiment, we focus on the average memory power during their execution, which represents between 24.6% and 33.9% of the total systemrpower. It is comparable to the entire processor power: on average, the memory system only consumes 6.3% less power than the processor.

Figure 1.2 shows the average memory power consumption of each benchmark when executing in isolation on the system. Different tasks incur different power consumption, with the maximum variation being 54%, between `482.sphinx3` and `462.libquantum` (from 25.7W to 40.4W). Hence, libquantum-like and sphinx3-like workloads executing for the same amount of time would incur significantly different energy consumption.

It is our position that accurately measuring the energy consumed by each task in a computer, instead of considering only the whole energy consumed by the computer, will have plenty of important applications. These applications will not only improve the energy usage attribution in the multicore system, but also enable optimizations on the design and management of computing systems. As a matter of example we list the following applications:

- In LSCF, the energy cost is already dominating the billing. For example, consider the cloud computing provider Cloudsigma [21]: the unit price is 2 cent per hour for a CPU running at 2.5GHz. Assuming an average 50W power consumption and 12.8 cent per kWH electricity price (industry), the energy cost is 30% of the bill without accounting the energy spent in other facilities ($12.8 \cdot 50/1000 = 0.6$ cent per hour). With such figure, users' billing without considering energy cost cannot be fair. Especially in multicore systems, according to our study in Chapter 4, the energy that a task consumes when it co-runs with different tasks can vary in the range of $[-25\%, 40\%]$. Despite this variability, it is our position that when a customer requests the same computing power to run the same task using the same input, the same energy cost should be accounted. Based on that, the provider should consistently charge the customers with the same billing.
- During the design of multi-core and many-core architectures, the per-task performance and system throughput have been mainly taken into account. However, the impact on per-task energy has been somewhat ignore. If the energy consumption of per-task can be measured, the energy efficiency of using multicore processor can be quantified, and more energy efficient design can be devised.
- For computing systems in different domains that use multi-core and many-core processors, collocating tasks with different power needs in the nodes must be done in a way to maximize their performance while minimizing energy consumption. In a given node, allocating appropriate resources to tasks and regulating the frequency/-voltage level to reach the optimal tradeoff between their performance and energy consumption has also prominent importance.

1.1 Thesis Contribution

In pursuance of building energy-aware multi-/many-core systems, numerous efforts are needed in different perspectives. In this thesis, we focus on the per-task energy measurement, as needed by fair energy accounting and system optimization.

In particular, we divide this topic into two distinct concepts: for a particular task, 1) measuring its actual energy consumption in a given workload; and 2) estimating its energy consumption with a given allocation of resources.

Since modern computer components are implemented with diverse techniques and designs, and thus have different structures and organizations, we propose techniques for the on-chip resources and off-chip memory subsystem separately in this thesis.

1.1.1 Per-Task Energy Metering (PTEM)

First, we propose *Per-Task Energy Metering (PTEM)*, which is a measurement of the actual energy consumption one task has during its execution in a multi-core architecture where the resources are shared with other tasks. We define this concept formally as follows:

Given a workload composed by n tasks $T_i, T_2, ..., T_n$ running in a processor with m hardware threads (e.g., m single-threaded cores), *Per-Task Energy Metering* consists in tracking the energy that a given task, T_i, consumes during a given period of time. This requires metering the energy a task T_i consumes in private hardware components (i.e. components only used by the task at a given point in time) for instance, the single-threaded cores in a multi-core CPU, and shared resources, such as Simultaneous Multi-Threaded (SMT) core and shared L2 or Last Level Cache (LLC).

The difficulty with shared resources resides on the fact that they can serve requests from different tasks concurrently, and each request type may generate different internal activity in the resource with variable duration. This seriously challenges per-task energy metering. Current methods for energy metering focus mostly on time-shared resources (e.g. CPUs) and are based on usage time and allocated resources. This may be adequate if static power dominate the total power consumption. However, this is no longer true with the shift towards energy-proportional systems [5] where most of the energy consumed by an application – and hence, its cost – is due to its activity. Hence, in an energy proportional system two customers that incur different utilizations across similarly allocated resources for similar usage time, will be accounted the same energy consumption while in reality their energy consumption profiles can be quite different. In [53] authors run several homogeneous programs in isolation on the same platform for a fixed period of time. Results show that power dissipation across these homogeneous programs with similar resource and time allocation may vary more than 20%. More heterogeneous workloads including database processing, I/O-intensive applications as well as high-performance ones exhibit higher power variations.

Our view is that, the energy metered to a given task should be proportional to its resource usage. This includes the number and type of accesses to the different resources and, for stateful resources (e.g., Branch Target Buffer, caches and Translation Look aside Buffer (TLB)) the fraction of the space occupied by the task. The accuracy of per-task energy metering depends on the characteristics of the hardware resources used and the hardware support enabled for energy metering. Note that when we have per thread energy metering, energy for multi-threaded applications simply consists in adding up the energy consumed by each of its constituent threads.

1.1.2 Sensible Energy Accounting (SEA)

PTEM provides a way to measure the *real* energy a task consumes in the computer. However, the energy metered to a task in a given system, despite it has the same input set, varies depending on other tasks that are running at the same time (co-runners). Apart from being able to measure the energy consumed by a task, we also aim at maintaining the same *Principle of Accounting* that holds for execution time (a.k.a. CPU accounting) [76]: the energy accounted to a task should be independent from the workload in which this task runs. Several runs of the same task with the same input should – theoretically – result in the same energy consumption and hence the same charge in a data center.

Therefore, we propose *Sensible Energy Accounting (SEA)* to fairly account a task a constant amount of energy as if it has been assigned a fraction of resources to use in the computer, regardless of the concurrently running tasks. We define SEA formally as follows:

Let us assume a workload composed of n tasks $T_1, T_2, \ldots, T_n$ running on a processor with m hardware threads (e.g., m single-threaded cores), SEA consists of estimating, for a given task T_i, the energy that it would have consumed if it had run in isolation with a given fraction of the hardware resources denoted *fhr*. Thus, SEA does not give the actual energy consumption of a task, but rather an abstraction of the energy consumption that the end-user can rely on to be fair and consistent.

The main challenge for SEA is how to compute the energy for any task and any valid fraction of the resources, despite the fact that a particular task may incur different activities in different workloads due to interactions with co-runners, and such variation makes its performance fluctuate. This effect makes energy consumption hard to account since it cannot be extracted directly from the energy measured. Our view is that, accurately accounting the energy to a task for a particular fraction of resources depends on precise estimation of activities and execution time this task should have incurred.

1.1.3 Future Impact of PTEM and SEA

We consider both concepts are key to provide clarifications on the per-task energy usage, from the perspective of actual workloads and feasible allocations of resources. We consider that PTEM and SEA can be separately used or combined, but in any case they will have several important applications across different computing domains. We show some cases where PTEM and SEA can apply to:

- *Selection of appropriate co-runners.* Task interaction in hardware shared resources may negatively affect tasks hurting performance and increasing energy requirements. PTEM and SEA can help the OS scheduler or a runtime-based scheduler to decide which tasks must be run and when, thus reducing the total energy profile.

- *Energy/Performance optimization.* While allocating more resources to a program may make it runs faster, it could also increase its power consumption, and vice versa. Thus, the net effect on how the total energy profile relates to the resource allocation is unclear. Accurately measuring the energy consumed in different processor (e.g. number of cores) and software (e.g. scheduling) setups will justify their effectiveness in energy-saving. Sensibly accounting the energy consumed per task would allow finding the optimal setup that leads to the lowest per task energy consumption, and thus the lowest system energy consumption (shown in Section 6.6.2).

- *Billing in data centers.* Data centers charge users for the use of their resources. The fact that costs will be dominated by energy, makes billing systems more and more energy-centric, so that part of the bill is directly dependent on the energy consumed by users' running jobs. Measuring the energy each task consumes, rather than evenly dividing the cost of energy among running tasks, would allow data centers to accurately account the energy cost. Sensibly accounting the energy each task consumes under a given fraction of resources would give a fair billing upon their energy profile. Such methods can facilitate the energy cost integration in different service models, even with higher abstraction levels such as SaaS and PaaS. Since the execution of each virtual machine instance can be tracked, the energy in that physical node can be metered and accounted using PTEM and SEA.

1.2 Thesis Structure

The structure of this thesis is organized as follows:

- Chapter 2 introduces the state of art of energy/power measuring and profiling techniques in computing systems. We abstract several important concepts to show how our work is different from them, and how we advance this topic from a new perspective.

- Chapter 3 introduces the simulation framework we used in this thesis. We give a detailed description of our architecture and power simulator. Also, we have also introduced the benchmark suite and metrics to evaluate our proposals.

- Chapter 4 has made a case of accurate PTEM for on-chip resources. We propose an idealized reference model to perform accurate PTEM based on the resource utilization of each task. A simple, yet accurate, implementation of such approach is also proposed. The focus is the main shared hardware resources in current multicore processors, including SMT core and LLC.

- Chapter 5 introduces the PTEM model in DRAM memory system and an efficient implementation of such method. A case study, in which the SPEC CPU 2006 benchmarks have been characterized using the proposals is also presented.

- Chapter 6 develops the concept of SEA from a theoretical point of view and discusses how it can contribute to different computing domains. Then, a low-overhead hardware mechanism to obtain SEA for on-chip resources in a multicore architecture is presented.

- Chapter 7 makes a case of SEA model in the DRAM memory system when one task has been assigned different fraction of resources on the chip. The interactions of memory requests in the memory controller and DRAM devices have been analyzed. A practical and low-overhead implementation is also proposed.

- Chapter 8 concludes the work in this thesis and shows directions for future work.

- Chapter 9 lists the publication related to this thesis and during the PhD study.

2

Related Work

Nowadays, modern microprocessors integrate in the order of billions of transistors on chip and operate at a frequency of several gigahertz. The power wall has already become a major obstacle in satisfying the growing computational needs. The multi-core/multi-threaded design paradigms have enabled the growth of throughput performance despite the slowdown in clock speed growth. Nevertheless, power dissipation and current delivery limitation make it hard to keep scaling indefinitely along the dimension of on-chip thread count. Therefore, accurate measurement and profiling of the energy/power consumption is needed so that future systems can optimize the power dissipation to better convert the electrical power into computing power.

In this chapter, we show the state of the art in the power/energy consumption measurement and profiling in computing systems. First of all, we introduce the energy consumption taxonomy, based on which, we elaborate the classification to better fit our needs. Then, we review several studies on how to abstract the energy of the devices using different means. In the fields of energy or power profiling, we illustrate works on different categories including: power characterization of computing systems, hardware components power consumption breakdown, and software component energy breakdown. In the context of these related works, we can see how our contributions advance the state of the art in this field.

2.1 Energy Consumption in Computing Systems, A Taxonomy

Within a given computing system, *energy* stands for the source which powers the hardware devices to operate. The standard measurement of *energy* is *Joules*, directly reflecting the electricity cost. And *power* is the rate at which the *energy* is consumed, and is measured in *Watts* which corresponds to *Joules* per second.

The distinction between *energy* and *power* is key to understand per-task energy measurement in multi-core architectures. For instance, when several tasks run concurrently in a multi-core system with abundant shared resources, the power dissipation of one task is most probably reduced compared with the case when it runs alone in the system. However, the energy consumption incurred by its execution is undetermined because of the – likely – prolonged execution time. In this thesis, we study the energy consumed by tasks when running in resource-sharing multi-core systems, by analyzing the power each task dissipated in time intervals during their execution.

In digital Complementary Metal-Oxide-Semiconductor (CMOS) technology, the energy consumption mainly comes from three resources: a) The logic transition that makes the current flow through the transistors. This occurs when the circuit transitions back and forth between the two logic levels. The electrical energy is consumed by the parasitic capacitances and resistance of transistors. b) The short circuit current that flows directly from supply to ground when the n-subnetwork and the p-subnetwork of a CMOS gate both conduct simultaneously. c) The leakage current between the *source* and *drain* of transistors. In former studies, they have been commonly categorized into *dynamic power* and *static power* [73, 113]. Specifically, *dynamic power* includes power dissipated by the logic transition and short circuit current, and *static power* refers to the power dissipated by leakage current. However, such classification, although has been conventionally used to study circuit and system power consumption, does not fit our need to attribute the energy consumption to tasks since we need more precise categories. Therefore, for the sake of clarity, we break down the energy consumed in a computing system into three main components: *dynamic active energy*, *dynamic maintenance energy* and *leakage energy.* These terms are consistently used in this thesis.

- *Dynamic active energy* corresponds to the energy consumed performing those actions needed by the instructions executed, such as the energy used to read a register or to issue an instruction. When considering the energy consumed during a given time interval, we can also express as the *dynamic active power*, which may vary along time.

- *Dynamic maintenance energy* corresponds to the energy wasted in useless activities not triggered by any particular instruction, for instance, the significant clocking power that is consumed in idle blocks. Similarly, many SRAM arrays such as cache memories precharge some bitlines every cycle in order to speed up accesses. However, such activity is useless if no access occurs [16]. Note that the energy consumed due to an access corresponding to a useless instruction (e.g., a misspeculated instruction) is considered as *dynamic active energy* despite such activity is useless because the action has been triggered by the instruction under execution. Due to the fact that these useless activities constantly consume energy during the whole system active period, *dynamic maintenance energy* can also be expressed as *dynamic maintenance power* in most cases.

- *Leakage energy* corresponds to the energy wasted due to imperfections of the technology used to implement the circuit. Thus, it includes all energy wasted due to undesired leakage current and parasitic current from supply to ground. Leakage energy persists whether a computer is active or idle, since the leakage and parasitic current flow through transistors even when the transistors are turned off as long as they are powered up. Thus, *leakage energy* consumed in a given time interval can also be expressed as *leakage power.*

Bear in mind that *dynamic active and maintenance power* are both derived as a superset of logic transition and short circuit current in the CMOS circuit, and they can be summed up to *dynamic power.* In this sense, our study can be easily aligned with former studies. Breaking down dynamic active and maintenance energy is useful in our context since it avoids mixing the energy consumed due to the activity triggered by the instructions executed and the energy that cannot be attributed to any task, especially if several of them are running. *Dynamic maintenance power* has been recognized as platform power in some other works [102].

For the studies related to hardware analysis, the focus is mainly on power consumption since it is a direct reflection of the device computation power and sensitive to the thermal capacity. In contrast, for software related studies, the focus is instead on the total energy consumed by the piece of software that is executed. In the following sections, these terminologies are carefully distinguished.

2.2 Computing System Power Measurement

Obtaining power figures of complex, highly-threaded multi-core system is a difficult challenge. The effort invested on this task is large. The most common approaches can be classified as either direct power measurement or indirect power measurement, although some studies have considered a hybrid approach to obtain improved results [43, 48].

2.2.1 Direct Power Measurement

Direct hardware-based power measurement consists of measuring the current and supply voltage level on a particular component, for instance, the processor and the memory system. Then the measured values are used to compute the power. Such measurement requires different types of meters, some specifically designed circuits embedded in the platform, and power sensors inside the device.

Meters measure the power dissipation of the device in a straightforward way by connecting them between the power supply and the measured component. However, the widely used digital multimeter [55] or AC power meter [109], sample the measured device at a coarse granularity, normally at around 1 Hz.

Direct power measurement typically needs specialized device support, as explained in [105]. Nowadays, most of the servers have the service processor designed inside, which is a hardware and software integrated platform that works independently from the processor and the operating system. The service processor uses the power sensors to monitor the power, and voltage and temperature sensors to refine the measurements [43, 48]. The monitored results are read through an interface by the controller to provide the data to the operating system. The information can be used by the software to optimize the performance, power and energy efficiency. Furthermore, such hardware/software support also allows promoting the sampling frequency up to 3kHz [27].

2.2.2 Indirect Power Measurement

Indirect power measurement can be performed on a simulation platform or at runtime inside the operating system. Thus, measurements are less accurate. However indirect power measurement does not need specific hardware support such as the service processor, and can better correlate the power with the performance of the device.

Hardware-level power model. In the case of microarchitecture simulation, normally a hardware-level power model is used [65, 91]. As described in section 2.1, the classical breakdown of energy in CMOS circuits includes *dynamic power* and *static power.* For a particular hardware device, or a component in the processor, the power can be derived with the following two formulas:

$$P_{dynamic} = C \cdot V_{dd}^2 \cdot \alpha \cdot f \tag{2.1}$$

$$P_{static} = I_{leakage} \cdot V_{dd} \tag{2.2}$$

where C stands for the load capacitance, V_{dd} stands for the supply voltage, α represents the activity ratio on the hardware device, f is the clock frequency and $I_{leakage}$ stands for the leakage current of the circuit. Such low level models are arduous to use in architectural studies since the low level details are hard to derive for all components, thus making the estimation inefficient and costly. Wattch and CACTI tradeoff the estimation accuracy with simulation time by flexibly modeling the structure of each component with a general purpose model [16, 87]. Such characteristic helps them being pervasively used in research studies. In many cases, these models are compared against approaches using circuit-based mechanisms.

Software-based power model. Although the hardware-level models can provide very accurate but time-consuming power information, the online power estimation often relies on the software-based models. Such models use performance statistics supplied by the operating system, where multiple indicators are used to reflect the hardware states and task execution. The selection of indicators is normally based on tuning the estimation by comparing with real system power and the result of hardware-level models [7,11,12,14,36]. In general, these models rely on collecting data from a set of events counters, voltage and temperature sensors, with coefficients derived from an empirical linear regression model. Depending on the system under study and the purpose of the power estimation (e.g., analyzing a hardware component, a process in the operating system or a program phase) different sets of events may be chosen. Note that in the software-based models, the term event may not directly map to the Performance Monitoring Counters (PMCs), but may also be a calculated metric, such as the Instruction Per Cycle (IPC).

Power model abstraction. In both cases, when estimating the runtime energy consumption, the power models can be generalized as analytic functions of a set of parameters, where the power consumption incurred by the execution of a program is derived based on their correlations. Therefore, in a system with J major components, each with I events count, the runtime power is computed as follows:

$$Power = \sum_{j=1}^{J} \beta_j \times \left(SP_j \times T_j + DP_j \times \sum_{i=1}^{I_j} \alpha_j^i \times e_j^i \right) \tag{2.3}$$

where SP_j and DP_j stand for the pre-calculated *static power* and *dynamic power* of component j. T_j is the activated time of component j and e_j^i is the event count obtained for j. α_j^i and β_j stand for a set of coefficients derived through a linear regression model.

The same formula can be applied to software-based models, the difference is that, J refers to a set of selected indicators in any particular use case, each with I as event count. SP_j, DP_j are derived through a linear regression model as well as α_j^i and β_j.

Note that we have used the approach based on the hardware-level power model in our simulation framework.

2.3 Energy and Power Profiling

2.3.1 Processor power consumption characterization

The power and thermal characteristics of a processor are essential for designing its power delivery system, packaging, cooling, and power/thermal management schemes. For such purposes, the maximum power and thermal profile of a processor need to be studied. In the following works, a set of micro-benchmarks – known as *power virus* – are designed to stress the processor to its peak power.

In [56], the concept of maximum power consumption has been refined into maximum sustainable power and maximum single cycle power. The *maximum sustainable power* is the maximum power of the processor that lasts for a time interval that is adequately long. Accurately characterizing it is important as it guides the design of the power delivery system and the packaging requirements for the microprocessor. Similarly, the *maximum single-cycle power* is the maximum power that can be consumed by the processor during one processor cycle. It holds an important key to estimate the maximum transient current that can be drawn by the microprocessor. By taking into account the information on instructions, input data and architecture details, this study generates micro-benchmarks and tests the above characteristics of a particular processor setup with simulation.

In [10], Bertran et al. present a tool to generate micro-benchmarks to explore the maximum power consumption of a real machine. With configurable low-level micro-architecture semantics knowledge of the machine, a taxonomy in terms of energy per instruction (EPI) and processor activity characteristics has been developed. Using such information, authors use a compiler-like pass-based code generator to provide flexibility and full control of the micro-benchmarks generation.

Using the same methodology, Kestor et al. [61] characterize the on-chip memory hierarchy by designing a set of micro-benchmarks that move data through different levels of cache.

2.3.2 Hardware component level power consumption breakdown

In recent years, there has been an increasing interest in breaking down the power consumption to different hardware component levels in different environments from data centers [9,59] to smartphones [17,20,90,92]. Those detailed power measurements improve the characterization of the hardware device, and thus the future designs and implementations can improve the power/performance characteristics of the system.

Many proposals [14, 17, 82, 90, 92, 95] estimate the overall system energy consumption within the software using similar PMC-based approaches as introduced in 2.2.2, and then break it down across the different hardware components at a coarse-granularity, such as the processor, memory and screen. POWER7 processor uses power proxy [32,48] where the monitored power estimations divided among each core. Such model uses as a proxy around 50 dedicated hardware counters, along with voltage, frequency and temperature sensors. Similar firmware is also implemented in Intel Sandy Bridge architectures power management module to break down the power consumption of the system [99]. That firmware uses 100 PMCs for *active power* distribution, and voltage and temperature sensors for *static power* distribution.

2.3.3 Software component energy consumption breakdown

From the software side, refining the runtime energy measurement during program's execution in a given platform is also a research hotspot. Several studies focus on attributing energy to the execution phases or blocks of a running program.

Performance and power vary through the execution of a program. To better analyze the program power behavior and optimize power usage, timing-based power behavior profile is required. Similar to Simpoint [103], techniques based on the basic block vector compare the similarities between different time intervals, to find the representative ones [44]. Based on this, Hu et al. [45] proposed a technique to find the representative phases in a given time interval by incorporating the control flow and runtime events profiles.

Systematic profiling tools characterize the program runtime behavior in different ways [54, 67, 70, 75, 97] (e.g., sampling events like stack traces, hardware events, etc.). By crosscorrelating this information with the executable binary, these tools can locate the hottest process, routine, code regions, library/kernel calls, and measure the performance across different compilations and/or platforms. By correlating with online power measurements, such tools can also enable fine-granularity distribution of the energy.

Shen et al. [102] proposed a request-level OS mechanism to meter power consumption to each server request based on PMCs [7]. The authors consider both active and maintenance power and attribute it to the responsible server requests. However, the per-task energy estimates obtained with this approach cannot be accurately obtained since, as stated by the authors, *"Request executions in a concurrent, multi-stage server contain fine-grained activities with frequent context switches, and direct power measurements on such spatial and temporal granularities are not available in today's systems"*.

In a given time interval during the execution of a program, its power consumption is determined by the bunch of instructions that execute through the pipeline, which may be of different types, exhibit data dependencies, incurred different activities, etc. Tools like *Linux perf* [70] and *oprofile* [54] can identify an executed instruction periodically, and thus allow locating a coarse code region where this instruction resides. By correlating such techniques with the power consumption sampling, authors in [69] attribute power consumption of each sample period to the basic block where the sampled instruction resides in. Conversely, authors in [68] propose to estimate the instant power consumption at runtime by pre-characterizing the power that could be consumed by each basic block. Their estimation not only takes into account the instruction types and mixes, but also explores the inter-block effects to recalibrate their estimates, which is enabled by fine-granularity simulation. However, the power of the processor is determined by the activities of basic blocks executed together in a time window, denoted as superblock [40, 49]. Superblocks has diverse combinations of basic blocks, thus, for an application with complicated control flow, it is infeasible to pre-determine the power consumption in advance.

2.3.4 Current per-task energy measurement models

The above studies have shown to be very accurate in profiling per-component and overall system's energy consumption. However, the hardware-level approaches focus on breaking down the energy to the main hardware components, in which only the activities in the hardware have been taken into account. The task-level interactions, either from the operating system or the Task-Level Parallelism (TLP) on the hardware, have been ignored. Therefore, these approaches do not fit for per-task energy measurements. In contrast, the software-component approaches can only be performed under an important assumption:

the application is the only one scheduled on the processor and it is accounted all the energy consumed in the system, which allows performing the component-level breakdown. In the scenarios where multiple tasks concurrently run, these approaches fail to abstract the task energy from the energy consumed in the system. In summary, the former introduced studies are denoted as Per-Component Energy Metering (PCEM).

Next, we analyze the mechanisms for per-task energy measurement that can be derived from current multicore and multi-threaded systems. In modern multicores, the total energy consumption of the system and its main components can be monitored or accurately estimated during a long-enough time interval. In the scenarios where N tasks $T_1, \ldots, T_N$ are concurrently running, the goal of per-task energy measurement is to distribute the energy among them. A simple and naive method is to evenly split the energy to the running tasks, which we denote *Evenly Split (ES)* model. Unfortunately, this is the most commonly used method nowadays as task-level hardware activities are not easy to identify in general and per-task energy measurement did not draw enough attention until recently. Thus, for a given task in *ES* model, the energy assigned to it is calculated as follows:

$$Energy_i = \sum_{j=1}^{J} \frac{e_j}{N} = \frac{Energy_{total}}{N} \tag{2.4}$$

where e_j stands for the energy consumed in each component, and the sum of all these values corresponds to the total energy consumption of the system $Energy_{total}$.

To take one step further, we can correlate some available task-level metrics with the energy attributed to the running tasks, e.g., the committed instruction count of each task and other PMC values. These task-level metrics roughly indicate the usage of the hardware resources done by each task. We denote this approach *Proportional To Access (PTA).*

Note that we have to derive the PTA model separately in different hardware structures, mainly the core, LLC and memory system. In the case of the LLC and memory, PTA is a simple approach that distributes energy to tasks proportionally to the number of accesses to each structure. In current processors, per-task LLC and memory accesses can be monitored with performance counters [58]. In contrast, the core slices have many components that can incur diverse activities. Thus, from the set of available PMCs, an empirical linear regression model is used to correlate the energy consumption with tasks.

Thus, for the *PTA* model, the energy attributed to a task i, can be formalized as follows:

$$Energy_i = Energy_{total} \times \left(\sum_{j=1}^{J} \left(\beta_j \times \frac{a_j^i}{\sum_{i=1}^{N} a_j^i} \right) + \alpha \right) \tag{2.5}$$

where $Energy_{total}$ stands for the energy consumed in the system, a_j^i stands for the activity count task i has in component j, and $\sum_{i=1}^{N} a_j^i$ stands for the sum of activities in component j. β_j and α are a set of coefficients derived from the linear regression model. Note that both *ES* and *PTA* models are closed-loop methods, since we perform the attribution of energy based on accurately monitored system energy.

It is our position that existing methods in current systems will not go beyond the scope of these two models. However, such models lack the capabilities to deliver accurate per-task energy estimates. In order to obtain more accurate estimates in multicores systems, we need support from the architecture level.

2.4 Summary

In this chapter, we have described the state of art on energy measurement in current computing systems. Directly measuring the power of the computing system demands external devices. This measurement represents the actual power consumption, but it does not provide enough information to estimate per-task energy measurement. To approximate this measurement, indirect approaches have been proposed based on performance monitoring counters. Such solutions have also inspired further studies on breaking down the energy to different components, both in the hardware and software level. However, as multicore processors have already become the reference platform in almost all computing domains, to the best of our knowledge, no model has been reported to accurately provide per-task energy measurement.

Per-task energy measurement can be easily distinguished from the former works, since all these studies are focused on the energy consumption of the whole system. The proposals on per-task energy measurement in this thesis aim at providing much more accurate information and concrete models to solve the ambiguities in distributing energy consumption of a computing system to its multiple running tasks. We will show in the following chapters that it is not trivial to achieve such goals. With simple and naive models that can be plainly derived, such as *ES* and *PTA*, none of them can estimate per-task energy consumption with satisfactory accuracy. In contrast, our proposals, PTEM and SEA, significantly advance the state of art in this field through hardware approaches.

3

Experimental Framework

In this chapter, we describe the simulation framework we use to implement and validate our PTEM and SEA proposals. To this end we build on a set of cycle-accurate architecture performance simulators, power simulators, and benchmark suites. Combining those elements we build our own experimental methodology, which we complement with the approppriate metrics to evaluate PTEM and SEA.

3.1 Simulation Framework

Simulation has shown to be a powerful and efficient tool for research in the computer architecture field. Simulation is used pervasively in both academic community and industry. In particular, microarchitecture simulation has been widely deployed in the computer architecture arena. The main advantages of microarchitecture simulation are as follows: a) those simulators are capable of modeling different levels of architecture details and setups; b) with reasonable tradeoffs between execution time and simulation detail, those simulators can achieve highly accurate results compared with executions on actual hardware or lower-level simulations (e.g. gate level or register transfer level [34,35]); c) those simulators are flexible and so convenient for applying hardware changes that are needed to evaluate novel ideas, such as PTEM and SEA. Such features make microarchitecture simulators – just *simulators* from now onwards – the most suitable platforms to perform the research of this thesis. Note that we have modeled a general-purpose processor and memory system, not a particular real system. On the one hand, this is because cycle-accurate simulation of a particular processor would require privileged access to its detailed design data. On the other hand, our study focuses on the methodologies to measure the energy for each

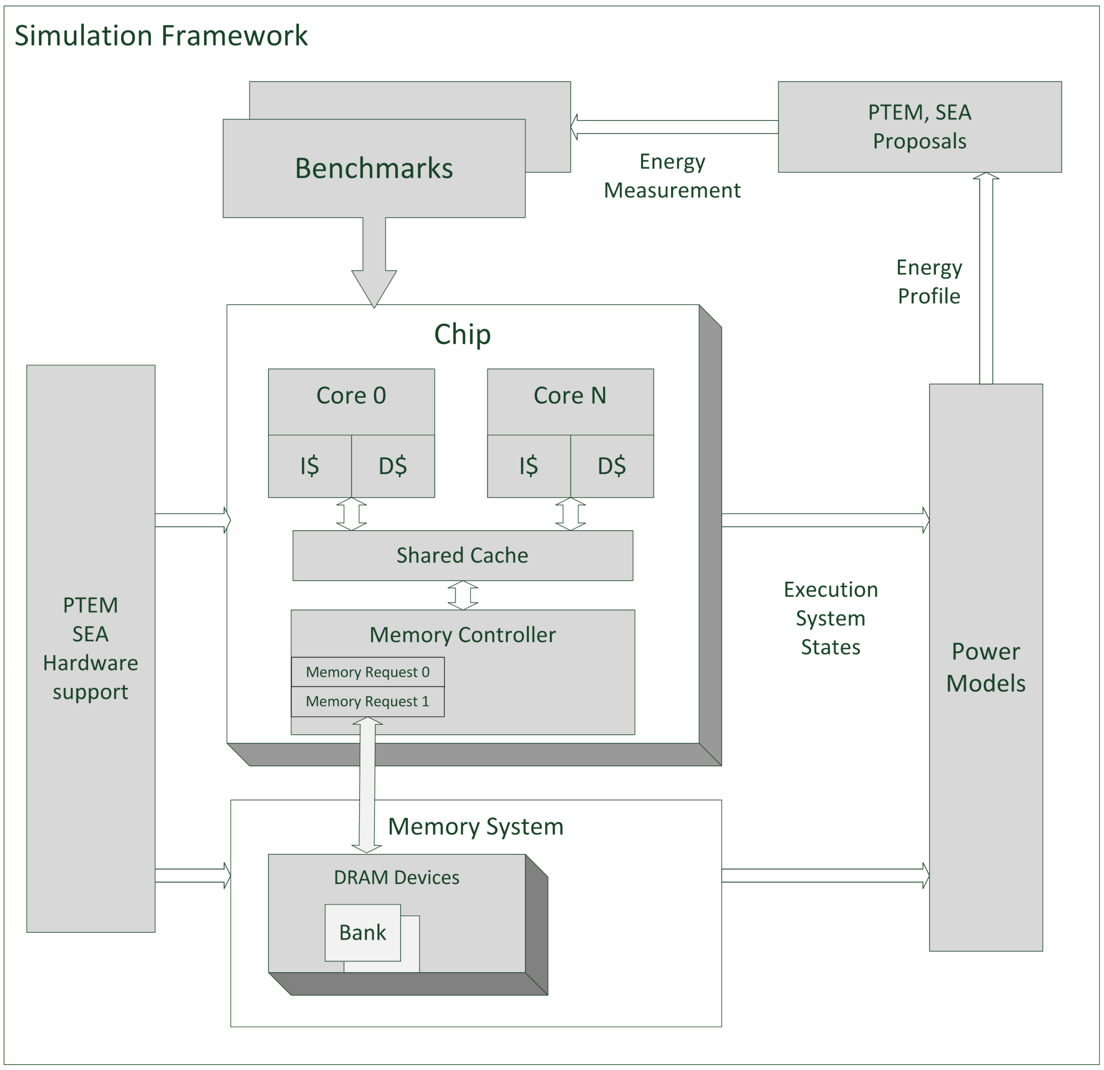


Figure 3.1: Diagram of the simulation framework

task in a generic way, which can be adapted to different processors and memory systems as we have modeled all the main components in modern processors.

The target of this thesis is to explore PTEM and SEA in two main components of a computer system: the processor (core slice[1], shared caches, buses, etc.) and the memory subsystem. To this end, we need concrete information on the dynamic behavior and energy consumption of a program during its execution in a computing system. In addition, we need efficient simulation, yet accurate, to allow a large amount of experiments to be performed, so that we can come up with enough results to prove the advantages of our solutions.

[1]In this thesis, we refer the processor pipeline units and the private caches as the core slice

Our simulation framework builds upon two pillars: performance and power simulators. A diagram of this framework is depicted in Figure 3.1. In this framework, we use the performance simulator to emulate the timing behavior of benchmarks, which are reflected into energy consumption by the power simulator. Based on the derived energy, our PTEM and SEA proposals attribute it to the running benchmarks. Since our proposal may need extra hardware support from the architecture, the feasible changes can be applied to the simulator, thus allowing us to explore the design space. In this thesis, we have used existing simulators instead of developing a brand new platform to avoid wasting efforts, and because using these already well-designed and validated simulators lets us have high confidence on the simulation efficiency and accuracy.

3.2 Performance Simulators

Most current performance simulators focus on the on-chip components, while the highly complicated behaviors in memory system have been somewhat ignored. In such simulators, some naive memory models have been applied assuming either fixed memory request latency with infinite bandwidth or simply aggregating the latencies of consecutive memory requests. This is problematic since processor and memory systems are highly dependent on each other. As the processor in general operates at a higher clocking frequency than memory, sometimes it is forced to be stalled a significant number of cycles waiting for the memory requests to be served. Specially, consecutive memory requests could generate different levels of conflicts, e.g., when accessing the memory banks, buses, memory controller resources, etc. Instead, it can also be the case that the latency of these memory requests gets totally or partially overlapped due to the large capacity of the memory system. Analogously to the case of standalone processor simulation, standalone memory system simulation can neither reveal the whole picture. For this reason, we build our performance simulator by interactively integrating a processor and a memory system simulator.

On the processor side, we use MPsim [3], a trace-driven cycle-accurate simulator that supports CMP and SMT architectures, which is an enhanced version of SMTSim [110]. This simulator is developed at UPC, and has been used in a large number of prior works [18, 77, 78, 86, 114]. MPsim emulates the processor with a model of the processor pipeline, on-chip cache hierarchy and buses. The simulated pipeline stages are as follows: fetch, branch predict, decode, register rename, register read/write, cache read/write, execute until commit. Since trace-driven simulation uses instruction traces that are recorded during a previous execution of a program, MPsim is adapted to emulate the impact from

Table 3.1: Summary of some DRAM device timing parameters used in DRAMSim2

Parameter	**Description**	**Cycles**
t_{RAS}	Time interval between a row access command and data restoration in a DRAM array	24
t_{CAS}	Time interval between a column access command and the start of data return from the DRAM devices	10
t_{RCD}	Time interval between row access and data ready in the sense amplifiers	10
t_{RTP}	Time interval between a read and a precharge command	5
t_{RP}	Time interval that it takes for a DRAM array to be prepared for another row access	18
t_{RRD}	Minimum time interval between two row-activation commands to the same DRAM device	4
t_{RC}	Time interval between accesses to different rows in a bank	34
t_{WR}	Minimum time interval between the end of a write data burst and the start of a precharge command	10
t_{WTR}	Minimum time interval between the end of a write data burst and the start of a column read command	5
t_{RFC}	Time interval between refresh and activation commands	107

wrong path instructions by using a separated dictionary to provide information on all static instructions to avoid compromising the accuracy of the simulation results. In addition, we add several enhancements to it in this work to make it better fit with the memory system simulator and power models. For instance, we have added an exact read/write port model for each components to ensure the activities incurred on each component in every cycle can be simulated precisely and power can be accounted conveniently.

For the memory system, we use DRAMSim2 [98], also a cycle-accurate simulator to emulate the DDR2/3 memory system with a set of DRAM devices, a memory controller and a standard memory bus. This simulator is either driven by a trace of memory requests with their timing information, or connected to a processor simulator through a robust interface. On the arrival of a memory request, the memory controller decomposes it into the corresponding DRAM device internal commands and schedules them to perform operations in the DRAM devices. And after the memory request finishes, DRAMSim2 returns the data to the processor. These internal procedures are modeled with circuit-level details, such as memory bank activating, data read/write and precharge, etc. The latency of each command follows a strict timing model, which is a generic abstraction of modern DDR2/3 memory systems. It enables convenient configuration in the simulator, since parameters from different technologies and designs of DRAM devices are largely different. A brief description of some parameters used in this model is shown in Table 3.1, along with example values obtained from the specification on DDR3_64B_SG15 with 0.68nm technology. A more detailed description of how DDR2/3 DRAM memory works can be found in Section 5.2.1.

Table 3.2: Configuration Summary

Parameter	**Description**
Chip details	
Cluster count	1, 2, 4 and 8
Core count	4 cores per cluster; 1-, 2-thread SMT
Supply voltage	1.0V
Technology	65nm
Core details	
Core type	out-of-order
Fetch, decode, issue, commit bandwidth	2/4 instr/cycle
Branch Predictor	Hybrid 256B Gshare
Branch target buffer	32 entries, 4-way
Return address stack	32 entries
Reorder buffer size	96 entries
Issue queues size	48/48/48 entries for INT/FP/Load-store queues
Register file	164 INT, 164 FP
Functional Units	2 INT ALU (1 cyc), 1 mult (4 cyc), 1 div (7 cyc) 1 FP ALU (6 cyc), 1 mult (6 cyc), 1 div (17 cyc)
Instruction L1	32KB, 4-way, 32B/line (2 cycles hit)
Data L1	32KB, 4-way, 32B/line (2 cycles hit)
Instruction TLB	256 entries fully-associative (1 cycle hit)
Data TLB	256 entries fully-associative (1 cycle hit)
Shared L2 Cache	
Unified L2	2MB, 16-way, 64B/line (3 cycles hit, 300 cycles miss)
Main Memory	
Size	8GB
Frequency	1000MHz
Row-buffer policy	Close-page or open-page
Address mapping scheme	Shared bank
Power-down mode	Fast
Supply voltage	1.35V
Technology	65nm

During the integration of two simulators, the synchronization was relatively straightforward since both simulators are cycle-based. In modern computers, processors normally work at a higher clock frequency than memory, commonly ranging from 1.5 GHz to 3 GHz. The frequency of DRAM DDR2/3 memory normally ranges from 667 MHz to 1666 MHz. In this thesis, as we assume a general purpose architecture, the processor frequency has been set to 2 GHz, and the memory frequency to 1000 MHz, although our findings are not specific to any particular clock frequencies. This particular setup has been chosen to avoid extra synchronization complexity. Although the memory requests generated from the processor in 2 cycles are dispatched together to the memory system, their order is maintained by the memory controller.

Keeping track of instructions in the two simulators is also trivial. As for a load/store instruction, its memory address is used to search through the on-chip cache-hierarchy in MPsim. If it incurs a Last Level Cache (LLC) miss, the same address is used for addressing in the memory system after its execution stalls in the pipeline. Also, the information of this LLC miss is stored in a Last Level Miss Status Handling Register (LLMSHR) to allow other concurrent misses. DRAMSim2 memory controller also preserves the information of each memory request. When one request completes its operations, the memory controller uses a callback mechanism to notify the processor of the returning data and the information related to the memory request. After receiving the information and data, the LLMSHR is iterated to find the matched entry. This entry points to the stalled instruction which generated the memory request, and upon the reception of the memory answer, such instruction is resumed to complete its execution. Note that from the memory side, there is no hard limitation on the number of co-running tasks in the system, which simplifies the integration process to connect DRAMSim2 with a regular multi-core architecture. Instead, the only limit is the number of pending memory requests that can be processed in parallel, which in turn is limited by the number of commands that can be stored in the memory controller command queue (128 entries in our case).

An overview of the configuration of the performance simulator used across this thesis can be found in Table 3.2.

3.3 Power Simulators

To simulate the energy consumption of a program during its execution, with the runtime information provided by the performance simulators, an infrastructure is needed to analyze and quantify the power dissipation of the program on the hardware components. In this thesis, we have used parameterized power modeling infrastructures on the processor and memory system components of different hardware structures. A tradeoff is needed between the low-level details of the hardware designs, the model accuracy and the simulation speed, so that diverse configuration setups and workloads can be experimented efficiently.

The power models for the processor we used in this thesis are analogous to those of Wattch [16]. Wattch-like power models provide a framework where the activity- and time-based power consumption of the major units in the processor are parameterized and quantified, which makes it suitable to be integrated into our performance simulator of the processor. As the technology and configuration continuously change, the power of cache and SRAM-based components in our setup are modeled on top of CACTI 6.5 simulation tool [87]. CACTI is a flexible tool to model delay, energy (dynamic and leakage) and area of cache memories and SRAM-based arrays. Power models for functional units have been

Table 3.3: Summary of the power models on Major On-chip Components

Unit	Parameters				Energy Consumption	
	Size (B)	R/W Ports	Block (B)	Type	Per Access (nJ)	Leakage (mW)
IALU					0.024	1.92
FALU					0.05	4.02
BTB	8192	2	8	cache	0.033	20.94
RAS	2048	2	8	cache	0.019	6.33
DCache	32768	2	64	cache	0.072	59.77
ICache	32768	2	64	cache	0.072	59.77
DTLB	2048	2	8	cache	0.013	5.74
ITLB	2048	2	8	cache	0.013	5.74
INT Register	1312	10/6	8	SRAM	0.016	5.06
FP Register	1312	6/4	8	SRAM	0.012	2.66
INT Issue queue	384	4/2	8	cache	0.013	1.01
FP Issue queue	384	2/2	8	cache	0.012	0.71
LS Issue queue	384	2/2	8	cache	0.012	0.71
Bus					0.004	0.21
ROB	2048	2/2	8	cache	0.021	8.35
LLC	2097152	1	64	cache	1.76	224.75

updated to use modern designs. Although in recent studies the power estimations made from McPAT, a CACTI based power model, show a big gap with the real computer [117], we still use it as our platform. On the one hand, McPAT/CACTI power models are the current *de facto* standard in the computer architecture community, being extensively used in research works for design space exploration. On the other hand, this thesis is neither improving nor covering the gap of such models, but exploring the per-task energy distribution mechanisms based on the estimates made by such models. Therefore, even if power estimates cannot perfectly match with the real system, such analytical model helps us to reveal the interaction between hardware resources and tasks in an analyzable way. In this perspective, McPAT/CACTI power models provide the capabilities for analysis and fast simulation speed to make our research feasible. In Table 3.3, we show an example of the CACTI configurations and output that we have used for some major components on-chip.

Unlike on-chip resources, the memory power is more sensitive to the timing and addresses of memory requests. Although CACTi can provide accurate estimation on the memory power based on a given activity factor, which requires deep understanding of the memory structure, the estimation is rather static and so misses important details. Micron has published a set of *data sheet specifications* for system designers to estimate the power consumption of DDR2/3 DRAM memory. We derive the power model from the data sheet and integrate it to DRAMSim2 seamlessly since they come from the same source. The power model provides the current profiles, which correspond to the state of the DRAM

Table 3.4: Summary of DRAM device current parameters used in DRAMSim2 power model

Current	**Description**	**Value (mA)**
IDD_0	Operate one bank active-precharge current	100
IDD_1	Operate one bank active-read-precharge current	130
IDD_{2P}	Precharge power-down current	10
IDD_{2Q}	Precharge quite standby current	70
IDD_{2N}	Precharge standby current	70
IDD_{3N}	Active standby current	90
IDD_{4W}	Operating burst write current	255
IDD_{4R}	Operating burst read current	230
IDD_5	Burst auto refresh current	305
IDD_6	Self refresh current	9
IDD_7	Operating bank interleave read current	415

devices and actions performed by those DRAM devices. The current profiles are monitored on the real devices when memory requests are processed in the memory system. In Table 3.4 we list some relevant current profiles in this power model. A detailed description of this model can be found in Section 5.2.2.

3.4 Benchmarks

3.4.1 SPEC CPU 2006 Benchmarks

Most of the experiments in this thesis are performed with the SPEC CPU 2006 benchmark [108] suite. This suite is designed and released by The Standard Performance Evaluation Corporation, and aims to provide a standard of measurement or evaluation on the speed and throughput of computer systems. The diverse benchmarks are developed from real user applications, and include compute-intensive and memory-intensive ones. They have been designed, therefore, to stress the processor and memory subsystems. Based on the components they stress the most in the processor, these benchmarks have been categorized as *SPECint* for integer components and *SPECfp* for floating point components.

We have used traces from these benchmarks which have been obtained from their execution on an AlphaServer DS25 with two Alpha 21264C processors running at 1 GHz with the operating system Tru64 5.1b. As for the compiler, we have used DEC Alpha AXP-21264 C/C++ compiler for the for those benchmarks programmed in C/C++, compiled with the `-O2 -non_shared` options, and the DIGITAL Fortran 90/Fortran 77 compilers for the remaining benchmarks. All benchmarks have been compiled with the reference input set. Although Alpha processors are not the state of the art processor nowadays, its Reduced Instruction Set Computing (RISC) instruction set has been widely adopted

Table 3.5: SPEC CPU INT 2006 benchmark description

Benchmark	**Description**	**Language**
400.perlbench	Devired from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).	C
401.bzip2	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O	C
403.gcc	Based on gcc Version 3.2, generates code for Opteron	C
429.mcf	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport	C
445.gobmk	Plays the game of Go, a simply described but deeply complex game	C
456.hmmer	Protein sequence analysis using profile hidden Markov models	C
458.sjeng	A highly-ranked chess program that also plays several chess variants	C
462.libquantum	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm	C
464.h264ref	A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2	C
471.omnetpp	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network	C++
473.astar	Pathfinding library for 2D maps, including the well known A* algorithm	C++
483.xalancbmk	Transforms XML documents to other docs using a modified Xalan-C++	C++

and developed in the community. Thus, its microarchitecture is still fairly similar to other RISC chips. Furthermore, we have simulated instruction traces obtained from this platform, but we have implemented new features in our simulation framework based on state of the art multi-core processors. As a result, we strongly believe that the conclusions obtained in this thesis are valuable across different platforms, since we study the activities triggered by the instructions, not the instruction set itself.

In Tables 3.5 and 3.6, we give a short description of each benchmark in *SPECint* and *SPECfp* together with the language in which the source codes were written. In the case of *SPECint* benchmarks, all the applications are written in C or C++, whereas in the case of *SPECfp* benchmarks, some of them are written in Fortran, C, C++, or a combination of C and Fortran codes. For example, in `435.gromacs` the only Fortran code is the inner loops (`innerf.f`) which typically account for more than 95% of the runtime.

Table 3.6: SPEC CPU FP 2006 benchmark description

Benchmark	Description	Language
410.bwaves	Computes 3D transonic transient laminar viscous flow	Fortran
416.gamess	Gamess implements a wide range of quantum chemical computations. For the SPEC workload, self-consistent field calculations are performed using the Restricted Hartree Fock method, Restricted open-shell Hartree-Fock, and Multi-Configuration Self-Consistent Field	Fortran
433.milc	A gauge filed program: lattice gauge theory with dynamical quarks	C
434.zeusmp	ZEUS-MP is a computational fluid dynamics code developed at the Laboratory for Computational Astrophysics for the simulation of astrophysical phenomena	Fortran
435.gromacs	Molecular dynamics: simulate Newtonian equations of motion for hundreds to millions of particles. The test case simulates protein Lysozyme in a solution	C, Fortran
436.cactusADM	Solves the Einstein evolution equations using a staggered-leapfrog numerical method	C, Fortran
437.leslie3d	Computational Fluid Dynamics (CFD) using Large-Eddy Simulations with Linear-Eddy Model in 3D. Uses the MacCormack Predictor-Corrector time integration scheme	Fortran
444.namd	Simulates large biomolecular systems. The test case has 92,224 atoms of apolipoprotein A-I	C++
447.dealII	deal.II is a C++ program library targeted at adaptive finite elements and error estimation. The testcase solves a Helmholtz-type equation with non-constant coefficients	C++
450.soplex	Solves a linear program using a simplex algorithm and sparse linear algebra. Test cases include railroad planning and military airlift models	C++
453.povray	Image rendering. The testcase is a 1280x1024 anti-aliased image of a landscape with some abstract objects with textures using a Perlin noise function	C++
454.calculix	Finite element code for linear and nonlinear 3D structural applications. Uses the SPOOLES solver library	C, Fortran
459.GemsFDTD	Solves the Maxwell equations in 3D using the finite-difference time-domain (FDTD) method	Fortran
465.tonto	An open source quantum chemistry package, using an object-oriented design in Fortran 95. The test case places a constraint on a molecular Hartree-Fock wavefunction calculation to better match experimental X-ray diffraction data	Fortran
470.lbm	Implements the "Lattice-Boltzmann Method" to simulate incompressible fluids in 3D	C
481.wrf	Weather modeling. The test case is from a 30km area over 2 days	C, Fortran
482.sphinx3	A widely-known speech recognition system from Carnegie Mellon University	C

3.4.1.1 Trace Extraction

In order to perform efficient simulations, we have to perform several optimizations on the simulation time which is sensitive to the size of trace.

Table 3.7: The input sets for SPEC CPU 2006 benchmarks and their simulation starting point (in millions of instructions) using the SimPoint methodology [103]

SPECint		
Benchmark	**Input**	**Fast Forward**
400.perlbench	-I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1	1439900
401.bzip2	input.program 280	107000
403.gcc	166.i -o 166.s	25500
429.mcf	inp.in	90700
445.gobmk	–quiet –mode gtp -i trevord.tst	50300
456.hmmer	–fixed 0 –mean 500 –num 500000 –sd 350 –seed 0 retro.hmm	14900
458.sjeng	ref.txt	822100
462.libquantum	1397 8	237000
464.h264ref	-d foreman_ref_encoder_main.cfg	382800
471.omnetpp	omnetpp.ini	683400
473.astar	rivers.cfg	220700
483.xalancbmk	–	–

SPECfp		
Benchmark	**Input**	**Fast Forward**
410.bwaves	–	1668800
416.gamess	-i triazolium.config	2980700
433.milc	–	897600
434.zeusmp	–	17939
435.gromacs	-silent -deffnm gromacs -nice 0	588700
436.cactusADM	–	18497
437.leslie3d	-i leslie3d.in	637200
444.namd	–input namd.input –iterations 38 –output namd.out	1200
447.dealII	23	41900
450.soplex	-m3500 ref.mps	67400
453.povray	SPEC-benchmark-ref.ini	168600
454.calculix	-i hyperviscoplastic	1099500
459.GemsFDTD	–	31713
465.tonto	–	11500
470.lbm	3000 reference.dat 0 0 100_100_130_ldc.of	17900
481.wrf		2749700
482.sphinx3	ctlfile . args.an4	1740400

Some SPEC CPU 2006 benchmarks execute multiple times with different inputs for the reference test. Those benchmarks are not convenient for us since they lead to an increased simulation time cost. In the study in [94], authors have pointed out that not all input sets are necessary for SPEC CPU 2006 benchmarks. That work shows, for those benchmarks that have multiple input sets, that running a subset of the input sets already provides similar timing behavior to that of all the remaining input sets. In the process of obtaining instruction traces from benchmarks, we use this approach, by executing these benchmarks with the input sets indicated in [94]. This optimization is applied for the *SPECint* benchmarks, including `400.perlbench`, `401.bzip2`, `403.gcc`, `445.gobmk`, `456.hmmer`, `464.h264ref` and `473.astar`, as well as for `416.gamess` and `450.soplex` from the *SPECfp* benchmarks.

Still, simulating the whole instruction trace of a benchmark in a cycle-accurate simulator is unaffordably time consuming. To reduce simulation time, the most commonly used approach is to select representative samples [116]. Random samples appear to be inadequate, while just choosing the beginning of a program could be incorrect due to initialization code. *SimPoint* methodology is proposed by Sherwood et al. [103], which detects program's phases by using the *Basic Block Vector (BBV)*, which counts how many times each basic block appears. Two phases are considered the same if Mannheim's distance between their *BBV* is small. At the beginning, the execution of the program is split into a set of *intervals* of fixed size (e.g., 10 million instructions). Using clustering algorithms, such as *random linear projection* or *k-means*, the samples are joined. The first algorithm is used to reduce the dimension of the *BBV* and, in that way, accelerate the *k-means* algorithm. This last algorithm is run for values of k between 1 and M (M is the maximum number of phases to use) and the intervals are grouped into phases. *SimPoint* chooses the representative of each phase that is closest to its centroid.

Our collection of instruction traces follows the same methodology, as a result, we take 100 million instructions from each benchmark. We list the fast forwards to apply to each benchmark and its used input sets in table 3.7, respectively.

Due to limitations of our simulation infrastructure, we were not able to create the traces from three benchmarks: `459.GermsFTDT`, `483.xalancbmk`, and `481.wrf` from SPEC CPU 2006.

3.4.2 High-Performance Computing Benchmarks

We have also used real traces from a parallel HPC application running on an actual supercomputer: `wrf`. The Weather Research and Forecasting (`wrf`) model [83] is a mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. In this experiment, we use the non-hydrostatic mesoscale model dynamical core.

Simulating all threads of the parallel MPI application implies a significant amount of simulation time as these applications usually run for days or weeks on a supercomputer. We use an automatic mechanism to choose the most representative computation regions to be traced and simulated with a cycle-accurate simulator [37]. This simulation methodology uses non-linear filtering and spectral analysis techniques to determine the internal structure of the trace and detect periodicity of applications. Afterwards, we use a clustering algorithm to determine the most representative computation bursts inside an iteration of the application.

Traces are obtained when `wrf` runs on the MareNostrum supercomputer at the Barcelona Supercomputing Center (BSC-CNS). We obtain four representatives for the five computation phases that compose the 64-thread MPI application. We have used these reduced trace files to feed the performance simulator. We simulate all threads sharing the LLC cache (four threads in this case study) in a CMP architecture (single-threaded cores). When a thread finishes executing, it waits until all other threads have also finished.

3.4.3 Workload Selection

In the experiments we perform in this thesis, the number of benchmarks in the workloads is identical to the thread count of the processor. For example, in a 4-core CMP architecture, we run four benchmarks in a workload; for a 4-core 2-way SMT architecture, we run eight benchmarks workloads. For a wide variety of configurations, as shown in Section 3.2, we need to generate workloads for each appropriately. Several issues have been taken into account for the generation of the workloads: the characteristics of benchmark behavior, the number of generated workloads and the type of the workloads.

As benchmarks with diverse characteristics co-running in a workload will generate very different behaviors, a certain amount of workloads are needed to come up with comprehensive conclusions for our studies. However, given that we have generated traces for 26 benchmarks, to generate the N-task workloads, we could have N^{26} possibilities, which is way too much. Thus, we randomly picked benchmarks to generate a fixed set of workloads. In order to facilitate the interpretation of results and understanding of the features proposed, an appropriate way to generate the workloads is needed. Since the most relevant parameter affecting the timing and power behavior in our environment is the time spent accessing memory, we classify SPEC CPU 2006 benchmarks into two groups. Based on metric Misses Per thousand Cycle (MPKC) in the LLC, we include in the MEM group those benchmarks presenting a MPKC value higher than 3 under a 2MB 16-way LLC setup when each benchmark runs alone. The remaining benchmarks fall in the ILP group. Although the threshold to classify benchmarks can only be arbitrary, as shown later, it was appropriate to segregate distinct timing and power behaviors.

In Table 3.8 we show the benchmarks we categorize into each group. Note that some benchmarks are very sensitive to the LLC size, so they could be classified into the other under a different LLC configuration. Especially when we study SEA, where we consider different LLC sizes, this might be a concern. However, we stick to this classification along all the thesis for the sake of consistency.

Then, from these two groups, we generate three workload types denoted as I, M and

Table 3.8: The MPKC of SPEC CPU 2006 benchmarks under a 2MB 16-way LLC setup, and the group they belong to.

MEM		ILP	
Benchmark	**MPKC**	**Benchmark**	**MPKC**
433.milc	15.90	435.gromacs	1.63
410.bwaves	15.47	473.astar	1.19
462.libquantum	15.08	401.bzip2	1.10
450.soplex	12.59	400.perlbench	0.96
470.lbm	10.34	456.hmmer	0.47
403.gcc	10.07	464.h264ref	0.45
437.leslie3d	5.93	447.dealII	0.43
434.zeusmp	4.80	458.sjeng	0.32
482.sphinx3	4.75	444.namd	0.29
429.mcf	4.54	416.gamess	0.29
436.cactusADM	4.50	445.gobmk	0.25
471.omnetpp	4.48	453.povray	0.02
454.calculix	3.04		

X depending on whether all benchmarks in a cluster belong to group ILP, MEM or a combination of both respectively. We generate 8 workloads per group for each processor setup. Benchmarks in each workload are randomly picked out from all the benchmarks of the corresponding type. In the case of X, half of the benchmarks belong to ILP and the other half to MEM. We do not put any constraint on whether benchmarks can repeat in a particular workload since the random selection of benchmarks is always performed out of the corresponding (original) group of benchmarks.

3.5 Metrics

Reference model. Since there is no reference model presented to meter or account the per-task energy, and due to the complexity of the hardware, there is not a direct way to measure it in real hardware. Thus, in each of our proposals, we first present an oracle model, which exhibits the best scenario where the energy can be measured with as much information as needed. We implement such models in our simulator, despite the fact that such models would incur unaffordable cost in practice, thus being infeasible to be implemented. Therefore, we also present practical and implementable approaches, which trade off the estimation accuracy with cost. We have also introduced several state of the art approaches that our approaches can compare with to show the improvements brought by our techniques.

In this thesis, we use several different metrics to evaluate our practical PTEM and SEA proposals, based on the reference model. The methodology we use is to measure the *off estimation* or *prediction error* of each model with respect to the reference model, which is computed as follows:

$$PredictionError = \left| 1 - \frac{Energy_{model}}{Energy_{ideal}} \right| \tag{3.1}$$

where $Energy_{ideal}$ stands for the energy derived from the reference model, while $Energy_{model}$ stands for the energy derived from the other models. We use this metric to evaluate the accuracy of our proposals on each task.

In some scenarios, we also measure the *prediction error* of the whole workload. In which, we accumulate the estimation of all benchmarks in the workload using the reference model as the baseline.

$$WldPredError = \frac{\sum_{i=1}^{N} |Energy_{ideal_i} - Energy_{model_i}|}{Energy_{measured}} \tag{3.2}$$

where $Energy_{ideal_i}$ stands for the energy derived from the reference model for task i, while $Energy_{model_i}$ stands for the energy derived from the other models. $Energy_{measured}$ stands for the actual measured energy for the whole workload, which is eventually identical to $\sum_{i=1}^{N} Energy_{ideal_i}$. Then, we take the average *WldPredError* across all benchmarks in each workload analyzed in each setup.

4

Per-task Energy Metering for The Processor

4.1 Introduction

Current computing systems lack a proper per-task energy measurement mechanism. Existing approaches to measure tasks' energy consumption evenly distribute computer system's energy across all running tasks, as if all of them were using resources similarly. However, different applications may easily incur vastly different resource utilization across similarly allocated resources. Such heterogeneous resource utilization translates into heterogeneous power dissipation per application, and therefore, simply dividing power across running tasks is neither fair nor accurate enough.

In Figure 1.1 we have shown an example of the energy variation across several workloads even if they are allocated the same amount of resources. These variations are already significant, and they will most probably increase in the future, as system manufacturers pay increasing attention to energy efficiency and energy-proportional computing [5].

A system is energy-proportional if (i) it presents the maximum energy consumption when achieving the maximum performance, (ii) the energy consumption is close to zero when the system is idle, and (iii) the energy increases between these two extremes as performance increases as well. Although current systems are not fully energy-proportional yet, the trend is to move towards this kind of systems. In the presence of more energy-proportional systems, static (and likely leakage) energy will decrease to some extent and dynamic energy will be the dominant source of energy consumption. Under this situation, energy consumption will be more dependent on the application activity, and thus considering per-task energy consumption will be even more necessary.

In this chapter, we make a case for accurate *per-task energy metering* (PTEM). In particular we propose an idealized reference approach to perform accurate PTEM based on the resource utilization of each task. We also present a simple, yet accurate, implementation of such approach. We focus on the main shared hardware resources in current multicore processors: At chip level, we deal with the shared Last Level Cache (LLC) and the network on chip; at core level, we consider simultaneous multi-threaded (SMT) cores, which have a massive amount of shared hardware resources and represent the worst scenario for achieving accurate energy predictions with PTEM.

The benefits of PTEM extend to different computing domains, such as data centers, smartphones or desktop systems. In this chapter, we take a cross-domain approach, in which, instead of focusing on a given target environment, we analyze how to perform accurate per-task energy metering and what hardware/software support is required for an efficient implementation.

Overall, the main contributions in this chapter are as follows:

- We propose an accurate (yet idealized) approach to perform per-task energy metering based on per-task resource utilization. Our approach considers the utilization of each hardware component in the chip (e.g., cores, caches, etc.) and its impact in *dynamic active*, *dynamic maintenance* and *leakage energy*. Both single-threaded and SMT cores are considered by our approach. To the best of our knowledge, it is the first reference approach against which per-task energy measuring mechanisms can be compared.

- We show how state-of-the-art approaches such as *Evenly Split* (ES) and *Proportional To Access* (PTA), as introduced in Section 2.4, fail to provide accurate enough per-task energy measurements.

- We propose efficient designs of our approach to perform per-task energy metering in multicore processors. We illustrate how our designs allow to accurately estimate the amount of energy each task consumes in the chip by means of lightweight hardware mechanisms tracking activity and occupancy of the main resources in a per-task basis. In particular, we show how different tradeoffs provide increasing accuracy at the expense of higher hardware and energy cost.

- We show a use case where the proposed PTEM technique is applied to measure the per-task energy consumption for a parallel application.

Our results over a variety of multicore processor setups and workloads, including SPEC CPU 2006 benchmarks and traces from a real High-Performance Computing(HPC) application called *wrf*, show that a low-cost implementation of our PTEM mechanism achieves tight per-task energy measurements with respect to an ideal non-implementable model. For a 64-thread setup, 32 cores where each core is 2-way SMT, PTEM reduces the average accuracy error from more than 12% when evenly splitting energy over running tasks, to less than 4% when our low-cost hardware support is used. The maximum observed error for any task in the workload we used reduces from 58% down to 9% when our hardware support is used.

The rest of this chapter is organized as follows. Section 4.2 presents our idealized approach to perform per-task energy metering and the efficient hardware implementation. The particular experimental setup used in this chapter, intra-cluster results and full processor results are detailed in Sections 4.4. Next, Section 4.5 presents several case studies, including the characterization of the significant differences in energy and performance variability (Section 4.5.1), a large-scale parallel application study (Section 4.5.2), and other issues related to energy metering (Section 4.5.3). Finally, Section 4.6 draws the main findings of this work.

4.2 Ideal PTEM for the Multicore: LLC and Core

This section presents an idealized utilization-based model for per-task energy metering. The result of this model is later used as a reference point for our models to measure per-task energy at an affordable hardware cost. For the sake of clarity, we assume a single voltage level and that energy consumption does not change with temperature. In Section 4.5.3 we show how to extend our models to consider the impact in energy consumption of multiple voltage levels and temperature ranges.

We assume a clustered multicore architecture where each cluster consists of a set of cores, having each core private data and instruction first level caches, plus a shared on-chip second level cache accessed through a shared bus, see Figure 4.1. We refer to such cache as LLC. All clusters are connected to memory through a shared bus. We focus on the shared L2 caches, the core slice and the shared buses. The rest of the on-chip resources (e.g., I/O interface, etc.) have low contribution to total energy consumption [89], so we simply assume an even distribution of their energy consumption over running tasks, which has negligible impact on our estimation. If other components had significant contribution to the total energy of the chip, energy metering should be extended accordingly following the same principles as for the components analyzed in this work.

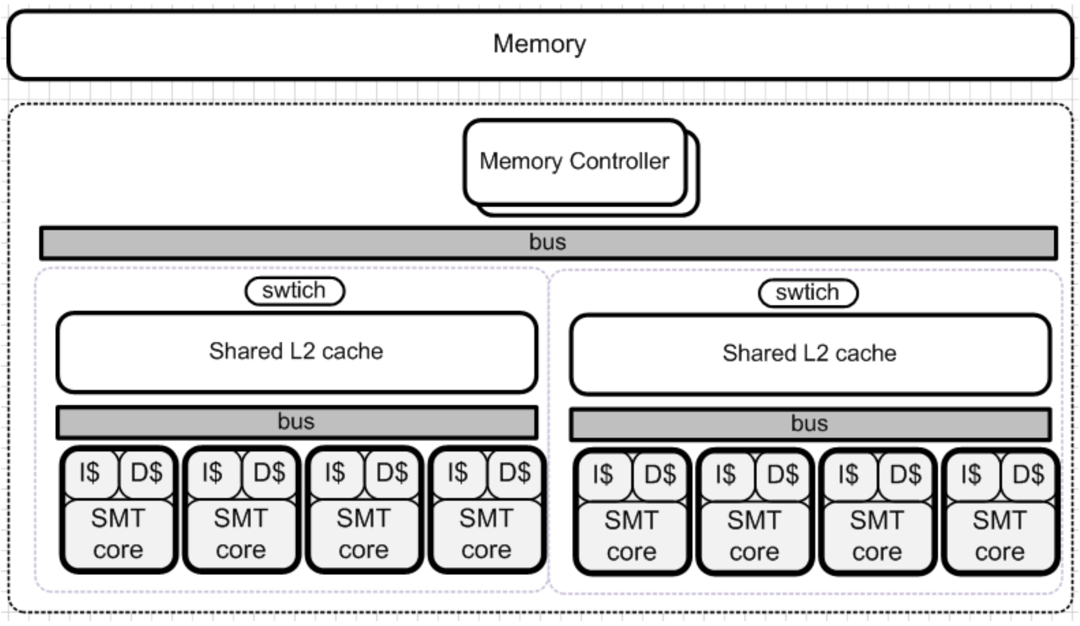


Figure 4.1: Diagram of high core-count cluster architecture

4.2.1 Shared Cache

The *active* energy consumption in the shared LLC for a given task i is proportional to the number of accesses. It can be computed as follows:

$$E_{act,total}^{LLC}(tk_i) = \sum_{k=1}^{K} \#action_k^{LLC}(tk_i) \times E_{action_k^{LLC}}^{LLC} \tag{4.1}$$

where $E_{action_k^{LLC}}^{LLC}$ stands for the energy per LLC access of type k, which is assumed to be available in this idealized model. $\#action_k^{LLC}(tk_i)$ stands for the number of LLC accesses of type k performed by the task i. Three main factors determine the access types we consider: whether an access reads or writes; hits or misses in LLC; and in the latter case whether it evicts a dirty line. The possible combinations are: read hit, write hit, read miss replacing a dirty line, read miss replacing a non-dirty line, write miss replacing a dirty line, and write miss replacing a non-dirty line. Under each combination of these factors, the energy consumption of an access can change. Extending the model to consider other access types (e.g., invalidations) is trivial since we only need to multiply the energy consumed by each access type by the number of those accesses.

Maintenance energy is consumed when resources are idle. We use cache occupancy as a proxy to measure maintenance energy: We assume that those cache regions (lines) not occupied by a given task could be turned off so that they would not incur any energy consumption [2]. The total maintenance LLC energy consumption for a task is obtained as follows:

$$E^{LLC}_{main,\ total}(tk_i) = Occ^{LLC}(tk_i) \times IdleTime(LLC) \times E^{LLC}_{main} \tag{4.2}$$

where $Occ^{LLC}(tk_i)$ stands for the average fraction of cache lines owned by task i, E^{LLC}_{main} corresponds to the maintenance energy per cycle consumed by the LLC when no access is performed, and $IdleTime(LLC)$ stands for the number of idle cycles for the LLC (no access to LLC). E^{LLC}_{main} is assumed to be provided under the ideal model.

Leakage energy is proportional to the cache occupancy and can be easily computed as follows:

$$E^{LLC}_{leak,total}(tk_i) = Occ^{LLC}(tk_i) \times ExecTime(tk_i) \times E^{LLC}_{leak} \tag{4.3}$$

where E^{LLC}_{leak} stands for the leakage energy per cycle consumed by the LLC. This value is also an input parameter for the idealized model.

4.2.2 Core Slice

Ideal per-task core energy metering requires tracking per-task activity in all core hardware blocks (e.g. Reorder Buffer, Issue Queues, etc) to count the number of accesses of each type. This would provide detailed information to accurately compute active energy by multiplying the per-type access counts by the active energy for each particular type of access (action):

$$E^{core}_{act,\ total}(tk_i) = \sum_{j=1}^{J}\sum_{k=1}^{K} E^{block_j}_{action_k} \times \#action^{block_j}_{k}(tk_i) \tag{4.4}$$

$E^{block_j}_{action_k}$ is the energy per action of type k (e.g., read) in block j (e.g., register file), which is assumed to be known. $\#action^{block_j}_{k}(tk_i)$ stands for the number of such actions on such block performed by task i. This applies to both single- and a multi-threaded (e.g., SMT) cores. J and K stand for the total number of blocks in the core and types of actions (e.g., read, write, flush) respectively.

Maintenance energy is measured in all those blocks having non-negligible energy consumption when no action is performed. Blocks can be classified into two categories depending on whether they allocate entries to tasks. Occupancy Blocks or *oblocks* allocate entries to tasks and hence their maintenance energy can be split based on the occupancy (e.g., precharge energy of first level caches). Conversely, in resources without memory or *eblocks* no entries are allocated, and hence maintenance energy can be evenly distributed (e.g., issue queue selection when there are no ready instructions). Maintenance energy is then computed as follows:

$$E^{core}_{main,\ total}(tk_i) = \sum_{x=1}^{ExecTime(tk_i)} \left(\sum_{j=1}^{J} \frac{E^{eblock}_{main}(x)}{\#Tk(C_k)} + \sum_{l=1}^{L} Occ^{oblock}(tk_i) \times E^{block_l}_{main}(x) \right) \quad (4.5)$$

where L stands for the number of *oblocks*, J for the *eblocks*, and $Occ^{block_l}(tk_i)$ for the average occupancy of block l by each task. $E^{block_l}_{main}(x)$ stands for the maintenance energy consumed by idle ports or in idle cycles of block l in cycle x. $\#Tk(C_k)$ stands for the number of tasks in core C_k.

Leakage energy can be easily tracked because it will be roughly constant throughout all the execution. If the core is single-threaded, then it is trivial to identify the owner of such energy. However, if the core is multi-threaded the occupancy per task in each of the blocks must be tracked to properly distribute leakage energy, as shown in the following equation:

$$E^{core}_{leak,\ total}(tk_i) = \sum_{j=1}^{J} Occ^{block_j}(tk_i) \times E^{block_j}_{leak} \times ExecTime(tk_i) \quad (4.6)$$

where $Occ^{block_j}(tk_i)$ stands for the average occupancy of block j by task i and $E^{block_j}_{leak}$ stands for the leakage per cycle of block j, which is assumed to be available.

4.2.3 Shared Bus

Ideal per-task bus energy metering requires tracking per-task accesses. Analogously to the case of the LLC, there are different types of accesses with different active energy consumption. For instance, if a cache line is sent over the bus, the energy consumed is higher than if just an address is sent, either because the cache line communication sends more bits simultaneously or because it requires several consecutive transactions to send all data over a bus narrower than a cache line. This would provide detailed information to accurately compute active energy by multiplying the per-type access counts by the active energy for each particular type of access (action):

$$E^{bus}_{act,\ total}(tk_i) = \sum_{k=1}^{K} E_{action_k} \times \#action_k(tk_i) \quad (4.7)$$

E_{action_k} is the energy per action of type k (e.g., cache line communication), which is assumed to be known. $\#action_k(tk_i)$ stands for the number of such actions performed by task i. K stands for the types of actions.

Note that different actions and energy per action values may be used for different buses such as the intra-cluster bus connecting cores to their LLC and the inter-cluster bus connecting cores to memory. Nevertheless, the same principle applies to compute active energy.

Leakage energy cannot be attributed to any particular task in the cores (tasks do not have any type of bus occupancy), so we evenly distribute it across all those tasks that could use the particular bus: tasks in the cluster for intra-cluster buses and tasks in the whole chip for the inter-cluster bus:

$$E^{bus}_{leak,\ total}(tk_i) = \frac{E^{bus}_{leak} \times ExecTime(tk_i)}{\#Tk(BUS_k)} \tag{4.8}$$

where E^{bus}_{leak} is the leakage energy per cycle of the bus, which is assumed to be known and $\#Tk(BUS_k)$ stands for the number of tasks in using bus BUS_k.

Note that, bus energy is dominated by active and leakage energy [66] due to wiring, repeaters and latches while maintenance energy is negligible. We evenly distribute maintenance energy over tasks.

4.3 An Implementable PTEM Approach

4.3.1 PTEM with Practical Approaches for the LLC

The ideal model for the LLC tracks two main per-task parameters: access (activity) counts per access type and cache occupancy. Our simplified PTEM model for the LLC relies on the fact that LLC accesses are not frequent, so they can be tracked with full accuracy. Conversely, tracking cache occupancy, which is required for maintenance and leakage energy estimation, would require counting how many cache lines each task owns every cycle, which is expensive. Tracking the ownership of cache lines requires: (1) tagging each cache line with a *task id*, (2) keeping a counter per task with the number of owned cache lines (*instant counter*), and (3) updating such counters on a replacement based on the ownership of the evicted and fetched cache lines, increasing the counter of the owner of the fetched line and decreasing the one of the owner of the evicted cache line.

In general, LLC access patterns and occupancy do not change abruptly. Similarly, the occupancy per set is quite homogeneous for any particular program [86]. Therefore, we propose sampling the LLC occupancy in two different 'directions'. First, only some cache sets will be monitored, so they will be the only ones for whom cache line ownership will be tracked. In order to avoid clustering effects due to contiguous allocation of data in memory for any particular task, sampled sets are located at a particular stride (e.g. only those sets whose x lowermost index bits are zero are monitored). How many x lowermost bits are considered depends on the desired sampling granularity. Second, the counters accumulating instant occupancy are not updated every cycle, but at a lower frequency.

For instance, for a LLC with 1,024 cache sets, 8 ways per set and a processor with 8 cores, cache sets can be sampled at a granularity of 1 out of 16, and time sampling occurs once every 256 cycles. In this case, the overhead of the LLC mechanism would be as follows:

- 8 instant counters (Occ_{inst}^{LLC}) of 10 bits each for tracking instant occupancy (1,024 sets x 8 ways / 16 sample granularity = 512 lines sampled, so 10 bits are needed).
- 512 3-bit owner identifiers for the 512 tracked cache lines. Note that all cache lines in the sampled sets always have an owner for energy metering purposes. Thus, on a context switch, the task being scheduled in becomes the owner of the cache lines used by the task being scheduled out (using the same hardware context, or CPU index).
- 8 cumulated occupancy counters (Occ_{cum}^{LLC}) of 48 bits able to track the occupancy during $2^{48} \times 2^{8} = 2^{56}$ cycles (48-bit counters and 256 cycles sampling frequency).

We assume that the number of cycles that a program takes to run is measured by an existing performance monitoring counter of the processor. Based on this hardware support LLC occupancy is obtained as follows:

$$Occ^{LLC}(tk_i) = \frac{Occ_{cum}^{LLC}(tk_i) \times SmpFreq \times SmpSets}{\#Sets^{LLC} \times ExecTime(tk_i)} \tag{4.9}$$

where $SmpFreq$ is the sampling frequency (256 cycles in the example), $SmpSets$ is the set sample granularity (16 in the example) and $\#Sets^{LLC}$ is the number of total cache sets (1,024 in the example). The impact of sampling in both time and sets is later analyzed in the evaluation section.

4.3.2 PTEM with Practical Approaches for the Core

Current processors, e.g. the IBM POWER7 [32, 48], can estimate the energy consumed by each core (even for SMT cores) based on a model that uses as proxy different performance monitoring counters, voltage, frequency and temperature. However, solutions to accurately distribute core energy across tasks in SMT cores have not been developed, while, in fact, multicores with SMT cores are becoming quite common [32, 104].

A real per-task core energy metering, cannot be done with the ideal model presented before since this models tracks too many events and the occupancy of many blocks. Instead of such a bottom-up model, PTEM builds a top-down model. Under this top-down model, during the execution of a workload we first breakdown the energy consumed into its main components, *active*, *maintenance* and *leakage* energy; and in a second step, we breakdown the energy of each component per task.

Step 1: Deriving active, maintenance and leakage energy components. We start determining the maximum power (P_{max}^{core}) and minimum power (P_{min}^{core}) dissipation in a given time interval.

The core maximum power dissipation, P_{max}^{core}, can be determined by running a high-power benchmark, a.k.a. power virus [88]. P_{max}^{core} can be decomposed as follows:

$$P_{max}^{core} = MaxDynP^{core} + LeaP^{core} \tag{4.10}$$

$MaxDynP^{core}$ is the maximum *dynamic power* of the core and $LeaP^{core}$ the *leakage* power of the core that can be obtained by measuring core power when the core is in *halt mode*. In this formula, we assume that all blocks are fully used so no *maintenance* power is dissipated. In reality, there will be still some *maintenance* power, but its relative weight with respect to *active* power is negligible in a maximum power scenario, so the loss of accuracy introduced by such an assumption is rather low.

The core minimum power dissipation, P_{min}^{core}, can be obtained running a low-power benchmark comprised, for instance of *no-ops*. P_{min}^{core} can be decomposed as follows, where $MaxMainP^{core}$ is the maximum *maintenance* power of the core:

$$P_{min}^{core} = MaxMainP^{core} + LeaP^{core} \tag{4.11}$$

In this formula we assume that all blocks are idle so that no *active* power is dissipated. Under that scenario, all activity in the core incurred *maintenance* power dissipation as these activities are not produced by tasks' execution. This is the scenario in which the *maintenance* power is the highest, $MaxMainP^{core}$. From Equations 4.10 and 4.11, we can derive $MaxDynP^{core}$ and $MaxMainP^{core}$.

Let's assume that the energy consumed by a workload during an interval T is $E^{core} = (LeaP_{core} + DynP_{core} + MainP_{core}) \times T$. In order to determine which fraction of $E^{core}(T)$ is *active*, *maintenance* and *leakage* we proceed as follows. Leakage power is roughly constant in all runs, so we take the value derived above, $LeaP^{core} \times T$.

We assume that all idle blocks have the same *maintenance* energy consumption when idle w.r.t. their *active* energy consumption. That is, for all blocks the relation between *active* and *maintenance* power is obtained as $MainDynRatio = \frac{MaxMainP^{core}}{MaxDynP^{core}}$. Hence, the *maintenance* energy for each block during a time interval is $MainDynRatio$ of its *active* energy.

During the execution of a workload in a given interval, a fraction of the resources will perform useful activity, thus consuming *active* energy in the interval ($DynE_j^{core}$). The remaining resources do not perform any useful activity consuming *maintenance* energy. The difference $(MaxDynP^{core} - DynP_j^{core}) \times T$ provides the amount of *active* energy not consumed in the execution of the workload with respect to the scenario in which the *active* energy is maximum. The maintenance energy $MainE_j^{core}$ is a fraction of that difference: $MainE_j^{core} = (MaxDynP^{core} - DynP_j^{core}) \times MainDynRatio \times T$.

Overall, E_j^{core} can be derived as follows:

$$\begin{aligned} E_j^{core} &= LeaE^{core} + DynE_j^{core} + MainE_j^{core} \\ &= LeaE^{core} + DynE_j^{core} + (MaxDynE^{core} - DynE_j^{core}) \times MainDynRatio \end{aligned} \tag{4.12}$$

where only $DynE_j^{core}$ is unknown and can, therefore, be derived.

Step 2: Breaking down active, maintenance and leakage energy components per task. Per-task energy distribution is done as follows:

- **Dynamic active energy**. Since tracking all events in the core is unaffordable, we use a simplified model based on the number of instructions fetched per task.
- **Dynamic maintenance energy**. Most maintenance energy in the core comes from register files and issue queues due to their large number of ports and high maintenance energy consumption per port. Such energy cannot be attributed to any particular task, so we evenly split maintenance energy across tasks.
- **Leakage energy**. Leakage energy mainly comes from first level (L1) caches and their occupancy correlates quite well with the occupancy of some other blocks (e.g., branch predictor tables, translation lookaside buffers). Thus, we track task occupancy in L1 caches. We need the same hardware support as in the LLC. We consider that L1 data and instruction cache occupancies have the same weight.

Therefore, task energy in the core is measured as follows for interval j:

$$DynE_j^{core}(tk_i) = DynE_j^{core} \times InstFetch_j(tk_i)/InstFetch_j \tag{4.13}$$

Table 4.1: PTEM hardware requirements

Block	Energy figures	Extra Logic
LLC	E^{LLC}_{action}, E^{LLC}_{main}, E^{LLC}_{leak} ,	$\#action^{LLC}_k(tk_i)$, $Occ^{LLC}_{inst}(tk_i)$, $Occ^{LLC}_{cum}(tk_i)$, $IdleTime(LLC)$, LLC Cache line owner's table
Core	E^{core}_{max}, E^{core}_{min}, $LeakE^{core}$	$InstFetch$, $InstFetch(tk_i)$, $Occ^{IC}_{inst}(tk_i)$, $Occ^{IC}_{cum}(tk_i)$, $Occ^{DC}_{inst}(tk_i)$, $Occ^{DC}_{cum}(tk_i)$, IC Cache line owner's table, DC Cache line owner's table, $E^{core}_{total}(tk_i)$
intra-cluster bus	E^{inbus}_{action}, E^{inbus}_{leak}	$\#action^{inbus}_k(tk_i)$
inter-cluster bus	E^{outbus}_{action} , E^{outbus}_{leak}	$\#action^{outbus}_k(tk_i)$

$$MainE^{core}_j(tk_i) = MainE^{core}_j/\#Tk \tag{4.14}$$

$$LeaE^{core}_j(tk_i) = LeaE^{core}_j \times \frac{Occ^{IC}(tk_i) + Occ^{DC}(tk_i)}{2} \tag{4.15}$$

where $InstFetch_j$ and $InstFetch_j(tk_i)$ are the total and task i fetched instructions in interval j respectively. $Occ^{IC}(tk_i)$ and $Occ^{DC}(tk_i)$ stand for the task i occupancy in the data and instruction caches. Then, we only need to cumulate the energy of the task across all intervals:

$$E^{core}_{total}(tk_i) = \sum_{j=0}^{\frac{ExecTime(tk_i)}{SmpFreq}} (DynE^{core}_j(tk_i) + MainE^{core}_j(tk_i) + LeaE^{core}_j(tk_i)) \tag{4.16}$$

4.3.3 PTEM with Practical Approaches for the Buses

The ideal model for the buses only needs to track access (activity) counts per access type per task. Our simplified PTEM model for the buses relies on the fact that, analogously as for the LLC, bus accesses are not frequent, so they can be tracked with full accuracy. Also, leakage energy is tracked trivially by considering how many cycles each thread runs and how many threads share each bus.

4.3.4 Putting It All Together

The practical PTEM approaches require reduced hardware overhead. PTEM mostly requires setting up some counters similar to the PMCs currently available in most high-performance processors. PTEM support, analogously to PMCs, does not interfere the execution of programs since it is not in any critical path.

Table 4.1 summarizes those parameters required from the chip vendor and the extra logic (counters, tables) that must be set up. The chip vendor is required to provide only few parameters that can be either obtained by running appropriate benchmarks or estimated using test chips or power models. Note that counters with the (tk_i) suffix must be replicated for each task. Analogously, *action* in the case of the LLC stands for the 6 different LLC actions considered in this work: read/write hit, read/write miss (no dirty line replaced), read/write miss (dirty line replaced), and for the 2 different bus actions considered in this work in the case of the buses: address communication and cache line communication. *Inbus* and *outbus* refer to the intra-cluster and inter-cluster buses respectively in the table.

Regarding the interface with the software, the OS is responsible for keeping track of the energy consumed by every task running in the system. PTEM exports a special register, called Energy Metering Register (EMR), that acts as the interface between PTEM and the OS. The OS can access that register for collecting the energy estimates made by PTEM. This typically will happen when a context switch takes place. At that moment, the OS will read the EMR using the hardware-thread index (or CPU index) for the task that is being scheduled out (T_{out}). Then, the OS will aggregate the energy consumption value received in the *task struct* for T_{out}. Right after the new task (T_{in}) is scheduled in, the LLC and L1 caches will continue to contain some lines belonging to T_{out}. These lines will be tagged with the same identifier as the one T_{in} is using. Although, PTEM will attribute maintenance and leakage energy consumption to T_{in}, we have empirically observed, that this occurs during less than 1 million cycles, since cache lines belonging to T_{out} will be quickly replaced and thus, evicted from LLC. Under a processor frequency of 2GHz, 1 million cycles are equivalent to $0.5\mu s$, while context switches occur at much higher granularity, every 10-$100\mu s$.

4.4 Evaluation

4.4.1 Experimental Setup

The general experimental setup used in this chapter is as introduced in Section 3.2, except that we have also taken into account large core-count scenarios. For these setups, we assume a clustered multicore architecture, as shown in Figure 4.1, where each cluster consists of a set of cores, having each core private data and instruction first level caches, plus a shared on-chip second level cache accessed through a shared bus. We refer to such cache as LLC. All clusters are connected to memory through a shared bus. Several studies show that hierarchical bus configurations scale quite easily to large systems and provide a good area-performance trade-off, while retaining many of the advantageous features of simpler bus arrangements [100]. In the same line, other studies show that bus-based networks can significantly lower energy consumption and simplify network protocol design and verification, with no loss in performance [111].

In order to evaluate the accuracy of PTEM, we make use of the benchmark suite and workload generation strategy introduced in Section 3.4. We also consider an HPC application, wrf, as described in Section 3.4.2. To measure accuracy, the make use of the metric described in Equation 3.1.

4.4.2 Intra-cluster evaluation

We evaluate the accuracy of our hardware support for per-task energy metering incrementally by analyzing the accuracy at intra-cluster level. Once we analyze the accuracy of the PTEM models for the cache and SMT core, in next section we show the results when we scale the number of cluster to sum up a total of 16/32 cores (32/64 threads). Due to the relatively low energy contribution of buses, intra-cluster bus energy is reported as part of the LLC energy.

The key idea of our per-task energy metering approach is to make the energy attributed to a task proportional to each resource *utilization*. In particular, to its *activity* and the *occupancy* of a given resource. If both activity and occupancy are accurately measured, the energy consumption can be accurately attributed to each running task.

Figure 4.2 shows the fraction of LLC energy consumption attributed to each benchmark in a 2-core workload (*gcc*+*mcf*), by using our ideal model presented in Section 4.2.

We observe that the activity does not necessarily reflect the occupancy of the LLC. In the figure, we can see that *gcc*, with 63.5% accesses, occupies less than 46.2% of LLC lines. That shows that a given workload may have very different consumption profiles in terms of active energy versus maintenance and leakage energy. Therefore, it is important to measure both activity and occupancy in order to improve the estimation accuracy. For instance, let us look again at the *gcc* case. If we estimate the energy only proportionally to the activity, LLC energy will be significantly overestimated for *gcc* and underestimated for *mcf*.

4.4.3 PTEM Energy Estimation

In this section we show the accuracy of the models presented in Section 4.2 for the core and the LLC at cluster level. In particular we measure the off estimation of each model

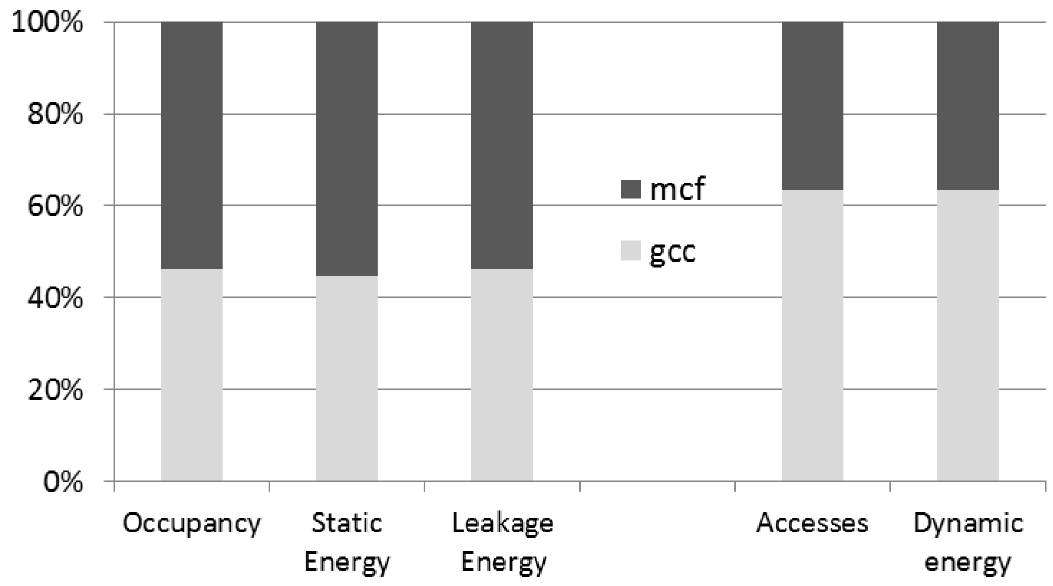


Figure 4.2: Per-task LLC cache energy breakdown and access/occupancy rates when executing *mcf* and *gcc* in a single-threaded 2-core configuration.

with respect to the idealized model. We include the *ES* model that uniformly splits the energy among all running tasks regardless of their occupancy and activity in the processor resources. This is indeed the common approach in current methods only considering execution time.

Core Energy Consumption Prediction: Figure 4.3 shows the prediction accuracy for the core under the setup C4S2. Each bar shows the average error of all 8 benchmarks in the workload.

In general, PTEM clearly outperforms ES providing tighter energy predictions. In particular, PTEM incurs a prediction error of up to 6.9% across workloads, while for the *ES* model it is higher than 13%. Predictions are more accurate for *I* workloads due to the highly homogeneous behavior of programs. Irregular workloads in *X* and *M* groups (some benchmarks are more memory-bound than others in the *M* group) lead to slightly higher error for PTEM and larger error for the *ES* model. This can be also seen when comparing the maximum error across individual tasks in the workloads (see Table 4.2). PTEM maximum error is highly constant across workload types (9-10%) whereas *ES* model error is particularly high for *X* and *M* workloads (28% and 22% respectively).

LLC Energy Consumption Prediction. Figure 4.4 shows the effect of sampling sets and period on the average LLC energy prediction accuracy for a 4-core configuration. The y-axis represents the sample period measured in processor cycles (e.g., 10K stands for 10,000 cycles). The x-axis is the sampling set configuration. For instance, *1e8* means

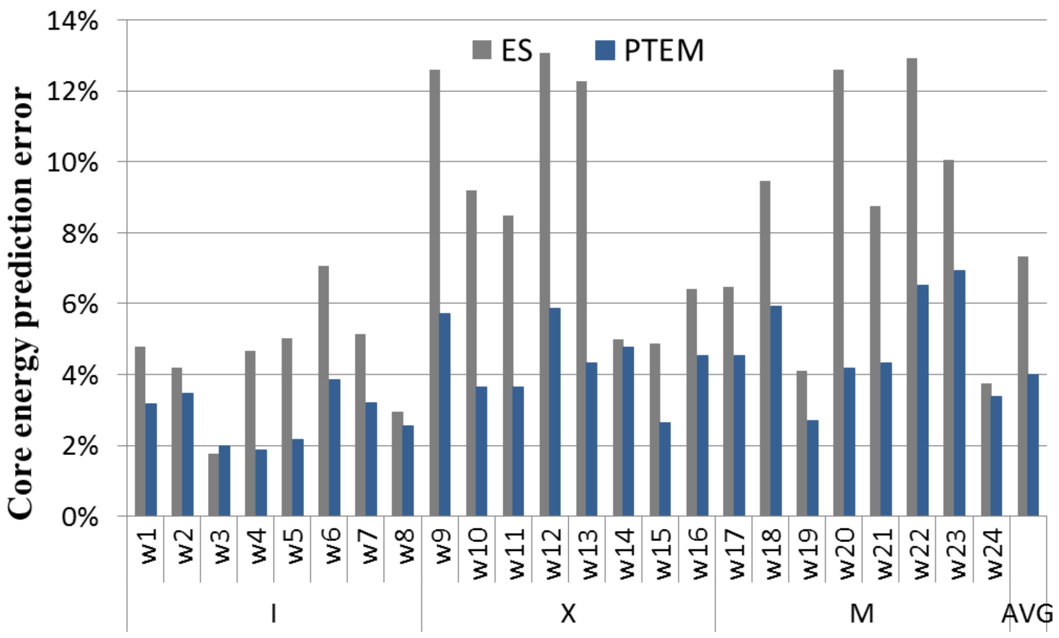


Figure 4.3: Per-task core energy prediction error rate (C4S2)

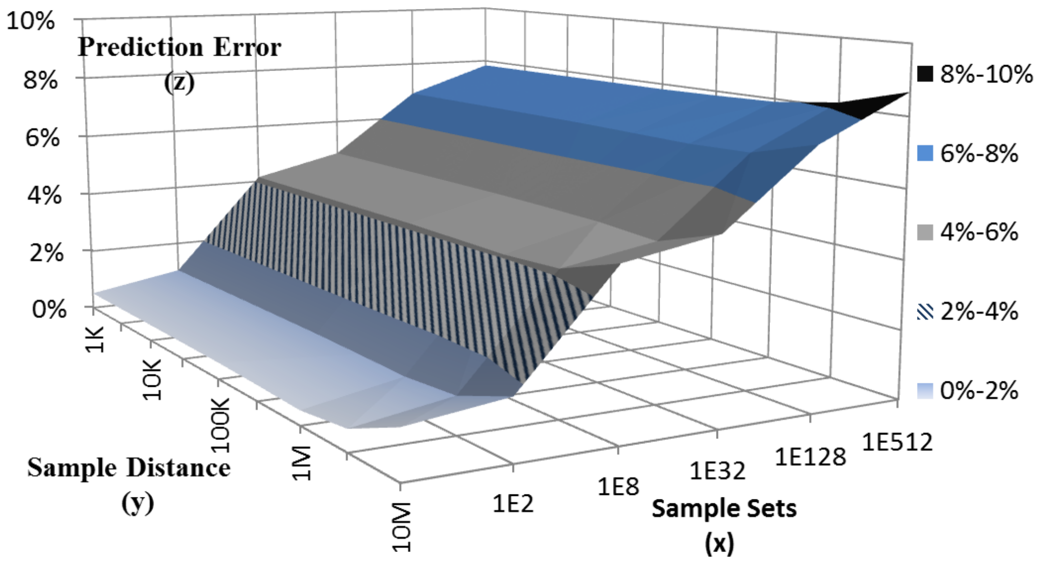


Figure 4.4: Per-task LLC cache energy prediction with sample set and period in a 4-core configuration.

that we sample 1 set every 8 sets.

We observe that the curve has a higher slope in the x-axis (set sampling). For instance, for a sampling distance of 10K cycles, the prediction error rate raises from less than 1% to almost 8% as the sample set reduces from 1e1 to 1e512 sets. Instead, the sample period

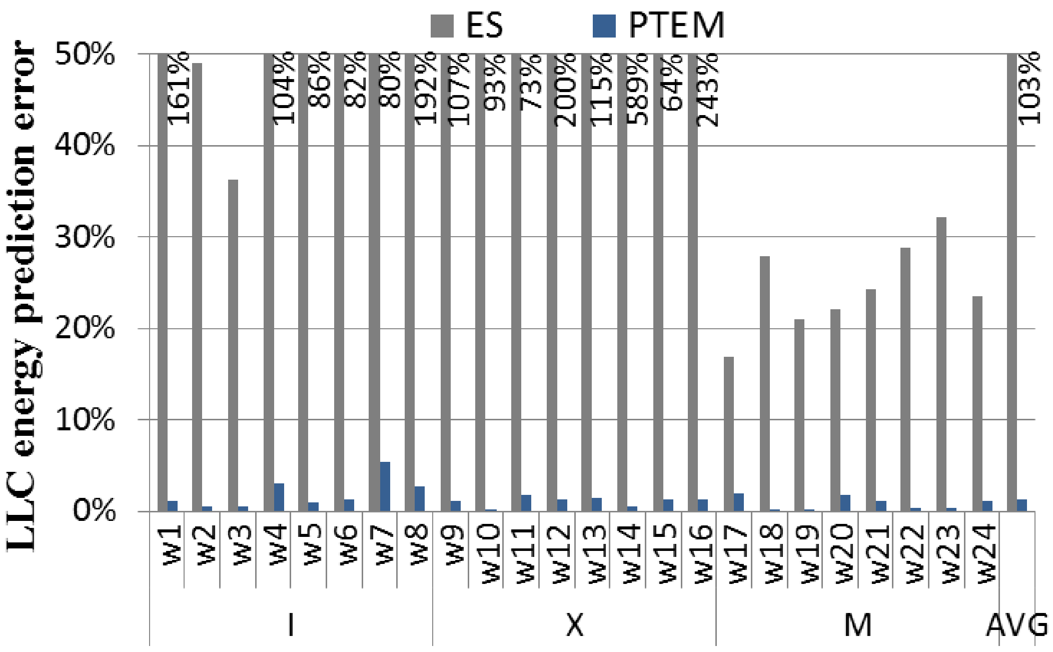


Figure 4.5: Per-task LLC cache energy prediction error rate (C4S2)

(y-axis) has limited effect on accuracy. With 1e8 sampled sets, the prediction error only raises 0.2% as the period increases from 1K to 10M cycles.

Considering that the hardware cost of set sampling varies significantly, we choose a moderate-cost configuration in which we use 1e2 and 10K cycle sampling period. This is the configuration we use to measure the energy per-task in the LLC in the following sections.

Figure 4.5 shows the LLC prediction error of each model under the C4S2 setup. Prediction error corresponds to the average error across benchmarks in each workload. We observe that PTEM largely outperforms *ES* model in terms of accuracy for all workloads and processor setups.

The *ES* model is highly inaccurate in general, more than 103% on average. The *ES* model accuracy is worse for *I* and *X* workloads due to the highly heterogeneous memory behavior of the tasks. In fact, even in *I* workloads behavior is highly heterogeneous because the relative LLC access frequencies and occupancies are very different across tasks. *ES* accuracy improves for *M* workloads where LLC occupancy and access frequency are more homogeneous. Our PTEM model, in contrast, has a considerably low prediction error, less than 2% on average. Further, as shown in Table 4.2, maximum error across all tasks for PTEM is 25.6% for *I* workloads because their low LLC utilization may make spatial sampling to experience some error. However, as long as *M* tasks are in place (*X* and *M*

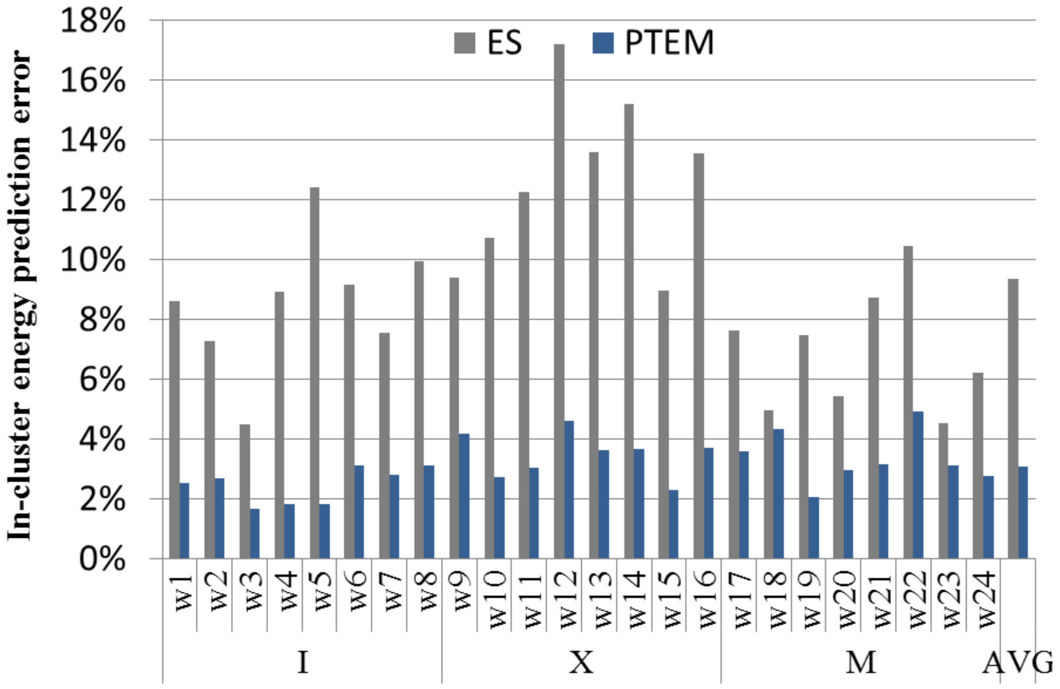


Figure 4.6: Cluster per-task energy prediction error (C4S2)

workloads), PTEM accuracy is very high (maximum error is always below 4.5%). On the other hand *ES* model error is huge (more than 3000%), especially for *I* and *X* workloads due to the highly heterogeneous memory behavior of the tasks in the workloads.

Cluster Energy Consumption Prediction. Next, we show per-task energy metering accuracy at cluster level, including core and LLC energy.

Figure 4.6 shows the average prediction error in each workload for a cluster consisting of 4 2-way SMT cores. First, we observe that prediction error for the whole cluster is very similar to that of the cores only (see Figure 4.3). This is so because the LLC energy contribution is typically in the range 15-20% due to the high activity of the cores (8 threads running). Therefore, core prediction error dominates the overall prediction error. As expected, the *ES* model obtains worse results than PTEM in all workload groups, with an average above 10%. The prediction error for PTEM is less than 3% on average across all workloads. Furthermore, we observe that the *ES* model error grows for *X* workloads since different threads perform highly heterogeneous activities. *ES* model average error is above 17% for one of the workloads. Instead, PTEM error remains quite stable across workloads and never exceeds 4.5%.

Per-benchmark data in each workload show that the maximum off-estimation that PTEM produces is 9.2% for one of the benchmarks in the *X* workloads, see Table 4.2

Table 4.2: Maximum per-task prediction error.

Core			
	I	X	M
PTEM	8.8%	9.6%	10.2%
ES	11.9%	28.3%	21.9%
LLC			
	I	X	M
PTEM	25.6%	4.0%	4.4%
ES	1112.6%	3593.8%	62.0%
Cluster			
	I	X	M
PTEM	6.6%	9.2%	7.5%
ES	25.8%	58.5%	23.6%

(recall that we use 8-benchmarks workloads and evaluate 24 different workloads, counting 192 benchmarks in total). For homogeneous workloads (*I* and *M*), the maximum error observed is 7.5% only. Instead, the maximum error for the *ES* model is 58.5%. Maximum error is lower for homogeneous workloads, but still in the order of 3-4x that of our PTEM model.

4.4.4 PTEM Energy and Area Overhead

PTEM requires few hardware counters to track LLC, core and bus activity, together with small arrays tracking the ownership of some cache sets in the LLC and L1 caches. For the sake of consistency, the energy of those components has been modeled using CACTI. In order to model counters, components such as internal cache buffers have been used, since they are comparable to latches in the pipeline.

Results for the 4-core 2-way SMT configuration show that the total energy overhead for PTEM is below 0.3%. Most of the overhead is due to the active energy of the ownership id arrays in LLC and L1 caches. Relative overheads do not change noticeably for different core counts. In fact, the relative overhead slightly decreases as the number of cores increases, which proves that PTEM scales well.

We have obtained the area overhead using CACTI with the following assumptions: LLC cache occupies 50% of the area in a 8-core configuration, counter bitcells have the same size as input/output buffers in caches (so they are large) and ALUs performing power computations use low-cost designs such as iterative multipliers and dividers (their latency is not critical as they are used seldom). We consider SMT cores, as they require more bits to track ownership and more counters to track per-task activity. Overall, we obtained that total area overhead is 0.49% (4 cores), 0.63% (8 cores), 0.75% (16 cores) and 0.82% (32 cores), proving that PTEM area cost is rather low. The area breakdown for the

32-core configuration is 0.20% LLC, 0.48% DL1+IL1, 0.09% core without DL1/IL1 and 0.05% bus. Similarly, the breakdown for the 4-core configuration is 0.22% LLC, 0.20% DL1+IL1, 0.04% core without DL1/IL1 and 0.03% bus. Thus, those arrays tracking the cache line ownership and counters tracking per-task activities in caches account for most of the area overhead, which anyway is rather low.

Overall, PTEM imposes neither limitations on the number of threads that can be run simultaneously in the processor (low and scalable hardware overheads), nor limitations on the number of tasks the OS can keep active simultaneously (a single counter per task needs to be tracked by the OS).

4.4.5 PTA Model Justification

Since *PTA* models have been widely used to estimate core and system-level energy [7, 14, 102], we also include PTA in our discussion. While these models typically rely on existing PMCs, so no extra hardware support is needed, their accuracy is limited and highly dependent on whether training workloads are similar to those at deployment.

Coefficients of the *PTA* model are obtained using our idealized model as the reference model, since no other reference model exists. We provide the linear regression with all per-task event counters in our simulator including number and type of instructions fetched, executed, committed, data and instruction cache hits and misses, etc. despite PMCs may not exist for many of those events.

We have used a 4-core 2-thread SMT setup. The training set consists of a workload with eight benchmarks randomly chosen from the SPEC CPU 2006 for each of the three categories described before: I, X and M. The evaluation workload consists of eight workloads generated analogously for each category.

As shown in Figure 4.7, the *PTA* model performs worse than PTEM. Linear regression is less accurate than *ES* for I workloads, and slightly more accurate for X and M ones. The average error for the *PTA* model is 7.8%, similar to *ES* one. Furthermore, we have observed that maximum estimation error is higher for the *PTA* model than for PTEM and *ES*. The reason for those large estimation errors for the *PTA* model is twofold: (i) its dependence on the training set and (ii) the fact that PMCs do not take into account occupancy, which is the parameter determining per-task leakage and maintenance energy in many components.

Finally, although not shown, results for other components (e.g., LLC) show similar trends because of the same limitations pointed out for the core. For instance, Figure 4.2 shows the dependence of LLC leakage and maintenance energy on occupancy rather than

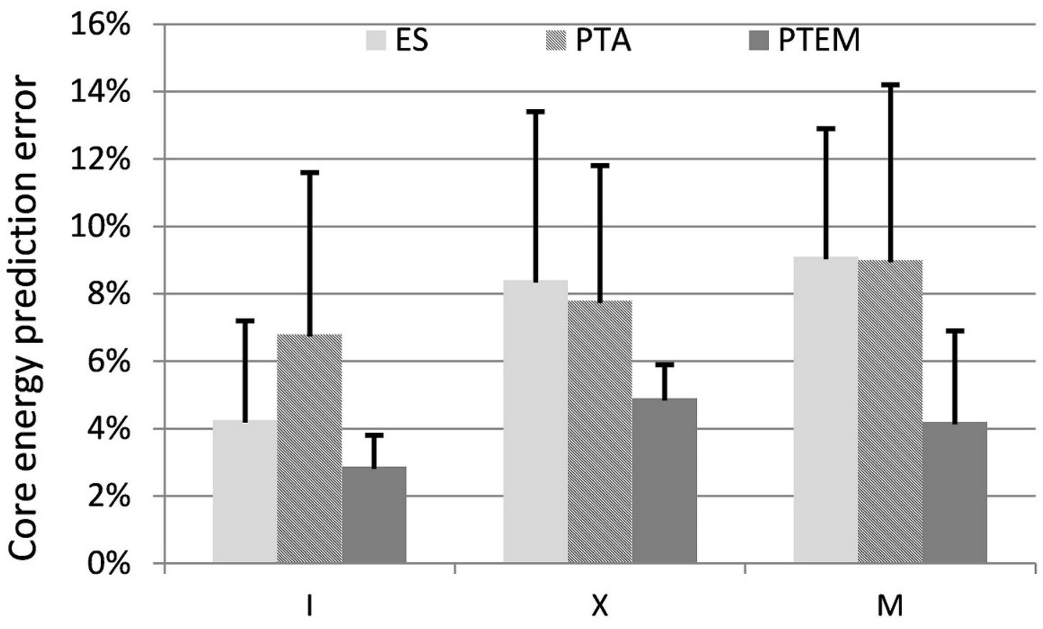


Figure 4.7: Core per-task energy prediction (C4S2) with ES, PTA and PTEM model, including the average error and maximum error.

on accesses.

4.4.6 PTEM for High Core-Counts

In this section we evaluate the accuracy of our PTEM model for large multicores with 4 and 8 clusters, counting 16 and 32 2-way SMT cores respectively. For that purpose, we have run experiments with 4 different types of workloads: pure *I* workloads, pure *M* workloads, *X* workloads (with 4 *I* and 4 *M* tasks per cluster) and hybrid workloads where half of the clusters run pure *I* workloads and the other half runs pure *M* workloads.

Memory bandwidth for the 8-cluster configuration has been increased by setting up 2 memory controllers instead of one able to issue memory commands in parallel as long as they do not conflict in any particular bank. This has been done in order not to overdesign memory bandwidth for the 4-core setup and not to underdesign memory bandwidth for the 8-core setup. The behavior of the different workloads is such that the relative execution time increase is low with respect to the single cluster setup since little memory contention is suffered in *I* tasks, and the higher contention paid by *M* tasks is still low in relative numbers.

Results in Figure 4.8 show that PTEM achieves higher accuracy for pure *I* workloads and hybrid *I*-*M* workloads. This is so because, as shown before, PTEM achieves higher

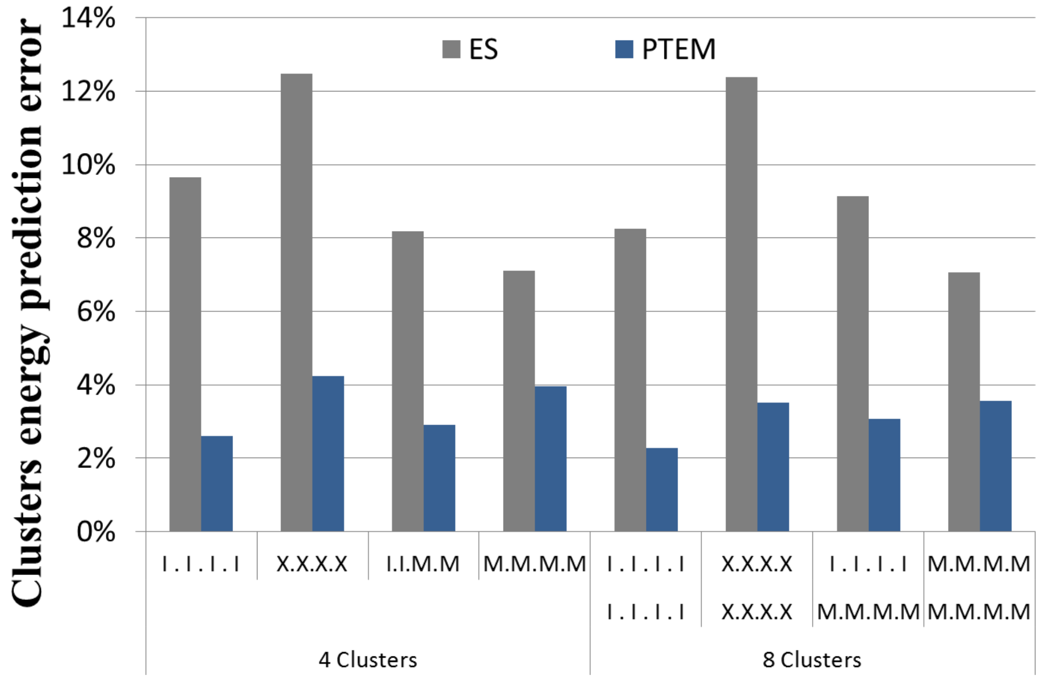


Figure 4.8: System per-task energy prediction error

accuracy for pure I clusters than for X or M ones. Instead, pure M and X configurations show slightly higher prediction error. Nevertheless, the average error is low and, in the case with the largest core count (8 clusters), the average error is below 3.5% regardless of the workload type. In the case of the ES model, prediction error is significantly higher than that of PTEM, being above 12% on average for X workloads. Other configurations show lower error since they mitigate the per-core prediction error. Nevertheless, PTEM largely improves accuracy with respect to the ES model across cluster counts and workload types, and opposed to the ES model, PTEM error decreases as the cluster count increases.

4.5 Case Study

4.5.1 Characterization of Energy and Performance Variation

Interferences among co-running tasks when accessing shared hardware resources in a multicore (a.k.a. inter-task interferences) result in different per-task performance depending on its co-runners [31]. In this section, we use our proposed PTEM model to show how the energy consumption of each task also significantly varies due to inter-task interferences, and *prove that such energy consumption variation cannot be directly inferred from performance variation.*

We focus on 2-task workloads, which we run in a 2-way SMT core setup of our baseline configuration. We construct all possible pairs of benchmarks from SPEC CPU 2006 suite, recording for each benchmark its energy consumption in each of the 2-task workloads in which it runs. The variation that each benchmark suffers across each 2-task workload is illustrated in terms of Cycles Per Instruction (CPI) in Figure 4.9 and in terms of Energy Per Instruction (EPI) in Figure 4.10. Results have been normalized with respect to the average CPI and EPI respectively for the sake of readability since CPI ranges between 1.03 and 11.36 cycles/instr, and EPI between 0.29 and 2.99 nJ/instr. Benchmarks are sorted from lowest to highest CPI.

We observe that CPI variation mostly concentrates in the range [+20%,-40%] w.r.t. their average for most of the benchmarks, whereas EPI concentrates in the range [+30%,-20%]. Hence, in both cases variations are significant and, therefore, we can conclude that performance and energy consumption strongly depends on the co-runners. In terms of performance variation, MEM benchmarks (*mcf*, *milc*, *lbm*, *libquantum*, *soplex*, *gcc*, *bwaves* and *omnetpp*) are among those with the lowest performance variation. For instance, *lbm* and *omnetpp*, both in MEM category, are the ones exhibiting the lowest performance variation across all benchmarks.

However, in terms of EPI this is not the case: Typically, EPI variation for ILP benchmarks decreases while MEM benchmarks have higher EPI variation than CPI variation. For instance, *libquantum*, which falls in the MEM category, is the benchmark exhibiting highest EPI variation. Analogously, *mcf*, *soplex*, *gcc* and *omnetpp* also experience a significantly higher variation increase in terms of EPI than CPI. In contrast, *astar* has a a significant variation in CPI, but reduced variation in EPI. Thus, the relation between performance and energy variation is non-obvious. We note that the two benchmarks in the middle of x-axis, *astar* and *perlbench*, both of them being ILP, have opposite trends across metrics: EPI variation for *astar* is much lower than its CPI variation. Conversely, EPI variation for *perlbench* is much higher than its CPI variation

We have also studied absolute EPI and CPI values, shown in Table 4.3. Values are sorted based on their CPI. We observe that MEM benchmarks have higher CPI than ILP ones, since they access memory more often and thus, experience higher latencies. Few ILP benchmarks have higher CPI than some of the MEM ones. Such higher CPI for MEM benchmarks translates into higher average EPI. In fact, only *zeusmp* (ILP) has slightly higher EPI than one of the MEM benchmarks (*bwaves*). The main reason for the increased EPI of MEM programs is the fact that they execute longer and occupy more resource space, which translates into higher static and leakage energy. However,

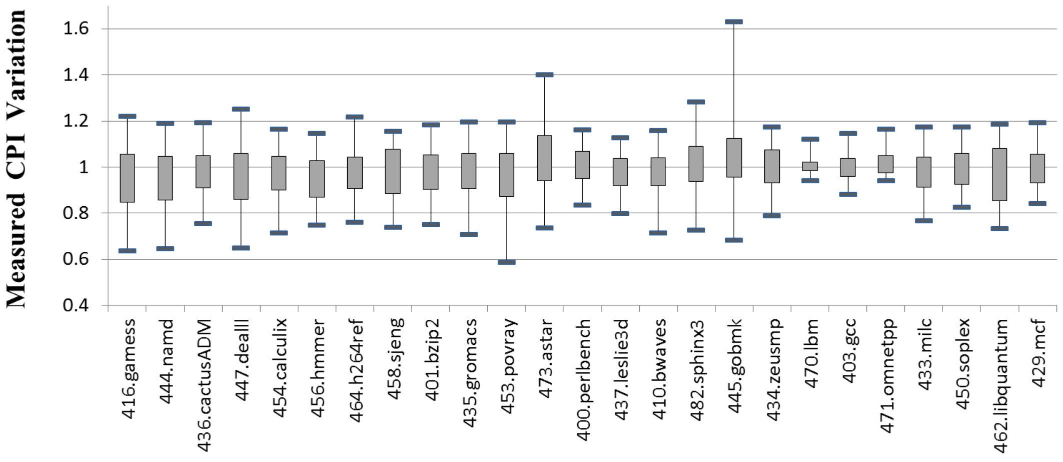


Figure 4.9: Per-benchmark CPI variation across all 2-task workloads in which the benchmark runs. Benchmarks sorted in increasing average CPI. Chart shows max, min, higher-quartile and lower-quartile values.

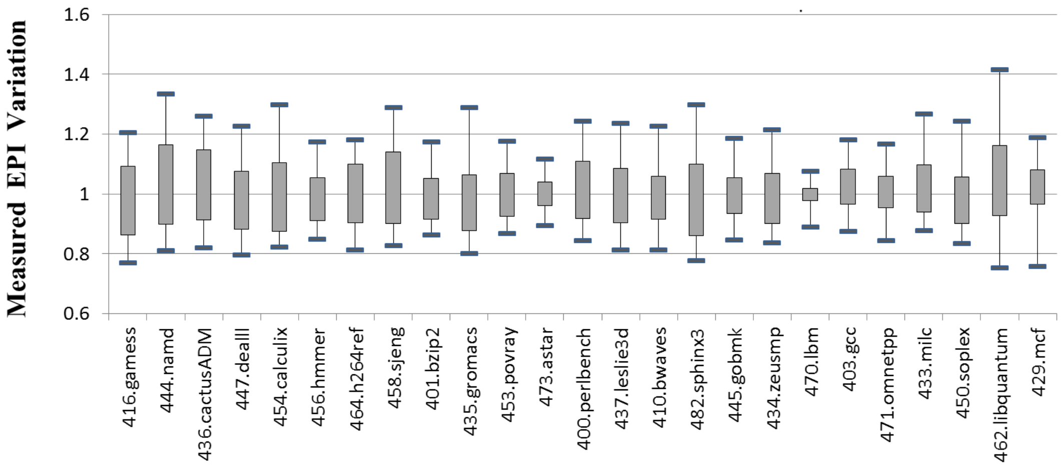


Figure 4.10: Per-benchmark EPI variation across all 2-task workloads in which the benchmark runs.

the particular interaction across different programs in shared resources leads to different behavior in terms of performance and energy, as it has been shown in Figures 4.9 and 4.10 in terms of CPI and EPI variability. Therefore, *performance cannot be used as a suitable metric to derive per-task energy consumption*, quite the opposite, these results confirm that our proposed energy metering technique, PTEM, is required to achieve accurate per-task energy metering.

Table 4.3: Average CPI (cycles/instr) and EPI (nJ/instr) for all benchmarks, sorted from lowest to highest average CPI from left to right and from top to bottom. MEM benchmarks in bold font.

	416.gamess	444.namd	436.cactusADM	447.dealII	454.calculix
CPI	1.03	1.03	1.05	1.05	1.10
EPI	0.35	0.29	0.40	0.36	0.32

	456.hmmer	464.h264ref	458.sjeng	401.bzip2	435.gromacs
CPI	1.22	1.23	1.24	1.28	1.30
EPI	0.40	0.48	0.37	0.46	0.41

	453.povray	473.astar	400.perlbench	437.leslie3d	**410.bwaves**
CPI	1.34	1.34	1.41	1.42	1.46
EPI	0.41	0.50	0.44	0.50	0.55

	445.gobmk	482.sphinx3	434.zeusmp	**470.lbm**	**403.gcc**
CPI	1.51	1.51	1.60	2.19	2.58
EPI	0.49	0.50	0.56	0.92	0.89

	471.omnetpp	**433.milc**	**450.soplex**	**462.libquantum**	**429.mcf**
CPI	2.83	3.22	3.88	4.90	11.36
EPI	0.95	0.89	1.14	1.26	2.99

4.5.2 Large-Scale Parallel Applications

4.5.2.1 Adapting PTEM to Multithreaded Applications

In our per-task energy measuring approach, the energy accounted to each thread is saved into a special purpose register per thread, denoted EMR. Section 4.3.4 shows how the OS handles the EMR of each task.

The support required for PTEM in the case of multithreaded applications is simple. In fact, no hardware changes in the PTEM logic are actually required, but only on how the OS handles the EMR: The OS or the parallel runtime, simply needs to aggregate the energy consumption estimates stored for all the threads belonging to the same multi-threaded application. $Emeter_{App} = \sum_{i=1}^{N} EMR_i$ where N is the number of threads of the application. When a cache line is shared in the LLC across different threads, its energy (static and leakage) must be accounted once, either by splitting it across the threads sharing it or by attributing it to one of those threads. In particular, we identify as owner the thread fetching the cache line to the LLC. Whether this energy is attributed to one thread or another of the parallel application is irrelevant since the energy of all threads will be finally aggregated to provide a single figure for the whole application. However, per-task energy can also be monitored individually and periodically during the execution, so that such information can be later used to optimize the energy profile of the application. This is better illustrated in the next subsection through a particular example. The information provided helps understanding the effects in terms of energy of unbalanced thread execution times.

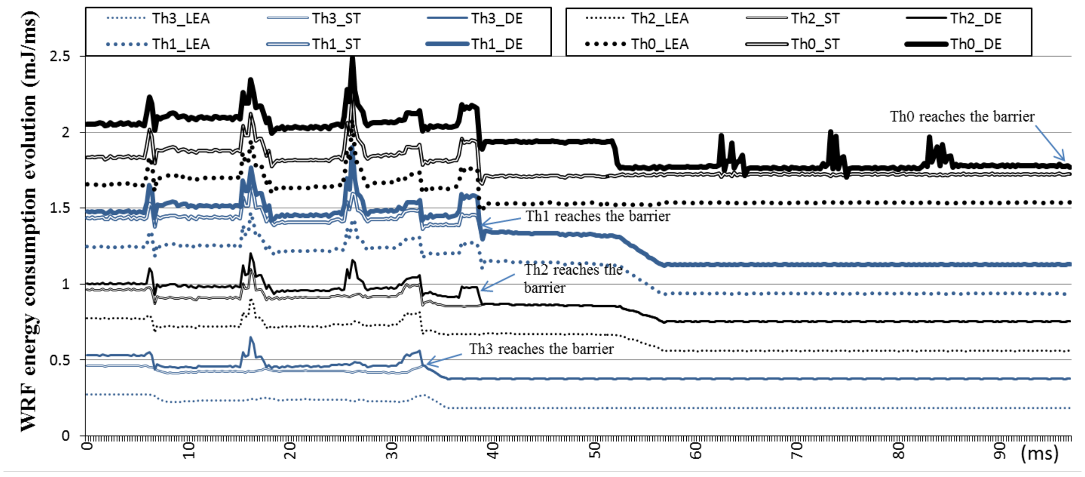


Figure 4.11: Stacked power consumption evolution for *wrf* between two barriers

4.5.2.2 WRF

In this section, we evaluate our energy metering mechanism with real traces from a parallel HPC application running on an actual supercomputer: `wrf`, as introduced in Section 3.4.2

Figure 4.11 shows the evolution of the per-task energy breakdown in the multicore between two barrier communications. Note that energy components are stacked in the plot. At the beginning thread 0 (Th0) consumes more energy than the other threads due to its higher activity (it behaves as an I task). Conversely, Th1, Th2 and Th3 behave as M tasks and therefore, their energy consumption is dominated by static and leakage energy. Eventually, Th3 reaches the barrier and stops consuming dynamic energy. Th3 quickly loses its LLC lines, which decreases its leakage energy. Hence, Th3 energy consumption from this point onwards corresponds to its core static and leakage energy. Behavior for Th1 and Th2 is analogous to that of Th3, but it takes longer for them to lose their LLC cache lines since they reach the barrier before 40ms and lose their LLC cache lines after 50ms. Th0, however, behaves as a I task for 52ms. Then it enters into an M phase, thus decreasing its dynamic energy. At this point Th0 starts increasing its LLC occupancy evicting Th1 and Th2 lines until it occupies the whole LLC after 57ms. This makes leakage energy contribution of Th0 to grow noticeably.

Notice that our energy metering mechanism does not need to be aware of the synchronization among threads of a multithreaded task. For example, if a thread is busy waiting on a lock, even if it is not progressing during that time, the thread is using the processor and it will be metered accordingly. In contrast, if the thread goes to sleep until the lock is released, the core will go to low power mode and less energy will be metered to the thread.

4.5.3 Voltage and Temperature Aware Energy Metering

Voltage and temperature influence energy consumption so they cannot be neglected in general. IBM POWER7 [32, 48] power proxy is already aware of voltage and temperature, which are obtained through sensors. The power proxy scales dynamic, static and leakage energy with constant factors associated to different voltage/temperature combinations. However, such proxy does not discriminate energy in a per-task basis, so it cannot be directly used for PTEM.

Instead, a potential implementation for PTEM could track activities in a per-voltage and per-temperature basis, in such a way that the number of counters required matches the number of combinations of voltage and temperature ranges. For instance, if our chip can operate at 0.8V, 0.9V and 1.0V, and temperature ranges considered are 320K-330K, 330K-340K and 340K-350K degrees, then 9 counters are required for each event to consider all combinations. Owner id tags in caches and occupancy counters will not need to be replicated (note that those arrays are responsible for most of the PTEM overheads).

While voltage and temperature parameters may impact energy consumption of PTEM, their variability is expected to decrease with technology scaling and the increasing number of cores per chip. In particular, smaller geometries suffer from process variations, which limit the minimum voltage that can be used [13]. On the other hand, power efficiency and heat dissipation push for lower operating voltages. Thus, although dynamic voltage scaling techniques may still exist in the future, the range of voltages is expected to decrease, thus leading to fewer voltage levels. Temperature variation may be significant across the chip, but cores will become smaller with technology scaling, thus exhibiting lower in-core temperature variation due to the fact that meaningful temperature gradients occur at a nearly-constant minimum distance [25]. For instance, a difference of 1 degree can only be observed at distances above $0.2mm$ and cores may occupy less than $1mm^2$ in the near future. Similarly, LLC will remain in a narrow range of temperatures due to its relatively low activity. Moreover, maximum allowed temperature decreases due to technology scaling because smaller devices age faster and aging has an exponential dependence on temperature.

4.6 Summary

In this chapter we have addressed the main challenges and opportunities associated with accurate per-task energy metering. As shown in this chapter, existing approaches based on an even distribution of energy across tasks are highly inaccurate. Therefore, we propose (i) a fair reference approach to distribute energy across tasks and (ii) an affordable implementation, PTEM, that tracks task activity and resource utilization at very low cost (below 0.3% energy overhead and 0.85% area overhead).

PTEM is shown to provide highly accurate per-task energy estimates with an average error of 3.1% for SMT multicore configurations and 2.1% for single-threaded multicore ones. We further discuss the required changes, both at hardware and software level, to provide such an accurate, yet implementable, per-task energy metering mechanism. Finally, we show how to use PTEM in the context of parallel applications and a use case where PTEM provides per-task energy measurement in a multicore system showing that the energy consumed by a task when it runs with different co-runners could be huge.

5

Per-task Energy Metering for the DRAM Memory System

5.1 Introduction

The energy demand and cost of computing systems have grown during the last years, and the trend is expected to hold in the coming future [8]. Conversely, computing hardware-related costs (e.g., servers) remain constant or even lower in data centers, desktops and laptops. This leads to scenarios where energy costs are as significant as hardware-related costs. In that respect, despite memory power keeps increasing, reaching 30–50W in high-performance computers [14], there is a lack of understanding on per-task energy consumption in memory. This is aggravated by the fact that memory power profiles across tasks may vary significantly. For instance, a variation of up to 36% in memory power consumption is observed across different SPEC CPU 2000 workloads (from 33.9W to 46.4W) when running four instances of the same benchmark in each workload [14].

As we have seen in Figure 1.2, the average memory power consumption of each benchmark when executing in isolation on the system significantly varies. Different tasks incur different power consumption, with the maximum variation being 57%, between `482.sphinx3` and `462.libquantum` (from 25.7W to 40.4W). Hence, libquantum-like and sphinx3-like workloads executing for the same amount of time would incur significantly different energy consumption. However, to the best of our knowledge, no mechanism has been proposed to measure accurately the memory energy consumed by each task in multicore architectures.

In this chapter we propose, for the first time, an ideal method to fairly distribute the energy consumed in DRAM memories to concurrent running tasks and an efficient implementation of such method. Our approach relies on tracking both the activity incurred by running tasks and the memory bank states they induce. Then energy is attributed fairly to tasks based on their utilization of memory. We show that a low cost and accurate implementation of the ideal model is feasible. Overall, the contributions of this chapter are as follows:

- We propose an ideal per-task energy metering model for DRAM memories, including those based on close-page and open-page policies, as needed for performance/energy optimization, task scheduling and billing in multicore systems. To the best of our knowledge, it is the first reference model against which per-task energy metering mechanisms in DRAM memories can be compared to.

- We devise `DReAM`, an accurate, yet low cost, implementation of the ideal model. `DReAM` requires few counters and registers to be set up in the memory controller to gather the required information. Our results show that such implementation is within a 5% average error with respect to the ideal model.

- We compare `DReAM` with two other energy metering approaches: ES and PTA. In this scenario, PTA is actually a simplified `DReAM` method that further trades accuracy and cost. Our results show that `DReAM` is far more accurate than ES and PTA with negligible hardware overhead.

- We characterize the SPEC CPU 2006 benchmark suite in terms of DRAM energy consumption. Our characterization allows identifying those properties of the applications that impact DRAM energy consumption the most, so that suitable scheduling algorithms can be devised.

In particular, we make the first proposal of (i) an ideal per-task DRAM memory energy metering model and (ii) the hardware support to accurately measure per-task memory energy consumption in multicores with multiple tasks executing concurrently.

The rest of this chapter is organized as follows. Section 5.2 provides background on memory energy consumption and existing approaches for energy metering. Section 5.3 presents our approach to perform ideal per-task memory energy metering. `DReAM`, our efficient hardware implementation of the ideal model, is described in Section 5.4. `DReAM` accuracy is evaluated in Section 5.5. Next, energy consumption of multi-programmed workloads is analyzed in Section 5.6. Finally, Section 5.7 draws the main findings of this chapter.

5.2 DRAM Memory System Fundamentals

5.2.1 DRAM Memory Organization

We focus on DDRx SDRAM as it is one of the most common memory technologies. A DDRx SDRAM memory system is composed by a memory controller and one or more DRAM devices. The memory controller controls the off-chip memory system acting as the interface between the processor and DRAM devices.

A memory *rank* consists of multiple devices (DRAM chips), which in turn consist of multiple banks that can be accessed independently. Each bank comprises rows and columns of DRAM cells (organized in arrays) and a row-buffer to store the most recently accessed rows in the bank. Rows are loaded into the row-buffer using a row activate command (ACT). Such command opens the row, by moving the data from the DRAM cells to the row-buffer sense amplifiers. Once a bank is open, any read/write operation (R/W) can be issued. Finally, a precharge command (PRE) closes the row-buffer, storing the data back into the row. The memory controller can use two different policies to manage the row-buffer: close-page that precharges the rows immediately after every access, and open-page that leaves the rows in the row-buffer open for potential future accesses to the same rows.

Different models can be adopted to access memory. Those models determine which ranks, devices, banks and arrays are accessed on each operation. We adopt the same model as *DRAMSim2*, which in turn models Micron DDR2/3 memories [98]. In this model, all devices in a rank are accessed upon every access. In each device, only one bank is accessed, in which all arrays are accessed. Each array provides the specified row to the sense amplifier on every access, where a number of contiguous columns are accessed over successive cycles to serve an incoming access. In our model, we use a single rank, 8 devices per rank, 8 banks per device and 8 arrays per bank configuration. In one cycle, one bank per device is accessed, thus providing 64 bits in total for the rank. A burst of 8 cycles provides 64 bytes on every access to memory, therefore matching the cache line size for the last level cache (LLC) in the processor.

Under this configuration, all devices are always in the same power state, which is equivalent to consider the power state at rank level. In each device, banks can be in different states. Note, however, that our approach can be easily adapted to other models.

They are not detailed in this thesis due to their similarity for the purpose of PTEM.

5.2.2 DRAM Memory Power Model

Micron has provided a power model based on an abstraction of the internal commands and states of DRAM devices [84]. With the supply voltage known, for a given command or bank state, this model calculates the power consumption through empirical electric current profiles. The electric current profiles are obtained by invoking the DRAM devices non-stop performing a particular command or by staying in one state as long as reliable measurements can be performed. The description and the example values of relevant currents of a particular specification are listed in Table 3.4.

This model provides formulas to calculate the power for each command and each device state. However, as pointed out by Chandrasekar et al. [19], such methodology has several limitations: i) does not consider for DRAM device states transition; ii) uses minimum timing constraints between consecutive commands instead of the actual time interval that the memory controller requires to arbitrate both commands; iii) inflexibility to adapt to other row-buffer management policies.

In our infrastructure, we make use of the power model from Micron and integrate it in a cycle-accurate memory system simulator, DRAMsim2, as introduced in Section 3.2. By doing so, we can easily overcome the limitations of the Micron model by refining the power profiles to the energy per cycle and per command. For example, the power dissipated in a particular DRAM device state can be expressed as the power consumption in one memory cycle. In this way, since DRAMsim2 emulates the bank-level operations in every cycle, we can map the energy to the state transitions accurately. Analogously, since DRAMsim2 manages the timing constraints, the command power can be converted to count-based energy consumption with timing parameters. This is analogous to the methodology used by Deng et al. [22], where the same data from Micron is used as input.

Next, we show the details on the main components of this power/energy model. DRAM devices can be in three different states: Power Down (P), Standby (S), and Active (A). In each state, the power dissipation in one cycle is P_P, P_S and P_A, respectively. P state is the one with the lowest power dissipation.

$$P_P = IDD_{2P} \times V_{DD} \times t_{ck} \tag{5.1}$$
$$P_S = IDD_{2N} \times V_{DD} \times t_{ck} \tag{5.2}$$
$$P_A = IDD_{3N} \times V_{DD} \times t_{ck} \tag{5.3}$$

where t_{ck} stands for the cycle time corresponding to the operating clocking frequency of the memory system, and V_{DD} stands for the supply voltage.

In the case of memory commands, since their electric current profiles are monitored when they execute with minimum timing constraints in DRAM devices, we normalize them with the same timing [84]. We calculate the energy consumed by the ACT and PRE commands as follows:

$$\begin{aligned} E_{ACT} &= (IDD_0 - \tfrac{IDD_{3N} \times t_{RAS} + IDD_{2N} \times (t_{RC} - t_{RAS})}{t_{RC}}) \times V_{DD} \times \\ &\qquad \tfrac{t_{RAS}}{t_{RC}} \times \tfrac{t_{RC}}{t_{RRD}} \times t_{RAS} \times t_{ck} \end{aligned} \tag{5.4}$$

$$\begin{aligned} E_{PRE} &= (IDD_0 - \tfrac{IDD_{3N} \times t_{RAS} + IDD_{2N} \times (t_{RC} - t_{RAS})}{t_{RC}}) \times V_{DD} \times \\ &\qquad \tfrac{t_{RC} - t_{RAS}}{t_{RC}} \times \tfrac{t_{RC}}{t_{RRD}} \times (t_{RC} - t_{RAS}) \times t_{ck} \end{aligned} \tag{5.5}$$

Since the set of ACT and PRE commands are recursively operated, the measured current IDD_0 includes the current that is incurred by the A and S states. The remaining current is split evenly among ACT and PRE commands, since the activities incurred by these commands are comparable. Note that the minimum timing constraint between two ACT commands during the current measurement is actually t_{RRD}, which is scaled down with t_{RC}. This is similar to the approach introduced by Chandrasekar et al. [19]. Separating the energy of the ACT and PRE commands pair is necessary, for example, under open-page policy. In close-page policy, they can be combined not to loss integrity.

In the case of READ and WRITE commands, consumed energy is computed as follows:

$$E_{READ} = (IDD_{4R} - IDD_{3N}) \times V_{DD} \times \tfrac{BL}{2} \times t_{ck} \tag{5.6}$$

$$E_{WRITE} = (IDD_{4W} - IDD_{3N}) \times V_{DD} \times \tfrac{BL}{2} \times t_{ck} \tag{5.7}$$

where BL stands for the burst length of the data being transferred on the bus. In DDR memories, this value is shortened to $\frac{BL}{2}$.

Unlike SRAM memory cells, DRAM cells are unable to retain contents indefinitely. Instead, DRAM cells discharge over time and eventually, they lose their contents. Therefore, they must be read and written back at a given minimum frequency to keep their contents. Although this has some implications in energy consumption (to read/write memory contents) and bandwidth (refresh operations may delay program's accesses), DRAM cells are smaller and less power-hungry than SRAM ones, so they are used to implement main memory. Thus, all memory contents need to be refreshed periodically. A refresh command is normally accompanied with several PRE commands, but the PRE command consumed energy can be computed with Equation 5.4. Then, the refresh energy is calculated as:

$$E_{REF} = (IDD_5 - IDD_{3N}) \times V_{DD} \times \frac{t_{RFC}}{t_{REFI}} \times t_{ck} \quad (5.8)$$

Then, we break DRAM memory energy consumption down into three components: active, refresh and background energy.

- Active energy corresponds to the energy spent to perform those *useful* activities, such as READ/WRITE, their related ACT/PRE commands and the termination energy due to terminating signals of other ranks on the same channel. The definition of it is aligned with Section 2.1
- Background energy includes the maintenance and leakage energy. Maintenance energy corresponds to the energy consumed due to *useless* activities not triggered by the program(s) being run. For instance, DRAM memory may stay in a higher energy consumption state during idle cycles so that it can quickly react and serve a new access. Alternatively, it may remain in a much lower power mode with lower maintenance power dissipation, but it may take longer to serve a new access due to the time required to transition to an active mode. Leakage energy corresponds to the energy wasted due to imperfections of the technology used to implement the circuit. Note that if circuits are implemented with *perfect* technology, no leakage power would be dissipated. This energy is referred to as maintenance or leakage energy indistinctly in other works [115]. For the sake of clarity, we make use of the term background energy to refer to all consumed energy except active and refresh energy.
- Refresh energy corresponds to the energy consumed to refresh periodically all memory contents. Although refresh energy somewhat belongs to maintenance energy, its incurred power is consistent overtime unlike the fluctuating maintenance power. We separate it out to better distribute the energy consumption to tasks.

5.3 Idealized PTEM for DRAM Memory

In this section we present an idealized model for per-task DRAM energy metering without considering hardware cost. The result of this model is later used as the reference for `DReAM` model to meter per-task energy with a low-cost implementation. We assume a multicore architecture where an on-chip memory controller serves as the bridge to the off-chip memory. Next we describe the memory model considered in this work, how energy is

Table 5.1: Memory commands, timing, device states and background energy breakdown for a read operation in close-page mode.

Command	T_0	–	ACT		READ	PRE	–
	T_1	–					
Timing	T_0	–	t_{XP}	t_{RCD}	t_{RTP}	t_{RP}	–
State	$Bank_0$	PD	S	A		S	PD
	$Bank_1$			S			
	$Bank_2$						
	$Bank_3$						
Power	Rank	E_{PD}	E_S	E_A		E_S	E_{PD}
	T_0	$\frac{E_{PD}}{2}$	$E_S - \frac{E_{PD}}{2}$	$E_A - \frac{E_{PD}}{2}$		$E_S - \frac{E_{PD}}{2}$	$\frac{E_{PD}}{2}$
	T_1	$\frac{E_{PD}}{2}$					

consumed in the different memory blocks, and our models to split energy among different tasks.

5.3.1 A Case of Energy Consumption in DRAM Memory

In this section, we introduce how the energy is consumed by internal activities in DRAM devices in the scenario when one memory read request is served in memory.

Table 5.1 shows the effect on memory of a read command. We observe that the device is in *PD* state when the memory controller is not processing any request. Note that in our configuration all devices in the rank are in the same state and therefore, rank and device states match. When the memory controller receives a memory access request from task 0 (T_0), it sends a clock enable (*CKE*) signal to transition the rank from *PD* to *S* state. The device stays in S state as long as all banks are powered up and idle. This includes the time the device is waiting for the memory controller to send those commands corresponding to the requests in the memory controller's queues. During the *S* state, background power is higher than in *PD* state ($P_S > P_{PD}$). *S* state lasts t_{XP}, as depicted in Table 5.1. Eventually, some banks are activated so that the device as well as some banks transition to *A* state. Note that in this model, when the ACT command is issued the device (and so the rank) switches to *S* state, and whenever the corresponding bank has been activated, the device switches to *A* state. The device and the accessed banks (Bank$_0$ in the example) are in *A* state during part of the activation period (t_{RCD}) and while the read/write command is served (t_{RTP} in the example for a read command). Note that there is another timing constraint: each bank can only be precharged after t_{RAS}. Therefore, in the case when $t_{RAS} > (t_{RCD} + t_{RTP})$, the bank stays in *A* state at least for t_{RAS} after being activated. While in *A* state, the device incurs the highest power dissipation,

P_A, with $P_A > P_S$. Once the only command being processed is the *PRE* command, the device and accessed banks transition to S state. When no command is executed and no memory access request exists in the memory controller buffer for a certain time interval, the memory controller returns the device to *PD* state.

Most modern memory controllers implement open-page and/or close-page policies. They differ on how the data array row-buffer is managed (for how long the row-buffer keeps open). Next, we present how per-task energy is metered under both policies.

5.3.2 Per-Task Energy Metering for Close-Page

Our idealized model relies on the fact that background energy consumption of a device depends solely on its current state, which can be induced by different, concurrent accesses. Therefore, our model attributes background energy to each task based on the state it imposes on memory. Memory occupancy is discarded as input for the model since background energy does not depend on it. As reported in [22] background energy accounts for over 50% of the memory energy consumed by programs. Memory occupancy is discarded as input for the model since background energy does not depend on it. Thus, distributing background energy according to resource utilization is crucial to meter per-task memory energy accurately.

1. During *PD*, only background power is consumed. Such energy cannot be attributed to any task since no task has any memory activity during *PD*. Hence, we divide background power evenly across all tasks running in the system.

2. Whenever a DRAM device switches from *PD* to S state, the extra background power incurred due to S state, i.e. $P_S - P_{PD}$ is distributed uniformly across all tasks with inflight commands that force the DRAM devices to stay in S state.

3. When a DRAM device is in A state (active), the extra power incurred (i.e. $P_A - P_S$) is distributed evenly across all tasks enforcing A state.

For instance, Table 5.1 shows the case where one task, T_0, issues a *read* command (first row) while another task, T_1, issues no command. Next, let us assume that those are the only tasks using the memory system. During the whole period, T_1 is responsible only for half of the P_{PD} power (last row), while T_0 is responsible for half of the P_{PD} and all P_S and P_A extra power (penultimate row).

When multiple commands are processed in parallel, we follow the same principle of attributing power to those tasks that impose the memory chip to be on a given state. In

Table 5.2: Memory commands, timing, device states and background energy breakdown for several operations in close-page mode.

Comm.	T_0	–	ACT		READ		PRE		–	
	T_1	–		ACT		READ		PRE		–
Timing	T_0	–	t_{XP}	t_{RCD}	t_{RTP}		t_{RP}		–	
	T_1	–		t_{RRD}	t_{RCD}	t_{RTP}		t_{RP}		$\dots$
State	$Bank_0$	PD	S	A			S	S		PD
	$Bank_1$			S	A					
	$Bank_2$				S					
	$Bank_3$									
Power	Rank	E_{PD}	E_S	E_A				E_S		E_{PD}
	T_0	$\frac{E_{PD}}{2}$	$E_S - \frac{E_{PD}}{2}$	$E_A - \frac{E_S}{2}$	$\frac{E_A}{2}$		$\frac{E_S}{2}$		$\frac{E_{PD}}{2}$	
	T_1	$\frac{E_{PD}}{2}$		$\frac{E_S}{2}$	$\frac{E_A}{2}$		$E_A - \frac{E_S}{2}$	$\frac{E_S}{2}$	$E_S - \frac{E_{PD}}{2}$	$\frac{E_{PD}}{2}$

the example in Table 5.2, we show a particular case where both T_0 and T_1 issue commands in parallel. First, the device is in *PD* state. Eventually, T_0 makes the device transition to *S*, so T_0 is responsible for the extra background power. Then, devices transition to A state and T_1 starts its activate command. Both tasks are equally responsible for P_{PD} and P_S power, but only T_0 is responsible for P_A power. Later, T_1 also enforces memory to be in A state so that the total power must be uniformly distributed across both tasks. Finally, as commands finish, tasks T_0 and T_1 stop enforcing high-power states and power dissipation is attributed only to those tasks imposing each particular state.

Regarding the refresh operations, according to the JEDEC standard of DDR2/3 SDRAM memory [52], it is required to issue eight refresh commands during a given time window. Thus, the memory controller has some flexibility to schedule those refresh commands, so that the execution of commands from tasks can avoid interruption. The refresh energy is guaranteed to be constant in the memory system during a given period of time, regardless of the activities of running tasks. Given that refresh commands occur in all banks simultaneously, they cannot happen in parallel with any other command. Thus, both dynamic and background energy incurred during refresh is accounted as refresh energy. Although refresh energy is not triggered by the execution of tasks, it is consumed as long as the system is powered up. Thus, tasks running in the system are assumed to be responsible for the system being up, and so refresh energy is evenly split across those tasks.

5.3.3 Per-Task Energy Metering for Open-Page

As opposed to the close-page policy, in open-page, ACT/PRE commands may not be needed by a memory access, since banks remain open after being accessed. However,

Table 5.3: Memory commands, timing, power states and background energy breakdown when hit in row buffer in open-page mode.

Comm.	T_0	–	ACT		READ	–	–	
	T_1	–				–	READ	–
Timing	T_0	–	t_{XP}	t_{RCD}	t_{RTP}	–		
	T_1	–					t_{RTP}	–
State	$Bank_0$	PD	S	A				
	$Bank_1$		S					
	$Bank_2$							
	$Bank_3$							
Power	Rank	P_{PD}	P_S	P_A				
	T_0	$\frac{P_{PD}}{2}$	$P_S - \frac{P_{PD}}{2}$	$P_A - \frac{P_{PD}}{2}$			$\frac{P_{PD}}{2}$	
	T_1	$\frac{P_{PD}}{2}$					$P_A - \frac{P_{PD}}{2}$	

energy consumed by open banks is still attributed to those tasks that opened the banks. Regarding background energy, the same principle as for close-page is followed: attributing the energy to tasks based on the state they impose to memory.

As in the close-page policy, devices are powered up and activated (A state) to execute commands. However, once the corresponding read/write operation finishes, those devices remain open in A state. This is illustrated in the example in Table 5.3 that reflects the case of a row-buffer hit. The task that opened the bank (T_0 in the example) is responsible for the extra background energy of the activated devices (after the first t_{RTP}). Eventually, another access to the open banks can occur. If this is the case, no precharge command is needed. Since T_1 read access is a row-buffer hit, it can directly read data from the row buffer. Consequently, T_1 becomes responsible for the extra background energy, while T_0 is only responsible for half of the PD energy.

Analogously, the same principle also applies when multiple accesses are interleaved, as shown in Table 5.4. In this particular case, T_0 has already opened one bank ($Bank_0$), which imposes the A state to the rank and the corresponding bank. Eventually, T_1 accesses the same rows which incurs a row-buffer hit. During this process, the extra background energy attribution switches like in the previous example. Then, after T_1 finishes its operation, T_0 accesses the same rows which incurs another row-buffer hit. Thus, the attribution of extra background energy switches back to T_0 again. Whenever the page is closed, T_0 is also responsible for the precharging dynamic energy, which should have been attributed to T_1 if T_0 had not accessed the open bank. The main reason why we distribute the extra background energy this way is that, when the bank is firstly opened, it is impossible

Table 5.4: Memory commands, timing, power states and background energy breakdown when multiple interleaved accesses from two tasks accessing the same bank in open-page mode.

Comm.	T_0	READ	–			READ	–	PRE
	T_1	–		READ	–			
Timing	T_0	t_{RTP}	–			t_{RTP}	–	t_{RP}
	T_1	–		t_{RTP}	–			
State	$Bank_0$	A						S
	$Bank_1$	S						
	$Bank_2$	S						
	$Bank_3$	S						
Power	Rank	P_A						P_S
	T_0	$P_A - \frac{P_{PD}}{2}$		$\frac{P_{PD}}{2}$		$P_A - \frac{P_{PD}}{2}$		$P_S - \frac{P_{PD}}{2}$
	T_1	$\frac{P_{PD}}{2}$		$P_A - \frac{P_{PD}}{2}$		$\frac{P_{PD}}{2}$		

to predict its future accesses, thus the activation energy is attributed to the first user. Similarly, the precharging energy is attributed to the last user, who triggered the PRE command. Regarding background energy, we also assume that the last task imposing a particular device state accounts for the extra energy. Although our choice is, to some extent, arbitrary, we regard it as fair.

In summary, activate and read/write dynamic energy is attributed to the task performing the access, whereas precharge energy is attributed to the last task accessing such row. Note that on a refresh command all banks need to be closed, and so precharge energy for open pages is attributed to the last task accessing each of them. Other than that, energy distribution is analogous for close-page and open-page policies.

5.3.4 Ideally Formalized Per-Task Energy Metering in Memory

We generalize the memory energy consumed by each task as follows.

1) The background (*bg*) energy attributed to a task can be generalized as follows for both open- and close-page policies:

$$
\begin{aligned}
E_{bg}^{mem}(T_i) \quad &= E_{PD} \times ExecTime(T_i)/N_T \\
&+ \sum_{j=0}^{ExecTime(T_i)} \left((E_S - E_{PD}) \times \frac{\delta_{i,j}^S}{N_{S,j}^T} \right) \\
&+ \sum_{j=0}^{ExecTime(T_i)} \left((E_A - E_S) \times \frac{\delta_{i,j}^A}{N_{A,j}^T} \right)
\end{aligned}
\tag{5.9}
$$

In the first addend each running task is metered an even part of E_{PD}, where $ExecTime(T_i)$ stands for the execution time of task i in cycles and N_T for the number of tasks running in the processor – not necessarily the maximum number of tasks allowed in the processor. The second and third addends meter $E_S - E_{PD}$ and $E_A - E_S$ for tasks enforcing those states. $N^T_{S,j}$ and $N^T_{A,j}$ correspond to the number of tasks imposing S and A states respectively in cycle j; and $\delta^S_{i,j}$ and $\delta^A_{i,j}$ indicate if the task i makes memory be in S and A state respectively, in cycle j. In other words, $\delta^A_{i,j}$ is 1 if task i is executing a *read*, *write* or *activate* (last t_{RCD} cycles) command in cycle j, and 0 otherwise; and $\delta^S_{i,j}$ is 1 if task i is executing a *precharge* or activate (first t_{XP} cycles) command or if it has pending commands in the memory controller while all banks are idle in cycle j, and 0 otherwise. Note that, as stated before, memory occupancy is not considered for metering energy to tasks since the memory regions not used by the task under consideration cannot be turned off when idle. Hence, background energy remains the same regardless of the memory space used.

2) Active energy for a task depends on the number of commands it performs, as shown in the following equation:

$$\begin{aligned} E^{mem}_{dyn}(T_i) = E^{mem}_{read} \times N_{RD}(T_i) + E^{mem}_{write} \times N_{WR}(T_i) \\ + E^{mem}_{ACT} \times N_{ACT}(T_i) + E^{mem}_{PRE} \times N_{PRE}(T_i) \end{aligned} \tag{5.10}$$

where E^{mem}_{read}, E^{mem}_{write}, E^{mem}_{ACT} and E^{mem}_{PRE} stand for the energy of each command, and $N_{RD}(T_i)$, $N_{WR}(T_i)$, $N_{ACT}(T_i)$ and $N_{PRE}(T_i)$ stand for the number of memory internal commands executed by task i.

3) *Refresh* operations may have some side effects such as delaying some commands issued by running tasks. However, this fact does not alter the energy model. Also, refresh commands consume some energy to access the corresponding rows. Since refresh operations are distributed evenly over time at a fixed rate and they are not originated by any particular task, their energy is split evenly across all running tasks. Thus, refresh energy per task is as follows:

$$E^{mem}_{refr}(T_i) = E^{mem}_{refr} \times N_{Ref} \times ExecTime(T_i)/N_T \tag{5.11}$$

E^{mem}_{refr} corresponds to the active energy of a refresh command. N_{Ref} corresponds to the average number of refresh operations performed per cycle.

5.4 DReAM, an Implementable Approach

Implementing the exact computation of the *idealized* energy model is expensive — if at all feasible — due to the large number of events to be tracked, the frequency at which they must be tracked, and the lack of information that the processor has about the memory state. On the other end, metering memory energy evenly among running tasks or proportionally to the number of accesses that they perform requires minor changes to current architectures. However, these approaches exhibit low estimation accuracy as shown later in Section 5.5.2. Therefore, we propose `DReAM`, our per-task energy metering approach that trades off energy metering accuracy and implementation complexity.

In `DReAM` memory model, active and refresh energy can be easily tracked as in the idealized model. This requires the memory vendor to provide the active energy per access type, namely E^{mem}_{read}, E^{mem}_{write}, E^{mem}_{ACT} and E^{mem}_{PRE} for tracking active energy and E^{mem}_{refr} for tracking refresh energy, as well as the average number of refresh operations per cycle (N_{Ref}). These parameters are already provided by chip vendors like Micron for DDR2/3 memories [84], so our model imposes no change to current DDR2/3 memories. In the memory controller, we only require per-task activity counters, namely $N_{RD}(T_i)$, $N_{WR}(T_i)$, $N_{ACT}(T_i)$ and $N_{PRE}(T_i)$. Total background energy, $E^{mem}_{bg,total}$ can be obtained by metering memory energy consumption [43] and subtracting active and refresh energy. The PD background energy is constant and hence easy to track. Meanwhile, the remaining background energy, E^{mem}_{rem}, is due to active and standby periods (i.e. $E^{mem}_{bg,total} = E^{mem}_{PD} + E^{mem}_{rem}$).

Our model distributes E^{mem}_{PD} uniformly across all tasks, while E^{mem}_{rem} is distributed based on access frequencies per task. To that end, we divide the execution into intervals of *IntMem* processor cycles and track the number of memory accesses sent to the memory controller (in a per-task basis) in the current interval. Thus, background energy is obtained as follows:

$$E^{mem}_{bg,\,total}(T_i) = \frac{P^{mem}_{PD} \times ExecTime(T_i)}{N_T} + \sum_{j=0}^{\frac{ExecTime(T_i)}{IntMem}} \frac{N^{T_i}_{acc,j}}{N^{TOTacc}_{j}} \times E^{mem}_{rem,j} \tag{5.12}$$

where P^{mem}_{PD} is the *PD* background energy, $N^{T_i}_{acc,j}$ tracks the number of memory accesses of task i during interval j, and N^{TOTacc}_{j} tracks the total number of memory accesses in interval j. $E^{mem}_{rem,j}$ is the non-power-down background energy in interval j, obtained by subtracting all other sources of energy consumption from the total energy measured in the interval. Sensitivity to the sampling interval ($IntMem$) is studied in the evaluation section.

Table 5.5: DReAM hardware requirements.

Block	Memory Vendor	Extra Logic
Memory	E^{mem}_{read}, E^{mem}_{write}, E^{mem}_{ACT}, E^{mem}_{PRE}, E^{mem}_{PD}, E^{mem}_{refr}, N_{Ref}	N_{RD}, N_{WR}, N_{ACT}, N_{PRE}, $N_{RD}(T_i)$, $N_{WR}(T_i)$, $N_{ACT}(T_i)$, $N_{PRE}(T_i)$, $IntMem$ cycle counter

Putting it All Together

DReAM requires little hardware overhead. DReAM mostly requires setting up some counters similar to the PMCs currently available in most high-performance processors. DReAM support does not interfere the execution of programs since it is not in any critical path. Table 5.5 summarizes those parameters required from the memory vendor and the extra logic (i.e. counters) that must be set up. Counters with the "(T_i)" suffix must be replicated for each task. Thus, how many of them are needed is dictated by the number of tasks that may run simultaneously in the chip.

Regarding the interface with the software, the OS is responsible for keeping track of the energy consumed by every task running in the system. DReAM exports a special register, called Memory Energy Metering Register (MEMR), that acts as the interface between DReAM and the OS. The OS can access that register to collect the energy estimates made by DReAM. This typically will happen when a context switch takes place. At that moment, the OS reads the MEMR using the hardware-thread index (or CPU index) for the task that is being scheduled out (T_{out}). Then, the OS aggregates the energy consumption value read in the *task struct* for T_{out}. Right after the new task (T_{in}) is scheduled in, the memory state may remain at a particular state due to an access triggered by the task that has been scheduled out. Although, DReAM attributes background energy consumption to T_{in}, this occurs during few cycles (in the order of tens or hundreds of cycles). Under a processor frequency of 2GHz, 500 cycles are equivalent to $0.25\mu s$, while context switches occur at much higher granularity, every 10-100ms.

As in chapter 4, the time the OS spends working on behalf of a given task is attributed to the calling task. The remaining energy consumed by the OS can be evenly attributed to all running tasks. In any case, DReAM provides the hardware support needed to attribute OS energy to tasks as required.

5.5 Evaluation

5.5.1 Experimental Setup

The main experimental setup is introduced in Section 3.4. In particular in this chapter, we consider three CMP processor configurations with 1, 4 and 16 single-threaded cores. The LLC is partitioned with 256KB 16-way per core. Therefore, the LLC size is 256KB, 1MB, and 4MB for 1, 4, and 16 cores, respectively. These configurations have been chosen to discount the effect of on-chip inter-task interferences due to shared resources (e.g., shared LLC cache), thus allowing to consider memory effects only.

For the DRAM memory we model an 8GB memory as it is large enough to support the workloads used in this work. DRAM memory is single-rank with 8 devices per rank, 8 banks per device and 8 arrays per bank. We evaluate close-page and open-page DRAM memory row-buffer management policies, but differences were negligible: Since many current DRAM memories have a low-power mode, the open banks under open-page policy quickly transition to power down state when there is no incoming request. In this case, open-page policy performs similarly to close-page in most of the cases. Thus, we only report results for one of the policies: close-page.

Average power consumption for the 8GB setup is 5.4W, 8.6W and 18.8W for 1-thread, 4-thread and 16-thread workloads respectively. For a setup of 64GB (results not shown in this work) power increases by a 2x-3x factor (e.g., 14.7W for 1-thread workloads). Note that this is around half the power consumption reported in Section 5.1, which is consistent since our setup is less aggressive than that of the particular server used in the real experiment. In particular, we assume a processor operating at 2GHz and DRAM operating at 1GHz, whereas the CPU of the server used operates at 3.2GHz and its memory at 1.6GHz. Nevertheless, our proposal is orthogonal to those parameters.

The benchmarks and workload generation strategy are introduced in Section 3.4.3. To measure accuracy in the energy estimations, we make use of the metrics in Equations 3.1 and 3.2.

5.5.2 `DReAM` Energy Estimation

In this section we show the accuracy of `DReAM` with respect to the ideal model presented in Section 5.3. We also include the ES model that uniformly splits energy across all running tasks regardless of their activity and memory behavior, together with a simple PTA model that splits energy across tasks proportionally to their memory accesses.

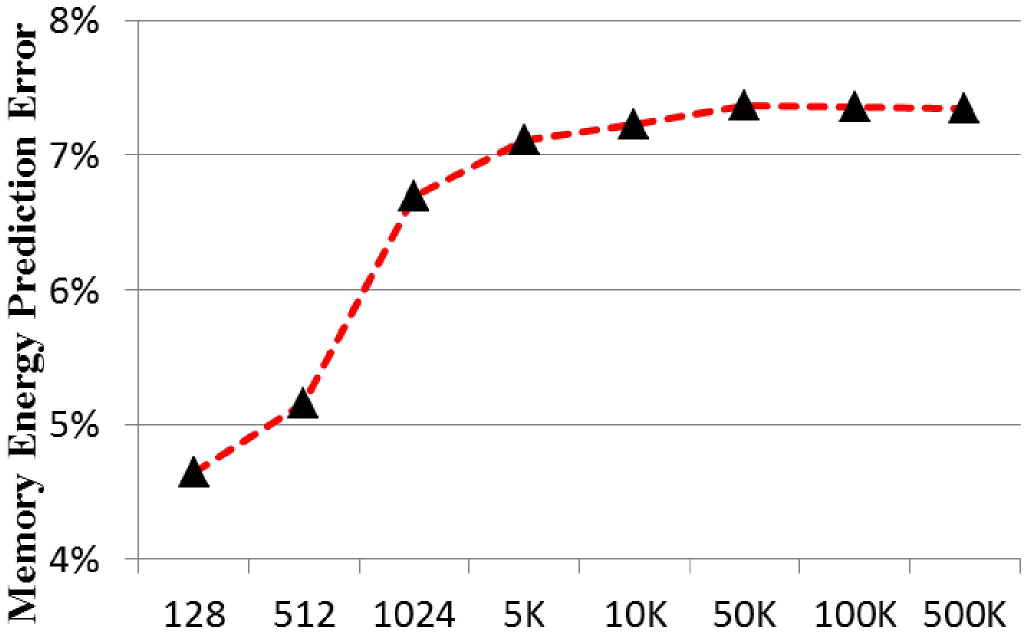


Figure 5.1: Per-task DRAM memory energy prediction of a 4-core workload *soplex+sjeng+gcc+namd* with different sampling intervals.

5.5.2.1 `DReAM` Sampling Interval (IntMem)

The memory energy consumption prediction of `DReAM` varies with different sample period (interval) lengths. When choosing the interval length, we seek for a reasonable tradeoff between accuracy and hardware cost. Figure 5.1 shows the average *WldPredError* for each task in a particular workload. This workload belongs to group *X* and runs in a 4-core configuration. We explore sampling periods from 128 to 500K processor cycles. Trends for most workloads are similar, so we have used this particular one to illustrate the sensitivity of `DReAM` to the particular sampling period.

As expected, higher sampling frequency increases accuracy. However, discrepancy between short and long sampling periods is not huge (from 4.6% to 7.4% average *WldPredError*). Some meaningful average *WldPredError* increase is observed when moving from a 512-cycles sampling interval to a 1024-cycles interval. Further increasing the interval size until reaching half million cycles has little impact on accuracy since deviation from the ideal model quickly flattens[1]. Thus, we have chosen two different interval sizes with different accuracy/cost tradeoff: 512 and 500K cycles sampling intervals.

[1]Longer sampling period is also applicable, however, `DReAM` aims to provide the estimation in a finer granularity than the operating system quantum to be of more flexible use.

5.5.3 `DReAM` Energy Consumption Prediction

Next we evaluate the off estimation for 4-core and 16-core processor setups with respect to the ideal model. Note that the ideal model is the only reference model as no existing hardware provides accurate per-task DRAM energy metering.

Figure 5.2 shows the result for the 24 workloads (8 of each type) for the 4-core setup. We observe that, in general, the ES model is highly inaccurate averaging over 45% prediction error across all workloads. Prediction is more accurate for L and H workloads than for X ones. This is expected since benchmarks in L and H workloads are more homogeneous, so their individual power consumption is also more homogeneous than in X workloads. In some particular workloads, the prediction error is even below 10%. Nevertheless, ES model prediction error is very high in general, ranging from 30% to 85% for most workloads. For X workloads, the prediction error is always above 58%. PTA model improves the estimation accuracy, with an average prediction error around 23%. PTA accuracy is high for H workloads (the errors are all under 10%) since the large number of accesses of H benchmarks makes energy more proportional to the number of accesses (dynamic energy becomes dominant). However, benchmarks in L group infrequently access memory, so their memory energy is mainly background energy, which PTA fails to predict accurately. This fact is particularly noticeable for workload $w4$ where, although all tasks have few memory accesses and so, their energy is dominated by PD background energy, the fact that one task has a number of access relatively much higher than the others makes it account for most of the energy, thus producing very high error prediction. Conversely, in this workload the ES model is far more accurate than PTA since energy is quite homogeneous across tasks in the workload. Our `DReAM` model improves prediction accuracy significantly over both ES and PTA. When the sample period granularity is 512 cycles, the prediction error is always below 10%, and 3.9% on average. If the sampling period increases to 500K cycles, the prediction error may reach 14.0% at most for one particular workload, and 6.1% on average. As shown, `DReAM` successfully predicts the energy consumed by each task consistently across workloads. In particular, this holds (i) when PTA works well and ES not (e.g., workload $w12$), (ii) when ES works well and PTA not (e.g., workload $w4$), and (iii) when both PTA and ES work badly (e.g., workload $w5$).

Figure 5.3 shows results for the 16-core setup. First, we observe that ES and `DReAM` accuracy remains similar to that of the 4-core setup. In contrast, PTA accuracy slightly improves.

The average prediction error across all workloads for the ES model rises to 53%. The increase is particularly noticeable for L workloads. Since total energy for L workloads

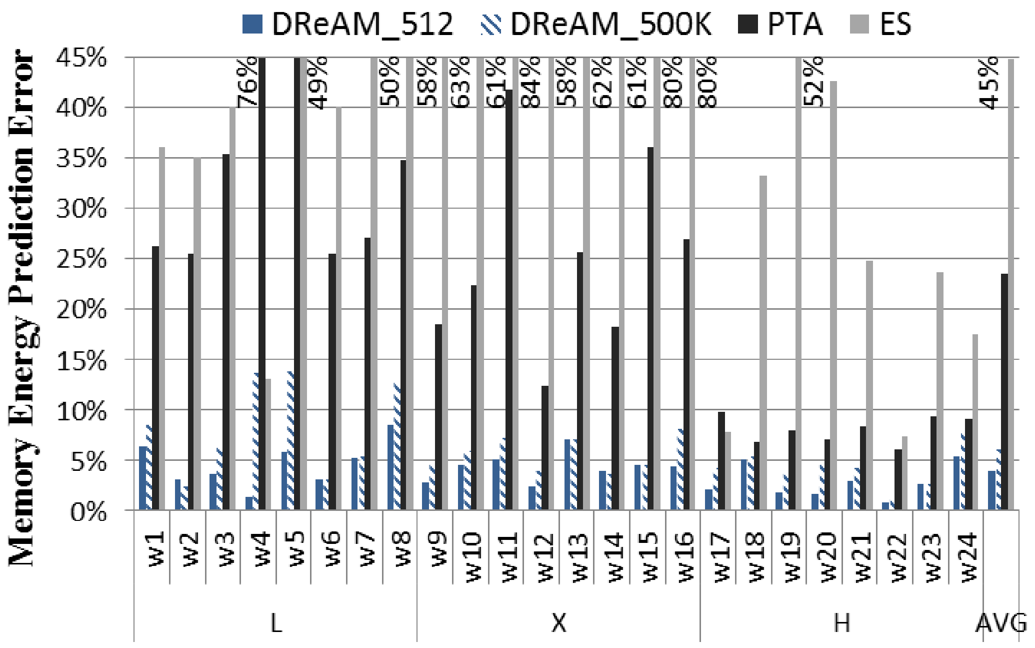


Figure 5.2: Per-task DRAM energy prediction error for 4-core workloads.

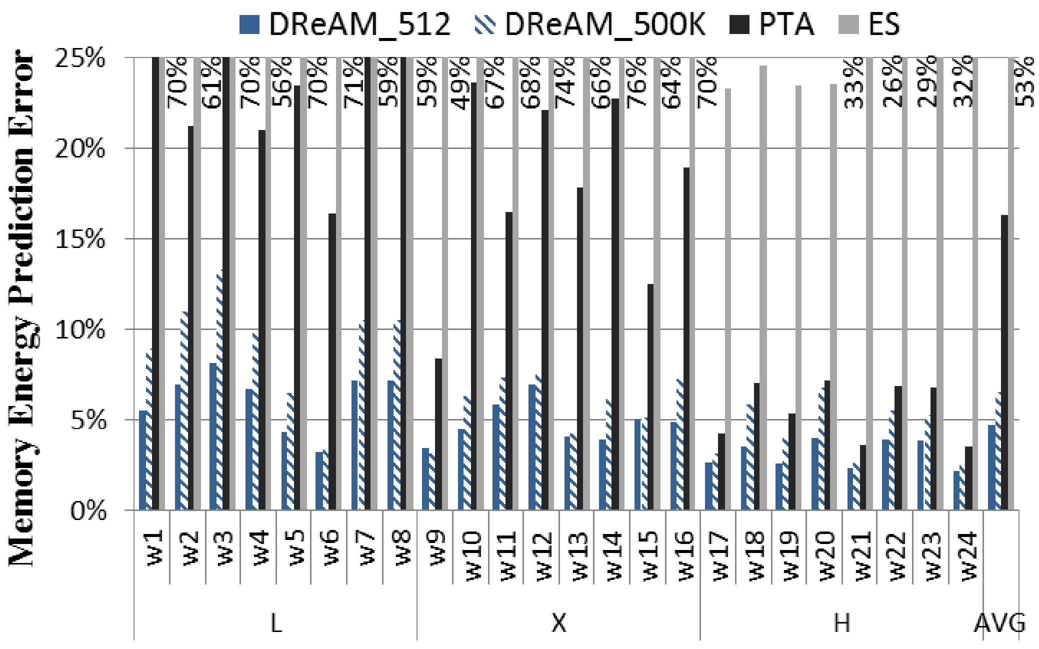


Figure 5.3: Per-task DRAM energy prediction error for 16-core workloads.

is relatively low, low deviations (in absolute numbers) become high in relative numbers. A similar effect occurs for `DReAM`, thus making L workloads to exhibit the lowest prediction accuracy, followed by X workloads, where half of the benchmarks are L benchmarks.

Conversely, *H* workloads consume higher energy and relative deviations become less significant for all models. Trends for PTA are similar to those for the 4-core setup, thus exhibiting higher accuracy for *H* workloads, although accuracy for the 16-core setup is higher. This is due to the fact that, with 4 cores, a large deviation for one benchmark has significant impact in average results, but such average impact becomes lower across 16 tasks. However, maximum error for individual benchmarks in each workload still remains high.

Nevertheless, PTA has an average prediction error above 10%, and around 23% for a particular workload. Opposedly, `DReAM` error is below 5% on average (512-cycles interval) and always below 8% across all workloads. Note that the gap between 512 and 500K cycles sampling intervals for `DReAM` is still around 2%, as in the 4-core case. Our results prove that `DReAM` is far more accurate than ES and PTA models across all workload types, and average prediction error remains nearly the same for 4 and 16 cores, thus proving that `DReAM` scales well.

In conclusion, `DReAM` model greatly improves per-task DRAM energy estimation over ES and PTA at low cost.

5.5.4 `DReAM` Area and Energy Overhead

`DReAM` requires some hardware support in the form of counters to track memory activity. Those counters are in the memory controller, which in general is on-chip, so the DRAM devices remain unchanged.

As shown in Table 5.5, `DReAM` needs few counters (5 shared counters and 4 extra counters per thread). 32-bit counters suffice to track the corresponding events. Further, few of those counters are accessed on a memory access and at the end of a sampling interval. Although computing the energy consumed by each thread in a particular interval involves few arithmetic operations, low-area and low-power arithmetic units (e.g., iterative multipliers [101] and dividers [57] operating at low frequency) can be set up for that purpose. We have considered the energy consumption for two different sampling intervals: 512 and 500K cycles. Area and power/energy overheads have been estimated with power models analogous to those of Wattch [16] built on top of CACTI 6.5 simulation tool [87]. CACTI is a flexible tool modeling delay, energy (active and leakage) and area of cache memories and SRAM-based arrays.

Results for 4-core and 16-core configurations show that the total energy and area overhead for `DReAM` is largely below 0.1% of the entire chip. If we compare `DReAM` energy overhead with DRAM energy consumption, it is also largely below 0.1% of total DRAM

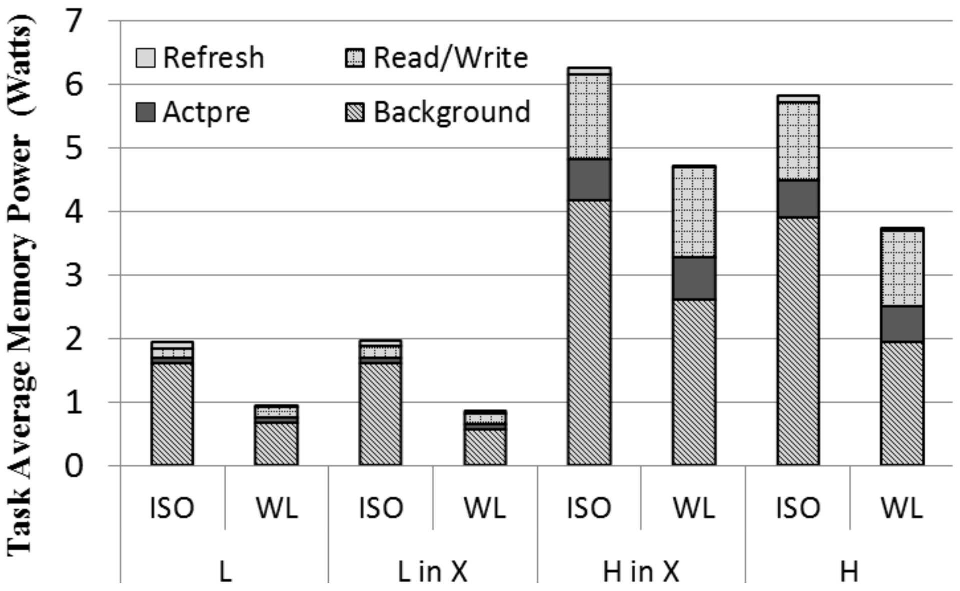


Figure 5.4: All workloads power consumption comparison in a 4-core setup.

energy consumption. Furthermore, relative overheads do not change noticeably if the core count is increased, which proves that `DReAM` scales well. Energy overheads for 512 cycles sampling intervals are higher than for 500K intervals, but still under 0.1% for the whole chip. Due to its higher accuracy and still low overheads, the sampling interval considered for the characterization presented in the next section is 512 cycles.

5.6 Case Study

In this section, we analyze how programs with different memory access profiles interact in terms of memory power consumption. For that purpose, we use `DReAM`, our proposed method for accurate per-task memory energy metering.

5.6.1 Workload as a Whole

We first analyze the different workloads with attentions on the power consumption of the different benchmark types rather than individual benchmarks.

Figure 5.4 shows the average[2] memory power consumption of benchmarks in L, H and X workloads under a 4-core setup, and the average memory power they would consume if they ran in isolation. The figure has 4 sets of columns. From left to right: L workloads,

[2]In fact, we use the harmonic mean for power in Figure 5.4 and 5.5 to take into account that slower (and lower power) programs run longer. Otherwise, we could not compare power and memory energy per instruction values fairly.

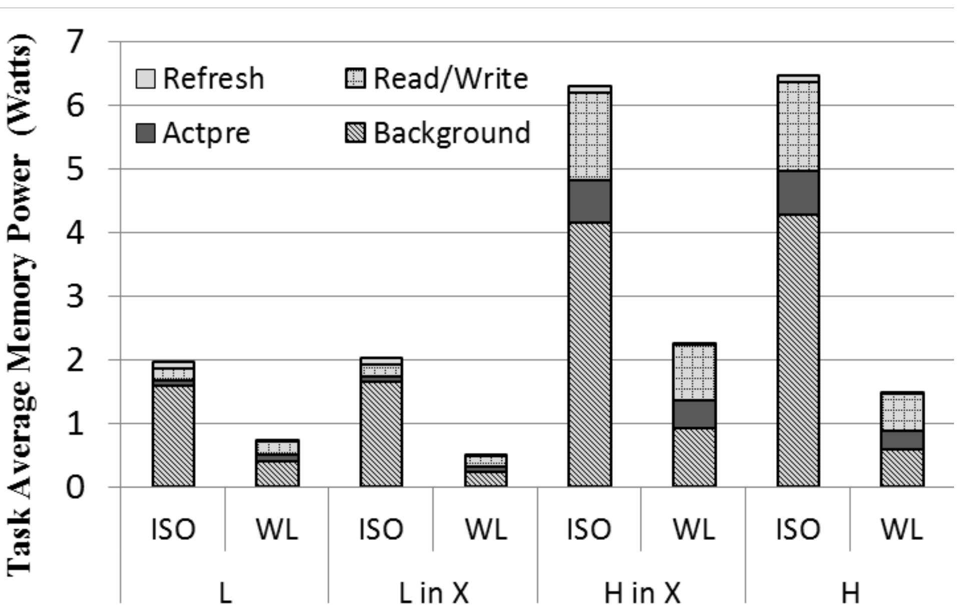


Figure 5.5: All workloads power consumption comparison in a 16-core setup.

L benchmarks in *X* workloads, *H* benchmarks in *X* workloads, and *H* workloads. For each set of columns, there are two columns labeled as *ISO* and *WL*. WL data shows the average per benchmark data in the corresponding category. For instance, the WL column in the *L* category shows the average memory power consumption per benchmark for the 32 benchmarks in those workloads (8 workloads with 4 benchmarks each). The ISO column corresponds to the average power of those 32 benchmarks when run in isolation. Note that separating results across benchmarks in workloads would not be possible without `DReAM`.

The first observation is that simultaneously running benchmarks in a multicore system decreases their individual memory power consumption. This fact is particularly noticeable for *L* benchmarks, whose average memory power has been decreased to less than half. Power consumption of *H* benchmarks decreases as well, but less than for *L* benchmarks. We also observe that those trends for *L* and *H* benchmarks hold independently of whether they run with benchmarks with similar or different characteristics in terms of memory access frequency.

The second observation is that, as expected, active power (activate, precharge, read and write) remains roughly constant regardless of whether benchmarks run in isolation or simultaneously with other programs. However, background and refresh power decreases remarkably since it is shared across benchmarks in the workload. In particular, *L* programs observe a significant reduction in terms of background power when running with other programs since they keep memory in *PD* state most of the time, and PD power is shared

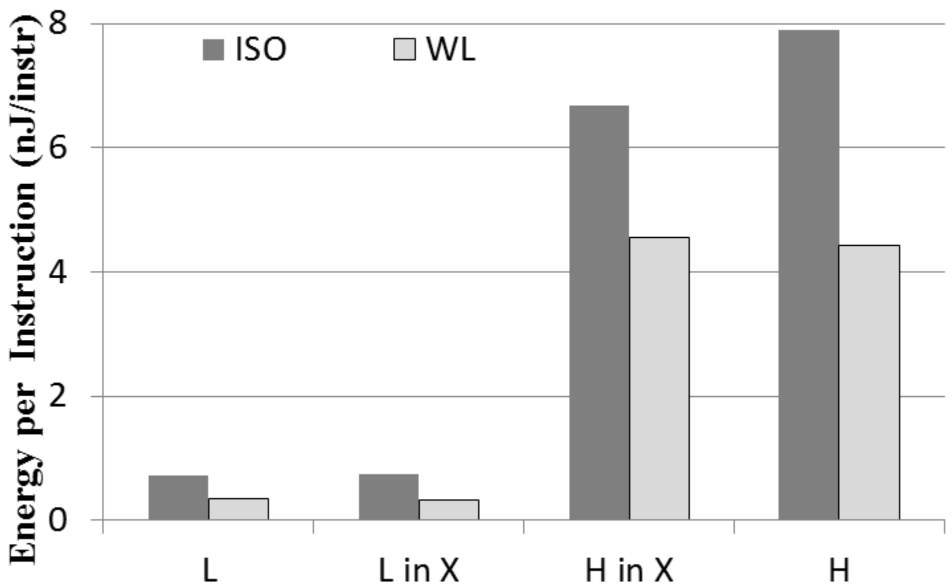


Figure 5.6: Average benchmark MEPI comparison in a 4-core setup.

homogeneously across running tasks. Conversely, *H* programs experience a lower reduction in terms of background power because background power during A and S states is the main source. This is so because accesses from different programs do not overlap often in time, and when they do, it is often the case that they need the same bank and thus, occur serially. Therefore, background energy due to A and S states is very similar in the workloads and in isolation.

Results for the 16-core setup, shown in Figure 5.5, resemble those for the 4-core setup with two main differences: (1) average memory power per program further decreases for the 16-core setup since power sources are shared across a larger number of programs; And (2) active power (activate, precharge, read, write) decreases for *H* benchmarks because energy for those operations remains constant, but since memory contention increases execution time, power decreases.

This second effect can be better observed in Figure 5.7, where the Memory Energy Per Instruction (MEPI) across workloads is shown. The MEPI of each benchmark for multi-programmed workloads is lower than for executions in isolation, but the ratio is not as favorable as in terms of power for *H* benchmarks. This is due to the longer execution time produced by banks conflicts, memory access contention and limitations on the number of simultaneously opened banks [52], which increases overall background and refresh energy, thus increasing the MEPI.

Figures 5.6 and 5.7 show the MEPI for 4-core and 16-core setups respectively. We

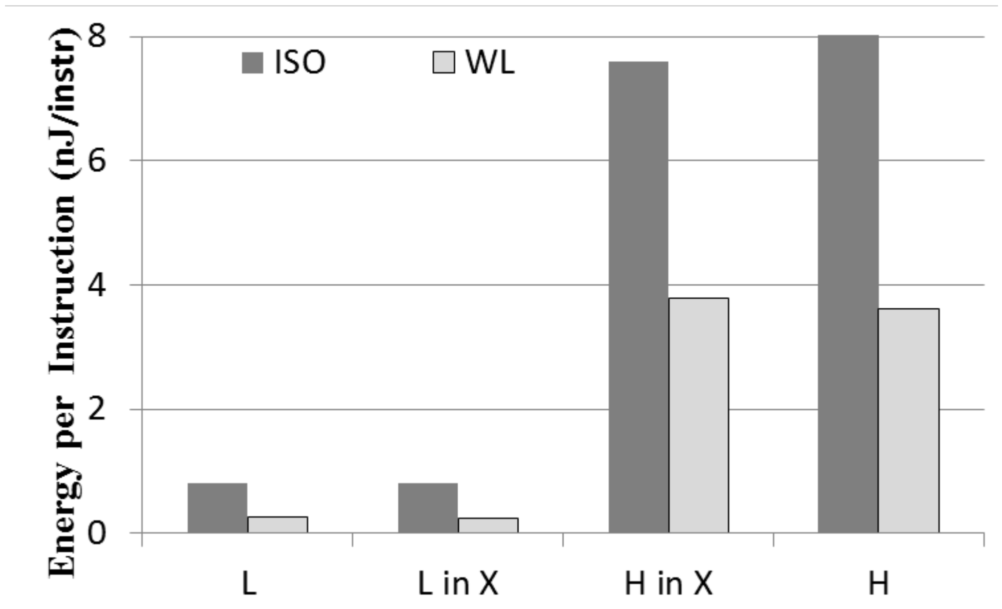


Figure 5.7: Average benchmark MEPI comparison in a 16-core setup.

observe that MEPI ratios between WL and ISO remain the same as for power for all workload types in the 4-core setup and *L* workloads in the 16-core setup. This is so because the impact in execution time due to memory contention is negligible. However, *H* workloads and *H* benchmarks in *X* workloads in the 16-core setup experience some MEPI increment due to contention with concurrent memory requests, which increased execution time, and so background and refresh energy. Note that power and energy for *H* (*L*) workloads and *H* (*L*) benchmarks in *X* workloads differ simply because benchmarks have been picked randomly and therefore, those sets contain different benchmarks (still of the same type). The same happens when comparing the MEPI in isolation in different processor setups.

5.6.2 Per-Benchmark Analysis

In this section we dig into the behavior of individual benchmarks in different workloads. `DReAM` enables this study, which could not be done otherwise. For that purpose, we picked the workload with the most varying behavior with respect to the average for each of the workload types (*L*, *X* and *H*) and core count (4 and 16), for a total of 6 workloads. In many cases, the most-varying behavior workload does not show big discrepancies with the average behavior for most of the benchmarks.

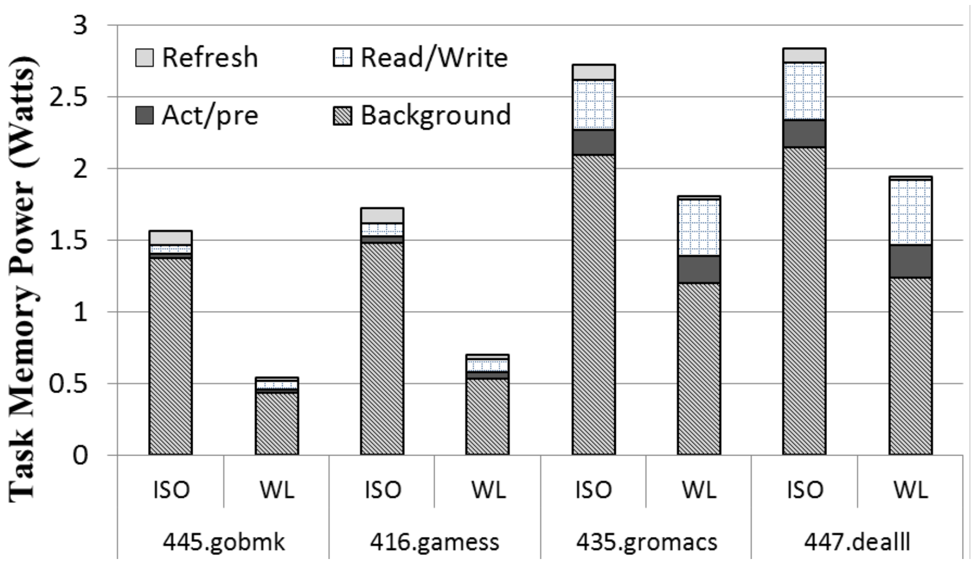


Figure 5.8: *L* type workload power consumption comparison in a 4-core setup.

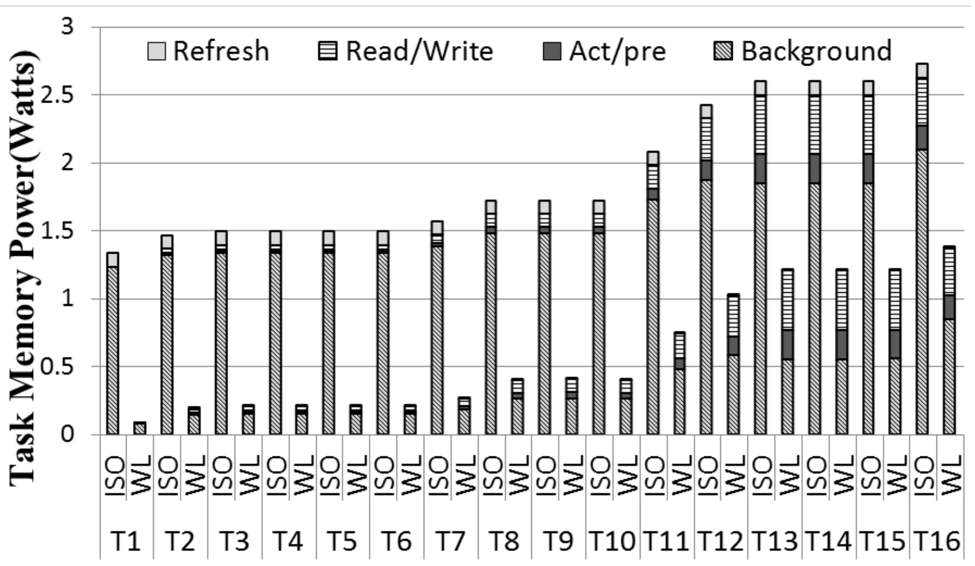


Figure 5.9: *L* type workload power consumption comparison in a 16-core setup.

L **Type Workloads** Figure 5.8 shows the power consumption in an *L* type workload with 4 cores. As shown before, power is reduced to less than half on average for *L* workloads in comparison with the ISO case. However, when we analyze benchmarks individually, we observe that those benchmarks with higher memory access frequency (*gromacs* and *dealII*) have higher WL case power consumption. This is so because workloads are not fully homogeneous and discrepancies in the memory access frequency lead to higher background power for those programs keeping the memory in a higher power consuming state longer.

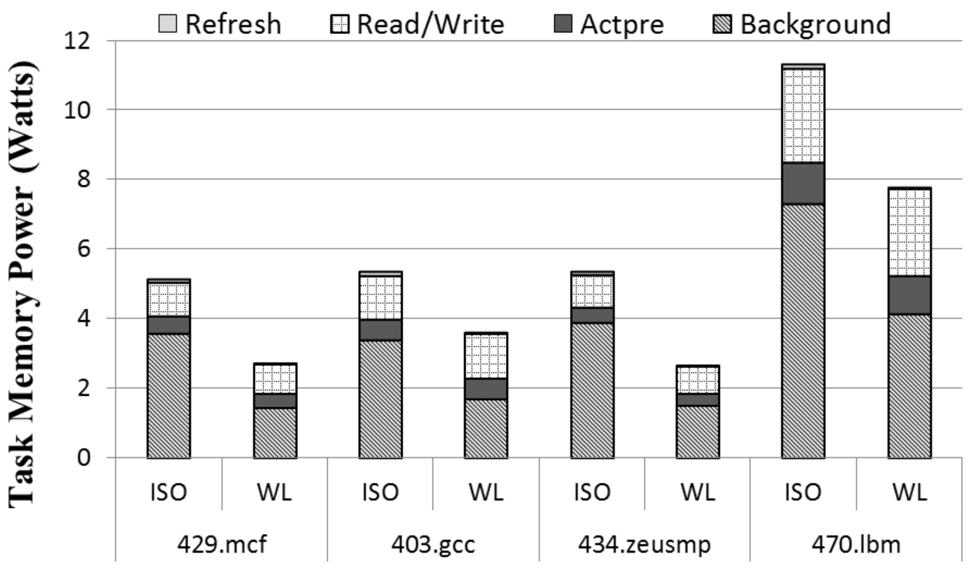


Figure 5.10: *H* type workload power consumption comparison in a 4-core setup.

The fact that PD state background power is very low makes programs with a relatively higher memory accessing frequency increase their background power noticeably in relative numbers. Therefore, they are responsible for a larger fraction of the total energy consumption (and so of the power consumption). Active power remains basically the same for ISO and WL since energy per access is constant and execution time barely changes.

Results for an *L* workload in a 16-core setup are shown in Figure 5.9. Trends are analogous to those reported for the 4-core setup with the only difference that power reductions are larger as already pointed out for the average results across all workloads.

H **Type Workloads** Figure 5.10 shows the power consumption in an *H* type workload on a 4-core setup. We can observe that, as on average, power decreases moderately in the WL case with respect to the ISO case. Analogously to the trends in *L* workloads, the higher the memory access frequency, the lower the power reduction in the WL case since access frequency strongly correlates with background power. This is the case for benchmark *lbm*, whose power consumption decreases only by around 40% instead of the average 55% for the whole workload. In a 16-core workload in Figure 5.11 we also observe similar trends as those in the average case. This is expected because *H* workloads are much more homogeneous than the others (*L* and *X*) since relative variations in accessing frequency across benchmarks is low (all of them access memory at least 5 times every 1000 cycles in isolation). Again, we observe that power in WL is much lower than in ISO, and such power decrease is much higher than for the 4-core case.

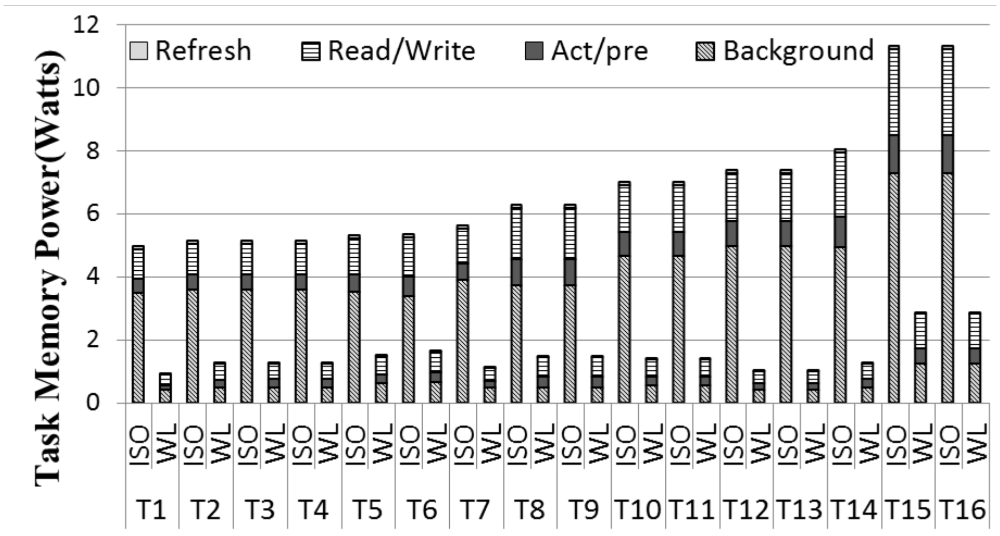


Figure 5.11: *H* type workload power consumption comparison in a 16-core setup.

X **Type Workloads** Figure 5.12 shows a 4-core *X* workload. In this workload, *bzip* and *soplex* are *H* programs whereas *gromacs* and *gamess* are *L* programs. Notably, the same trends observed in pure *H* and *L* workloads still hold for each *H* and *L* benchmark in *X* workloads. As expected, *soplex* is the program experiencing a lower power reduction when moving from ISO to WL due to its high access frequency. In the 16-core setup (see Figure 5.13), those trends still hold. Only *T11* behaves differently since its power reduction in WL is not as significant as for the other benchmarks with similar access frequency. The reason is that this program accesses memory frequently (therefore its active power is high), but it does it in bursts, so that the amount of time that DRAM devices are imposed to be at high power states (active or standby) is relatively low, and it makes its ISO background power low (e.g., compared to that of *T10* or *T12*). Therefore, its relative background power reduction in the WL case cannot be as significant as for other benchmarks with similar average access frequency but with different access patterns.

We do not further discuss the MEPI for those particular workloads since the conclusions are similar as those for the power.

We have shown that multicore architectures help reducing per-task memory power and energy. Energy savings are more significant for those programs with lower memory access frequency on higher core count setups, and trends do not change across workloads. Furthermore, exceptions do not deviate much from the average case, and when they do, it is because of their access patterns (burst versus scattered).

We have also shown that the impact of memory contention highly correlates with the

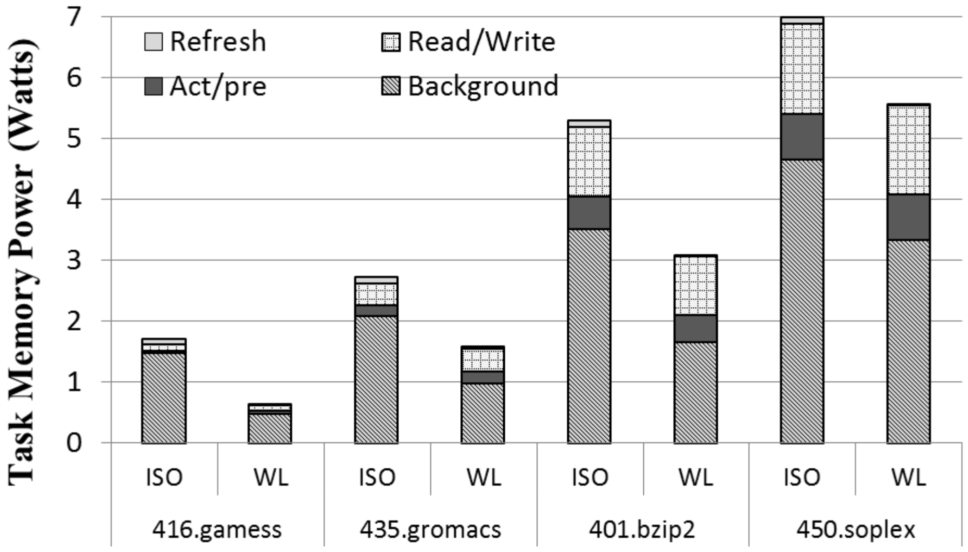


Figure 5.12: X type workload power consumption comparison in a 4-core setup.

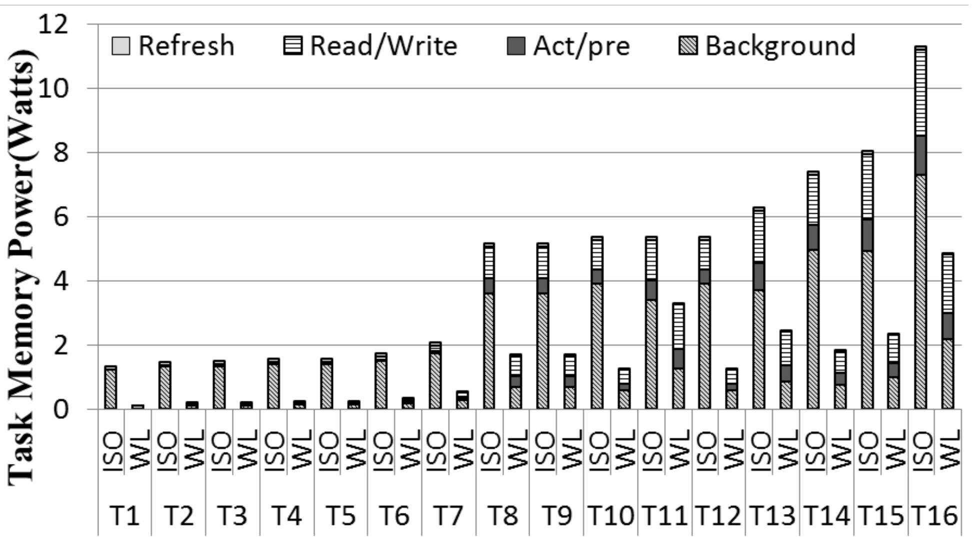


Figure 5.13: X type workload power consumption comparison in a 16-core setup.

accessing frequency of benchmarks. Our results show that high-access-frequency programs decrease their power at the expense of increasing the energy. Our study proves that memory energy profiles are quite stable for applications despite the programs running simultaneously. Besides, it is preferable to run H programs with L programs to reduce the negative impact of memory contention in terms of energy consumption (once discounted LLC interferences). This information is very useful to perform task scheduling on multicore setups.

5.7 Summary

Per-task energy metering is needed in multicores to enable a number of performance/energy optimizations. In this chapter, we propose the PTEM models in DRAM memory system. Including ideal models for both close-page and open-page policies and devise `DReAM`, an efficient and accurate implementation of such ideal model. We show how `DReAM` achieves a prediction error between 3.9% and 4.7% with respect to the ideal model with negligible overhead for 4- and 16-core setups respectively. The error is largely below the error introduced by approaches such as distributing energy evenly or proportionally to memory accesses.

Next, we have shown that multicore architectures help reducing per-task memory power and energy. Energy savings are more significant for those programs with lower memory access frequency on higher core count setups, and trends do not change across workloads. Furthermore, exceptions do not deviate much from the average case, and when they do, it is because of their access patterns (burst versus scattered).

Finally, we have proved that the impact of memory contention highly correlates with the accessing frequency of benchmarks. Our results show that high-access-frequency programs decrease their power at the expense of increasing the energy. Our case study proves that memory energy profiles are quite stable for applications despite the programs running simultaneously. Besides, it is preferable to run H programs with L programs to reduce the negative impact of memory contention in terms of energy consumption (once discounted LLC interferences). This information can be very useful to perform energy-efficient task scheduling on multicore setups.

6

Sensible Energy Accounting for the Processor

6.1 Introduction

Energy is becoming the most expensive resource in computing systems and this trend will continue as the price of energy continues to rise (increasing in recent years by up to 70% in several European countries [28]). Under these circumstances, metering energy consumption of a computing system enables energy optimizations and hence ultimately helps to reduce system operation costs. In a datacenter or supercomputing setting, charging users for energy rather than time makes sense because energy usage is more proportional to the cost of operations. The establishment of multicore and manycore as the *de facto* hardware paradigm across most computing domains, together with increasing core counts in each new generation, highlights the need for energy metering. Furthermore, applications are increasingly diverse, with many different providers and quite different energy profiles. Thus, accurate energy metering and optimization techniques are essential.

In this chapter, we make the case for Sensible Energy Accounting(SEA), as introduced in Section 1.1.2. In contrast to PTEM, SEA does not give the actual energy consumption of a task, but rather an abstraction of the energy consumption that the end-user can rely on to be fair and consistent.

Let us illustrate the concept of SEA and how it differs from PTEM with an example. We simulate several SPEC CPU 2006 benchmarks on a 4-core multicore architecture com-

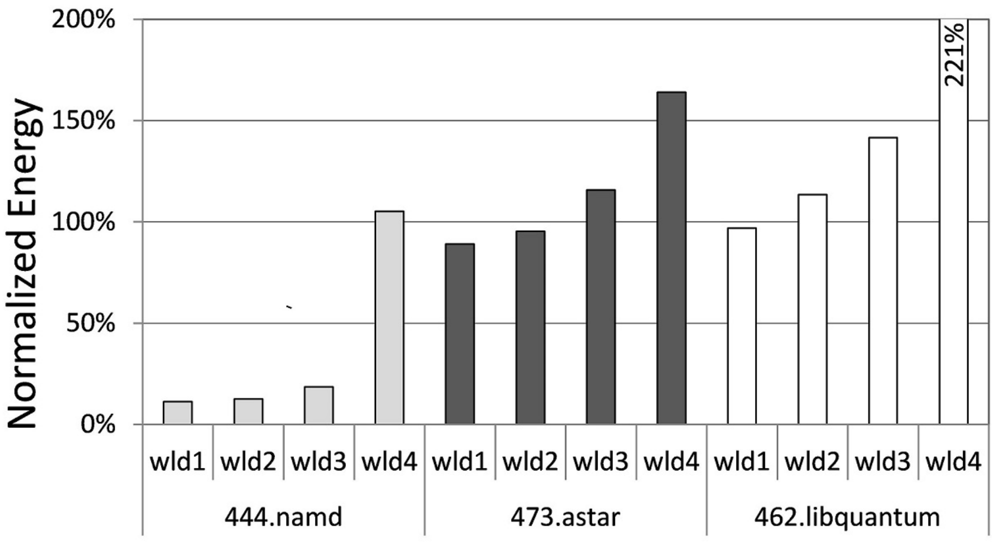


Figure 6.1: Energy usage of *namd*, *astar*, and *libquantum* in different workloads w.r.t their energy usage when executed in isolation with a fair share of resources.

prising a shared last-level cache[1] and the PTEM technique is introduced in chapter 4. We choose *namd*, *astar* and *libquantum* benchmarks since they have different LLC utilization levels. We run each benchmark as part of 4 different 4-task workloads. The other 3 tasks in the workloads are only considered as co-runners, affecting the LLC behavior of the target benchmark. For instance, workload 1 comprises 3 copies of *namd*, which will cause almost no conflict to the target benchmark in the LLC. In contrast, workload 4 comprises 3 copies of *libquantum*, which makes the most intensive LLC use across those benchmarks. Workloads 2 and 3 have a mix of benchmarks to show some intermediate points in terms of LLC contention. Figure 6.1 shows the energy metered to the target benchmark in the workload, which is normalized to the energy the benchmark consumes when it runs in isolation with a fair-share of the cache (i.e. 1/4 in our case). We observe that, despite the fact that each benchmark executes exactly the same instructions in each run, the energy it consumes significantly varies depending on the co-running applications. Sometimes the benchmark consumes much more energy, up to 2.2x, than when it runs in isolation with 1/4 of the cache, and other times it consumes as little as 11% of that.

This inconsistency is particularly problematic in environments where users are charged for the usage of resources including energy. Users running the same applications with the same inputs would observe different energy profiles for their applications and hence would

[1]The experimental setup is described in Section 3.2.

unfairly receive different amounts billed. SEA helps by providing, for every task in a workload, the energy it would have consumed if run in isolation with a fair share of the shared resources.

The energy charged is not exactly the energy consumed, but it is far more fair for end users (their billing solely depends on their own tasks) and still appropriate for the data center operator since typically actual energy consumed is lower than energy accounted due to using non-partitioned shared resources. Note that those energy savings for the operator can be shared with end users by applying discounts for a mutual benefit. In this case, we assume that $fhr = 1/N$, where N is the number of hardware threads (cores in this case) in the system. The best value of fhr may vary across domains as shown in the following sections.

In this chapter, we develop the concept of SEA from a theoretical point of view and discuss how it can contribute to different computing domains. Secondly, focusing on the on-chip resources, we present a low-overhead hardware mechanism to obtain SEA for a shared last-level cache in a multicore architecture. Our results show that SEA allows saving up to 39% of energy if used for scheduling purposes. Finally, we present a SEA mechanism for on-core resources taking into account SMT. Our results show that prediction error is only 5% on average for the core and between 4% and 8% on average for the whole chip when using SMT cores and a shared last-level cache. We also show how SEA attains much higher accuracy than other state-of-the-art mechanisms such as evenly splitting the energy across tasks or distributing it based on several metrics (number and type of instructions, etc.).

The rest of this chapter is organized as follows. Section 6.2 provides background on performance accounting and reconfigurable computing. Section 6.3 explains our theoretical approach towards SEA. Section 6.4 presents SEA for a shared on-chip cache and the core resources, while Section 6.5 presents an evaluation of SEA shared cache and cores resources and integrates them to cover the whole chip. Section 6.6 describes a case study and, finally, Section 6.7 draws the main conclusions of this chapter.

6.2 Background

SEA comprises two main building blocks: PTEM techniques and performance (CPU) accounting techniques. In this section we elaborate on the state of the art for both.

Table 6.1: PTEM and performance accounting in two workloads

	h264ref	*calculix*	*povray*	*namd*
PTEM, EPI(nJ)	0.41	0.25	0.39	0.27
CPU utilization	68%	83%	75%	64%

	h264ref	*milc*	*sjeng*	*gcc*
PTEM, EPI(nJ)	0.73	0.70	0.43	0.82
CPU utilization	24%	86%	45%	75%

6.2.1 Per-Task Energy Metering

As energy costs rise, interest in energy metering continues to increase in different computing domains from datacenters to smartphones [17, 90, 92]. PCEM techniques that we introduced in section 2 focus on single-core architectures or multicores in which only one application is executed at one time and provide per-component energy estimations. However, processors incorporate an increasing numbers of cores, each implementing SMT, and running several applications with different energy profiles.

In this scenario it is essential to determine energy consumption for each task. Shen *et al.* [102] proposed a request-level OS mechanism to meter power consumption of each server request based on PMCs [7]. The authors consider both active and maintenance power and attribute it to the responsible server requests. However, per-task energy estimates obtained with this approach cannot be validated since, as stated by the authors, *"Request executions in a concurrent, multi-stage server contain fine-grained activities with frequent context switches, and direct power measurements on such spatial and temporal granularities are not available in today's systems"*.

PTEM cover this gap by proposing new hardware support to measure the *real* per-task energy in multicores systems by tracking utilization of hardware resources for each task, including activities they have incurred and the fraction of resources they have used, to determine their fraction of energy used. Results show that under different workloads, the variation of metered energy to some particular tasks can vary up [−25%,40%] with respect to their average energy.

6.2.2 Performance Accounting in Multicores

The concept of SEA is inspired by CPU accounting [76] developed for multicores [77] and for SMT cores [29, 30, 78]. CPU accounting measures the CPU utilization of a given task during a period of time when it runs on a multithreaded processor. CPU utilization depends on both the time the task is scheduled on the CPU and the progress (or slowdown) the task experiences with the multicore. The latter is computed by determining which accesses to shared resources of a given task are delayed due to conflicts with other running tasks. For instance, if a task runs for a period of 1,000 cycles in which it suffers a slowdown of 30%, its progress is 70% of what it would be w.r.t. its execution with a fair share of the resources. Thus, it is only accounted $1,000 \times 0.7 = 700$ cycles.

Performance accounting has been shown to be a powerful tool for performance optimization. For instance, it can be used to predict the performance with different degrees of contention to co-locate applications within the system. Results show that individual application's performance can be improved by up to 22% and system utilization can be increased by 50% to 90% [80, 81, 107].

Using CPU accounting to scale energy estimated by PTEM as a way to achieve sensible energy accounting leads to inaccurate results. For instance, instruction mix and data locality have large impact on energy that cannot be distinguished with CPU utilization.

To illustrate this point consider the execution of benchmark *h264ref* under two different 4-task workloads as shown in Table 6.1. In the first workload *h264ref* incurs an Energy-Per-Instruction (EPI) of 0.41 nanojoules (nJ) and it is accounted 68% of CPU utilization, while in the second workload, *h264ref* incurs 0.73 nJ EPI and accounts for 24% CPU utilization. One intuitive way to scale energy is to map CPU utilization to resource utilization. In this case, this method estimates that under any resource utilization ru and EPI *h264ref* would incur $SEA_{ru} = N_{ins} * ru * EPI$ (where N_{ins} stands for the instruction count). So in the first workload $SEA_{0.68} = N_{ins} * 0.279 \quad (0.41 * 0.68)$ and in the second $SEA_{0.24} = N_{ins} * 0.175 \quad (0.73 * 0.24)$. As shown, the discrepancy across energy estimates in different workloads is huge across workloads (around 60%) if only CPU accounting is used and thus, sensible energy accounting is needed.

6.3 Theoretical SEA

In this section we introduce our theoretical approach towards SEA showing some cross-domain applications of SEA and present the scenario considered in the rest of the chapter.

6.3.1 Theoretical Approach to SEA

SEA estimates an accounting for each task T_i while it runs with other tasks (i.e. as a part of a workload), the energy it would have consumed, $E_{T_i}^{fhr}$, if it had run in isolation with a certain fraction of hardware resources, fhr. Note that, in this abstract model, when running in isolation, T_i would be granted access to that fraction of resources, but is prevented from using more, although with shared resources T_i's usage may be more.

Interestingly $E_{T_i}^{fhr}$ has to be estimated while T_i runs simultaneously with other tasks. In varied workloads, T_i can receive more or fewer resources than fhr, depending on co-runners. SEA must provide an accurate $E_{T_i}^{fhr}$, regardless of the particular usage of hardware resources that other tasks have[2].

Note that SEA's accounting model is conservative. It is possible that a given task may negatively affect co-running tasks by e.g. thrashing the cache. In this case SEA's abstract metering model would assign an overall energy cost to the tasks that is less than the actual cost to the provider. For this work, we assume that such situations would be dealt with by other means, e.g. migrating cache-thrashing or other misbehaving tasks to cores where they can do less damage.

Problem Statement. Let's define $\mathcal{W}$ as a set of workloads composed of N tasks, in which a given task T_i is always present. Further define $W_j \in \mathcal{W}$ as $W_j =< T_i^{W_j}, T_{j_1}^{W_j}, \ldots, T_{j_{N-1}}^{W_j} >$, where $T_i^{W_j}$ corresponds to the actual execution of T_i in the workload W_j, and $T_{j_k}^{W_j}$ are any other tasks executing in the workload.

In this scenario, the energy accounted to task T_i in a workload W_j, $E^{fhr}(T_i^{W_j})$, has to be as close as possible to the energy consumed in isolation with the same resource usage fhr by this task, $E_{T_i}^{fhr}$. This means that with SEA, for any workload $W_j \in \mathcal{W}$, we expect that $E_{T_i}^{fhr} = E^{fhr}(T_i^{W_j})$.

Next we illustrate two concrete applications of SEA, one of them particularly suitable for environments in which users are charged by the use of resources they incur and a second suitable across multiple domains.

Billing. When billing users for their use of resources, it is desirable to ensure that the same execution of the same application with the same input data result in the same charge. However, as shown in Figure 6.1, the energy consumed by a task can vary drastically depending on the co-runners. In this scenario, SEA can be deployed with $fhr = \frac{1}{N}$, where N is the number of hardware threads (i.e. the number of cores in a multicore processor) so that fhr corresponds to a fair share of the resources. Each task T_i is always charged $E_{T_i}^{1/N}$ which is independent of the actual energy consumed by the task, since the latter depends on T_i co-runners. If the actual energy consumed when running a workload E_{wld} is smaller than the energy accounted $\sum_{i=1}^{N} E_{T_i}^{\frac{1}{N}}$ the owner of the data center benefits from the $\left(\sum_{i=1}^{N} E_{T_i}^{\frac{1}{N}} - E_{wld}\right)$ energy not actually consumed. This encourages the datacenter

[2]The SEA hardware support proposed in this work is able to estimate the energy a task should be accounted under several values of fhr at once, not just one. For the sake of clarity we will be talking about a single fhr value without loss of generality.

Table 6.2: Synthetic example of energy consumption (in arbitrary units) under different fractions of resources

	$E^{1/4}(T_i)$	$E^{2/4}(T_i)$	$E^{3/4}(T_i)$	$E^{4/4}(T_i)$
T_1	1.7	1.4	1.0	1.3
T_2	1.1	1.0	1.1	1.3

owner to apply SEA, while the user enjoys workload-independent accounting. In our view, if $E_{wld} > \sum_{i=1}^{N} E_{T_i}^{\frac{1}{N}}$ it should be the data center owner taking this extra cost, since assigning it to any task or proportionally to all tasks will break the principle of workload-independent energy accounting. As mentioned before these situations can be prevented by for instance properly allocating cache trashing tasks.

Energy optimization. Energy efficiency is pursued in all computing domains. Predicting the energy consumed by each task (or the system as a whole) under an arbitrary workload *a priori* is complex due to the many different ways the tasks composing the workload can interfere with each other. SEA can help in this respect. As we show later, SEA hardware support allows predicting the energy consumed by each task with an arbitrary fraction of the resources (fhr). For a discretized number of m valid values $\mathcal{F} = \{fhr_1, ..., fhr_m\}$ for fhr, SEA can predict the energy consumed by any task with any of those fractions of resources, resulting in m estimations. If this is done for every task in the workload we can identify the resource partition that minimizes the total energy consumed by all tasks: $FHR_{min} = \min \sum_i E_{T_i}^{fhr_{i_j}}$ with $\sum_i fhr_{i_j} = 1$[3], and $i_j \in [1, N]$. Note that partitioning of shared resources is not needed by SEA. This example assumes it as a way to implement this optimization.

For instance, let assume a 2-core processor with single-threaded (i.e. non-SMT) cores comprising a shared 4-way last-level cache implementing way partitioning. Further assume two tasks T_1 and T_2 so that energy consumption under each different fraction of LLC is as shown in Table 6.2. We can see that total energy is minimized when $FHR_{min} =< 3/4, 1/4 >$, as this leads to a total energy of 2.1 units. Any other partition leads to higher energy consumption. Also, if tasks are given the whole LLC space and executed serially, energy would also be higher (2.6 units) than for FHR_{min}.

6.3.2 SEA for On-Chip Resources in Multicores

SEA can be applied to any component of a computing system. In this chapter we focus on on-chip resources in multicore processors, since the CPU is one of the major energy-

[3]Note that we could distribute less than 100% of the resources, but for the sake of simplicity we assume that all resources are used by running tasks.

consuming hardware blocks. In particular we focus on a homogeneous multicore architecture deploying a shared last-level cache as the one described in section 6.5.

SEA, as shown later, incurs some hardware overheads. As a result SEA must be applied judiciously, taking into account the tradeoff between accuracy in the energy predictions and hardware cost. With that goal, on the one hand, we only apply SEA to those resources that account for most of the energy consumed on-chip. We first consider the LLC of multicores. In a second step, we consider SMT cores whose resources are shared (i.e. the core itself, L1 data and instruction caches). On the other hand, accounting the energy for all possible fractions of resources would be infeasible. Hence, we focus on a set of predefined fractions. We consider each resource as a separate entity with a set of predefined granularities that represent the relative amount of resources assigned. In general, we will have granularities $g = \frac{M}{N}$, where $M \leq N$.

For the LLC, we consider only set-associative caches in this work, and define cache ways as the atomic granularity unit. For instance, in a 4-way LLC, N is 4, then, M is a integer in the range of $(0, 4]$. $\frac{1}{4}$ LLC for task T_i means T_i can use 1 way in each set of the LLC. Note that, although SEA partitions the resources for accounting purposes, this is applied only to an abstract model to estimate energy consumption. SEA can target either shared or partitioned resources.

For the core, we use the fetch bandwidth as N, so that fetch bandwidth determines the partition granularity. Then, all other resources in the core, including all hardware blocks and bandwidths, are partitioned with the same degree. For instance, in an SMT core fetching up to 4 instructions per cycle, if T_i is given $\frac{1}{4}$ of the core, it receives $\frac{1}{4}$ fetch bandwidth, $\frac{1}{4}$ registers, $\frac{1}{4}$ issue queues entries, $\frac{1}{4}$ L1 ways, etc. By doing so we have a limited number of possible partitions for each hardware resource and their granularities facilitate the hardware implementation of such partitions.

The main challenge for SEA is how to compute $E_{T_i}^{fhr}$ for any task and any valid fraction of the resources. In the next sections we present our approaches in steps, first for a multicore processor where only the LLC is shared, and then for a processor where both core slices and LLC are shared. In both cases, we first propose an ideal SEA mechanism, and then we propose a efficient solution with hardware support that approximates such ideal values, assessing how our implementation of SEA performs in comparison with the ideal scenario.

6.4 SEA for Multicores: LLC and Core

This section presents our approach for SEA in the presence of a shared LLC and SMT cores. First, we describe an ideal SEA model for each. Then we propose the accurate, yet low-cost, implementations.

6.4.1 Ideal SEA for the LLC

As explained in Section 2.1, *dynamic active* energy is proportional to the number of LLC accesses performed by T_i. Maintenance energy and leakage energy are proportional to the time and the fraction of the LLC used by T_i.

Sensible LLC active energy accounting. The key insight to accurately account for active energy, E_{act}, is that each action type in the cache incurs different energy consumption. For instance, a write operation requires more energy than a read. Hence, in the ideal case, we should collect the number of events of each action type that a task experiences with a given fraction $\frac{M}{N}$ of the LLC space, denoted $\frac{M}{N}LLC$:

$$E_{act}^{\frac{M}{N}LLC}(T_i) = \sum_{j=1}^{ActionTypes} Num_{action_j}^{\frac{M}{N}LLC}(T_i) \times E_{action_j}^{LLC} \tag{6.1}$$

where $E_{action_j}^{LLC}$ stands for the energy per access to LLC of type $action_j$ (e.g., read-hit, write-miss, etc.). $Num_{action_j}^{\frac{M}{N}LLC}(T_i)$ is the number of LLC accesses of type $action_j$ performed by the task T_i if it is given M out of the N LLC ways.

The difficulty lies in estimating $Num_{action_j}^{\frac{M}{N}LLC}(T_i)$ for any valid value of M (number of cache ways) when T_i runs as part of a workload using a fully-shared LLC. This is so, because under each workload T_i may receive a variable number of cache space which affects the number of events of each action it has.

Sensible LLC maintenance energy accounting. The dynamic maintenance energy of the LLC is the energy consumed during idle periods due to useless activities such as, for instance, clocking and precharging bitlines when no access occurs. Potentially, LLC maintenance energy consumption could be avoided if we turn off unused LLC parts (e.g., banks, lines, etc.). The fact that they are used by tasks prevents us from turning them off, so we account maintenance energy proportionately to the cache space each task is entitled to use. Thus, maintenance energy to be accounted to T_i given a fraction $\frac{M}{N}$ of the LLC space is the same fraction of the total maintenance energy. Such total maintenance energy is the one that would be consumed assuming that the LLC is idle when T_i does not use it. Thus, maintenance energy is accounted as follows:

$$E_{main}^{\frac{M}{N}LLC}(T_i) = \frac{M}{N} \times P_{main}^{LLC} \times \Bigg(ExecTime^{\frac{M}{N}LLC}(T_i) - \sum_{j=1}^{ActionTypes} Num_{action_j}^{\frac{M}{N}LLC}(T_i) \times Latency_{action_j}^{LLC} \Bigg) \quad (6.2)$$

P_{main}^{LLC} is the LLC maintenance power, $ExecTime^{\frac{M}{N}LLC}(T_i)$ is the total time task T_i when executed with $\frac{M}{N}$ LLC ways, and $Latency_{action_j}^{LLC}$ stands for the latency of an action of type $action_j$. P_{main}^{LLC} and $Latency_{action_j}^{LLC}$ can be provided by the chip vendor. However, some parameters still need to be determined such as $Num_{action_j}^{\frac{M}{N}LLC}(T_i)$, which is also needed to account active energy, and the execution time that would be had with exactly $\frac{M}{N}$ LLC ways, $ExecTime^{\frac{M}{N}LLC}(T_i)$. Note that such execution time cannot be easily estimated from the actual execution time when running as part of a workload sharing the LLC given that inter-task interferences in the LLC may increase execution time, and T_i may use more than $\frac{M}{N}$ cache space, thus decreasing its execution time.

Sensible LLC leakage energy accounting. Finally, accounting leakage energy to T_i for a given fraction $\frac{M}{N}$ of the LLC space can be done based on the leakage energy per time unit, the fraction of cache space used and the execution time of T_i as follows:

$$E_{leak}^{LLC}(T_i) = \frac{M}{N} \times P_{leak}^{LLC} \times ExecTime^{\frac{M}{N}LLC}(T_i) \quad (6.3)$$

P_{leak}^{LLC} is the LLC leakage power. As for the maintenance energy, we need to determine $ExecTime^{\frac{M}{N}LLC}(T_i)$.

6.4.2 Ideal SEA for an SMT Core

Active, maintenance and leakage energy are accounted separately, as in the case of the LLC.

Sensible SMT core active energy accounting. Active energy depends on the number of actions performed in each hardware component by a task T_i. Therefore, ideally we would like to track the number of actions that would be performed by T_i in each resource if it was allowed to use $\frac{M}{N}$ of this resource exclusively. While defining $\frac{M}{N}$ of the resources is relatively easy for storage resources (e.g., caches, register files, issue queues, etc.), bandwidth resources (e.g., fetch bandwidth, issue bandwidth, etc.) can be split by allowing different tasks to use a fraction of the bandwidth [47]. However, other resources such as functional units may need to be split in a different way. Given a partition granularity of N, if a task is allocated $\frac{M}{N}$ of the resources, this bandwidth splitting can be achieved exactly by allowing this task to use *all* resources during M out of N cycles. Still, in order to provide homogeneous behavior, we do so by providing the closest fraction to $\frac{M}{N}$ every cycle. For instance, if we have 4 adders and a task is allocated $\frac{1}{2}$ of the resources, it will get 2 adders every cycle. Similarly, if there are 2 adders and a task is allocated $\frac{1}{4}$ of the resources, it will get 1 adder every two cycles.

Active energy is, therefore, accounted as follows:

$$E_{act}^{\frac{M}{N}LLC}(T_i) = \sum_{k=1}^{Res} \left(\sum_{j=1}^{Actions(k)} Num_{action(k)_j}^{\frac{M}{N}k}(T_i) \times E(k)_{action_j} \right) \tag{6.4}$$

Res stands for the number of different resources in the SMT cores, $Actions(k)$ for the number of action types in resource k, $Num_{action(k)_j}^{\frac{M}{N}k}(T_i)$ for the number of actions of type j performed by task T_i in resource k when given $\frac{M}{N}$ of this resource, and $E(k)_{action_j}$ for the energy of one action of type j in resource k.

Sensible SMT core maintenance energy accounting. In order to determine the maintenance energy to be accounted to one task T_i when given $\frac{M}{N}$ of the core resources, we use the same approach as in [72]. First, we classify resources into two different categories: occupancy-based ($oRes$) and non-occupancy-based ($nRes$). Maintenance energy for $oRes$ is accounted exactly as for the case of the LLC. Conversely, $nRes$ maintenance energy (e.g., selection logic in the issue queue when no instruction is ready) is simply split proportionally to the fraction of resources allocated. Thus, maintenance energy is accounted as following:

$$\begin{aligned} E_{main}^{\frac{M}{N}core}(T_i) \quad &= \textstyle\sum_{k=1}^{oRes} E_{main}^{\frac{M}{N}k}(T_i) + \\ &\frac{M}{N} \times \textstyle\sum_{k=1}^{nRes} \left(\textstyle\sum_{x=1}^{ExecTime^{\frac{M}{N}core}(T_i)} E_{main}^{\frac{M}{N}k}(x) \right) \end{aligned} \tag{6.5}$$

$E_{main}^{\frac{M}{N}k}(T_i)$ for $oRes$ is obtained as for the LLC (see Equation 6.2). $ExecTime^{\frac{M}{N}core}(T_i)$ stands for the execution time of T_i when given $\frac{M}{N}$ of the core resources and $E_{main}^{\frac{M}{N}k}(x)$ is the maintenance energy consumed by resource k in cycle x when T_i executes with $\frac{M}{N}$ of the resources.

Sensible SMT core leakage energy accounting. Leakage energy can be accounted using the same methodology as in the LLC. Given a fraction $\frac{M}{N}$ of the core resources, leakage energy accounted to task T_i derives from the core leakage power per time unit (P_{leak}^{core}) and the execution time of T_i with $\frac{M}{N}$ of the core:

$$E_{leak}^{\frac{M}{N}core}(T_i) = \frac{M}{N} \times P_{leak}^{core} \times ExecTime^{\frac{M}{N}core}(T_i) \tag{6.6}$$

6.4.3 Implementation of SEA for the LLC

The accounting mechanism introduced in Section 6.4.1 is based on the estimation of the number of LLC accesses of each type (for active and maintenance energy accounting) and execution time task of T_i (for maintenance and leakage energy accounting) with $\frac{M}{N}$ ways of the LLC. Next we describe affordable ways to approximate accurately those values.

Estimating access counts. Our approach to estimate the number of LLC accesses of each type when $\frac{M}{N}$ ways of the LLC are used relies on the Auxiliary Tag Directory (ATD) proposed by Qureshi and Patt [96], which focuses on a least recently used (LRU) replacement policy. The LLC is shared among all tasks each of which keep a local copy of the tag directory, the ATD, that is only updated with the accesses of the owner task.

If the LLC implements LRU, one can predict whether an access would hit in the LLC for any number of cache ways M lower or equal to the actual number of LLC ways (N). This is so because LRU keeps in each set the position in the LRU stack of each address, and so the order in which they will be evicted if they are not reused. For instance, if in a 4-way LLC we access addresses A, B, C, D such that they are placed into the same set, the LRU stack, from the most recently used (MRU) entry to the LRU entry is as follows: $< D, C, B, A >$, thus meaning that if a new cache line is fetched into this set A will be evicted.

Based on the LRU stack one can determine whether a given access would hit or miss with M ways (where $M \leq N$) by simply checking if it hits any of the M MRU entries. For instance, in our example, if we want to know whether accesses would hit in a 2-way cache given the LRU stack of the 4-way cache, we only need to check whether it hits in the 2 most recently accessed entries. In our example, only accesses to D and C would be hits. In general, we can set up $N + 1$ counters, $C_1, ...C_{N+1}$ so that C_i where $1 \leq i \leq N$ is incremented every time there is a hit in the way_i of any cache set, and C_{N+1} is incremented if X misses in all cache ways. Then, the number of hits and misses for $\frac{M}{N}$ ways of the LLC is obtained as:

$$Num_{hit}^{\frac{M}{N}LLC}(T_i) = \sum_{j=1}^{M} C_j \tag{6.7}$$

$$Num_{miss}^{\frac{M}{N}LLC}(T_i) = \sum_{j=M+1}^{N+1} C_j \tag{6.8}$$

If different types of accesses have different energy consumptions (e.g., read and write operations), then $N+1$ counters need to be kept by each operation type so that each access updates the counter corresponding to its type. In practice, pseudo-LRU replacement is commonly used for LLCs. Although the ATD has been devised originally for LRU caches, it has been shown to be highly accurate if pseudo-LRU is used instead [60]. Adapting the ATD to other replacement policies is left as future work and beyond the scope of this work.

Therefore, the ATD allows computing the number of accesses of each type ($Num^{\frac{M}{N}LLC}_{action_j}(T_i)$). However, keeping one ATD per thread may be over costly. Thus, the authors in [96] propose the Sampled ATD (SATD), which relies on keeping the tags only for a reduced number of the cache sets. For those sets it is also computed the overall hit probability for the different number of ways, $h_1, ..., h_N$, so that on an access to a set not present in the SATD, which will likely be the case of most accesses, can be predicted to be a hit or a miss. For that purpose, we use a *Monte Carlo* approach, that offers a high degree of accuracy and can be applied to each access at runtime. In particular, a random number RN is generated in the range $[0, 1]$. This random number, RN and the actual hit probabilities for each number of ways, $h_1, ..., h_N$, are used to decide whether the current access should be a hit or a miss under each number of ways. Given that increasing the number of cache ways can only increase the hit rate[4], we have that $h_i \leq h_{i+1}$ for $1 \leq i < N$. In order to mimic a given hit probability h (e.g., $h = 0.7$), we use RN such that the access is a hit if $RN \leq h$ and a miss otherwise. Thus, we have to find the value of k where $1 \leq k \leq N + 1$ so that $h_{k-1} < RN \leq h_k$. Such k value indicates that the access is a hit for caches with $M \geq k$. For instance, in our example of a 4-way cache we could have hit probabilities $0.2, 0.3, 0.7, 0.9$. If $RN = 0.6$ then $k = 3$ as RN is between h_2 and h_3, thus meaning that the access is assumed to be a hit if $M \geq 3$, so if the thread is given 3 or 4 LLC ways. Similarly, if $RN = 0.95$ then $k = 5$, thus meaning that the access is a miss for any number of ways in the LLC.

The SATD trades hardware cost for accuracy: the lower the number of sets sampled, the lower the cost but the lower the accuracy. The particular degree of sampling used for the SATD is indicated later in the results section.

Estimating the execution time with a given cache fraction. CPU accounting for multicores, introduced in [76], relies on using the ATD to decide whether each cache miss for a task T_i would hit or miss with a given fraction of the cache (typically a fair share of the cache space). A miss is caused by inter-task interferences if the access hits in the task's local ATD and misses in the LLC. In that case, if the processor stalls, the cycles

[4]Given a cache with X ways, increasing its size by any number of ways (Y) so that its total number of ways becomes $X + Y$, can only have a hit rate higher or equal than with X ways only. This is so because the LRU stack for the X ways closer to the MRU position in the $X + Y$ cache is *identical* to the LRU stack of the X-way cache. Thus, all accesses hitting in the X-way cache will hit in the X ways closer to the MRU position in the $X + Y$-way cache. Then, the remaining Y ways may provide some more hits.

needed to serve the miss are not 'accounted' to the task, meaning that the task would not suffer that miss, and hence the associated penalty, if it had run a given share of the cache. Similarly, this CPU accounting mechanism accounts extra cycles to T_i in case of an LLC hit that would have been a miss if T_i had run with a given fraction of the cache space .

This CPU accounting mechanism can be used to estimate the execution time that a task would have used to run with a given fraction of the resources, $ExecTime^{\frac{M}{N}LLC}(T_i)$. This helps estimating the maintenance and leakage energy for a task since they are affected by the time the task would run with a given fraction of the resources. Hence, we extend the CPU accounting mechanism for an N-way LLC to estimates the execution time of the task under any fraction of cache ways ($\frac{M}{N}$ where $1 \leq M \leq N$). CPU accounting uses the ATD as if the full cache is allocated to the task T_i. Cache accesses are considered to hit if they hit in the ATD, and to miss otherwise. In our case, we want to retrieve such information for different numbers of cache ways. The ATD provides such information by considering only those M entries closer to the MRU position. Thus, given a cache access we can determine whether it would hit in any cache with $1 \leq M \leq N$ cache ways by checking the M ATD entries closer to the MRU position. Then, we can use such information to perform CPU accounting simultaneously for all different cache sizes. For each task we need N cycle accounting (CA) registers, $CA_1, ..., CA_N$, which are updated as described in [77], but where the decision on whether an access should be a hit or a miss – and so how CPU cycles need to be accounted – for CA_M is done assuming $\frac{M}{N}$ cache ways. Finally, note that CPU accounting can be implemented on top of the SATD with the same pros and cons as for counting the number of events of each type.

Overall, hardware requirements of the SEA for the LLC approach include a SATD for each task, the minimal logic and registers for accounting the CPU cycles per task introduced by Luque et al. [77], and N+1 counters per task to obtain access counts for different numbers of LLC ways at once.

6.4.4 Implementation of SEA for an SMT Core

Tracking the activities of a given task T_i in all resources in the core is unaffordable. Instead, we propose periodically running a task T_i in isolation with a given fraction of the core resources and directly measure the energy, based on which we account the energy sensibly. Thus, we make use of the Micro Interval Based Time Accounting (MIBTA) approach introduced in [77], which has been used for performance accounting, and PTEM for per-task energy measuring to derive the accounting energy to T_i. MIBTA divides execution time into time intervals in which the execution of running tasks are sampled alone in turn.

During these sample phases, while one task has been granted the use of all resources in the core, the other running tasks are stalled temporarily. In our case, we need to carry out such sampling, but only allowing T_i to use $\frac{M}{N}$ of the core resources.

The purpose of using these approaches is to sample T_i's energy consumption periodically when it uses $\frac{M}{N}$ of the core resources alone. During the sampling phases, PTEM can be used to measure T_i's actual energy consumed in the core. PTEM provides accurate measurements of the active, maintenance and leakage energy consumption in the core, so their addition during the sampling intervals provides an accurate estimate of the energy accounting to T_i.

In the case to account active energy, the metered energy is nearly the energy that needs to be accounted. However, maintenance and leakage energy to account are corresponded to the fraction of maintenance and leakage energy of the whole core. Thus, SEA_{core} is estimated as follows:

$$E_{act}^{\frac{M}{N}core}(T_i) = \ P_{act,PTEM}^{\frac{M}{N}core}(T_i) \times ExecTime_{MIBTA}^{\frac{M}{N}core}(T_i) \tag{6.9}$$

$$E_{main}^{\frac{M}{N}core}(T_i) = \frac{M}{N} \times P_{main,PTEM}^{\frac{M}{N}core}(T_i) \times ExecTime_{MIBTA}^{\frac{M}{N}core}(T_i) \tag{6.10}$$

$$E_{leak}^{\frac{M}{N}core}(T_i) = \frac{M}{N} \times P_{leak,PTEM}^{\frac{M}{N}core}(T_i) \times ExecTime_{MIBTA}^{\frac{M}{N}core}(T_i) \tag{6.11}$$

$P_{act,PTEM}^{\frac{M}{N}core}(T_i)$, $P_{main,PTEM}^{\frac{M}{N}core}(T_i)$ and $P_{leak,PTEM}^{\frac{M}{N}core}(T_i)$ stand for the active, maintenance and leakage power respectively estimated by PTEM mechanism when running T_i during sampling periods. $ExecTime_{MIBTA}^{\frac{M}{N}core}(T_i)$ stands for the execution time predicted during the MIBTA phases when T_i is running with $\frac{M}{N}$ of the core resources.

Before entering the MIBTA phases (every 2.6 million cycles [78]), the execution of all tasks is stalled. Then, a controller restores the execution of a particular task to allow it run alone in the core for 50,000 cycles to warm up. When time is up, controller grants it another 50,000 cycles, during which some specified events are monitored to predict its execution time and energy consumed in such condition. The state of the other tasks is stored in the LLC when they get stalled, and their execution is restored after each MIBTA phase. In order to provide SEA_{core} capability, right after stalling the execution of the other tasks, the core is reconfigured to use $\frac{M}{N}$ resources. Adaptive processors (or reconfigurable processors) have already been studied in the past to reduce power consumption [4, 23, 47]. In each component, such as the branch predictors and the buffers [46], register files [1, 42], issue queues [18, 33, 93], caches [4, 96, 106], functional units, and fetch, decode and issue bandwidth [4, 23, 47], power gating techniques have also been proposed with minimal area and energy overheads to power down different sections, with negligible impact on the delay.

With these techniques that are already in place, in the cache-like blocks, SEA_{core} can assign $\frac{M}{N}$ of the ways to T_i during the MIBTA phases with the remaining ways power gated. Similarly, during the sample phases, T_i is only allowed to use $\frac{M}{N}$ entries in the SRAM-like components, such as the issue queues and renaming registers, etc. In contrast, non-occupancy-based blocks are reconfigured in a way that $\frac{M}{N}$ of the bandwidth and the resources can be used in every cycle. If this fraction cannot be applied exactly, it is enforced the closest value while still allowing T_i to progress. For instance, if T_i is entitled to use $\frac{1}{2}$ of the resources and there are 3 adders, it will be allowed to use either 1 or 2. In this case we break the tie providing the lowest value (1 adder) given that for some resource fractions can only be rounded up (e.g., if there is just 1 integer multiplier). SEA_{core} has considered ALUs, on-chip network bandwidth, as well as fetch, decode, issue and commit bandwidth. Note that during each MIBTA phase, some instructions may be squashed (i.e. when tasks are stalled to run one of them in isolation). They are reexecuted when the corresponding task is resumed since the program state (register contents) has been saved. In addition, the stalled task may have their used cache lines evicted by the running task, and thus incur extra cache misses. The result performance loss is detailed in [78] and described in later sections.

6.4.5 Putting It All Together

We have introduced the SEA proposals in LLC and SMT core separately, the correlation must be taken into account when integrate them. In general, there is no conflict on the configurations of SEA_{LLC} and SEA_{core}, in the sense that one can use any fraction of its resource. Note that SEA_{core} needs to account energy of each task in the core sequentially by sampling them one after another in a particular order. However, the SEA_{LLC} does not impose any constraint on how tasks must run to account their energy. Therefore, while MIBTA, needed by SEA_{core}, sample one task at a time in any particular core, this can occur while other tasks run in other cores. Thus, the overhead of serializing tasks execution for sampling is limited by the degree of multi-threading *in one core*, but not by the number of tasks in the whole processor chip. Therefore, one can sample tasks in different cores simultaneously in a way that scalability is not challenged when a large number of cores is in place.

Tasks interacting in the L1 cache have an impact on the number of LLC accesses, potentially causing inaccuracy in SEA_{chip}. To eliminate this effect, we monitor the number of LLC accesses per instruction during MIBTA phases when tasks run in isolation and thus have exclusive access to the L1 cache. The resulting LLC access frequency is assumed

Table 6.3: SEA hardware requirements

	Description	HW overhead (8 core)
(S)ATD	ATD with sampled sets LRU stack distance counter	Total of 1920B per task, *e.g.* 0.7% of the LLC space
ITCA	logic to determine IT misses logic to account CPU cycles	Negligible
Reconfig. core	Branch predictor and buffers [46], register file [42], issue queue [18, 33, 93], ALU, and fetch, decode and issue bandwidth [4].	Negligible
MIBTA	$CycleAccount_{MIBTA}$ $InstCommit_{MIBTA}$	2B per task 2B per task
PTEM	Energy Metering Registers Occupancy Counters	0.63% chip area overhead, 0.3% energy overhead [72]
SEA	**Energy Accounting Registers** **Target core and LLC resources**	**2 counters of 4B per task** **2 counters of 4B per task**

constant until the next MIBTA phase.

SEA hardware support and overhead. Regarding the hardware support incurred overheads, SEA mostly inherits them from PTEM and MIBTA, as shown in Table 6.3. Such overhead has been proved low, as can be seen in the same table with a 8-core configuration. Both PTEM and MIBTA require the SATD, whose area overhead is around 0.7% of the LLC [77,78,96]. Few extra registers are needed by PTEM and MIBTA with negligible area overhead. In terms of energy, overheads are largely below 1%, which have been reported for PTEM in Section 4.4.4, and they have been shown not to grow with the number of cores. MIBTA also introduces some performance overhead, which ranges between 1.0% and 3.2% [78]. Given that we have enhanced the MIBTA approach by allowing sampling tasks in all cores simultaneously instead of serializing task samplings across cores, the overhead is mildly reduced and does not grow with the number of cores. Our results show that MIBTA performance overhead remains around 2% on average regardless of the number of cores. In terms of energy, reconfiguring components in the core needs little extra logic to perform clock (or power) gating of unused parts during MIBTA monitoring periods. Such logic has been proven to have negligible area and power overhead and, in fact, it has been used to implement low power mechanisms sharing the costs [4,18,33,42,46,93]. Finally, SEA incurs very low overhead on its own due to those registers to store the accounted energy per task for the target core and LLC resources. Which we call Energy Accounting Register (EAR), that acts as the interface between PTEM and the OS.

Other considerations SEA may require considering temperature and voltage changes due to DVFS. We note that the LLC typically operates in a separate voltage domain as its voltage cannot be easily decreased. Memory cells are sized to maximize integration, thus small transistors are used which are highly susceptible to process variations requiring high voltage operation to read/write cells. Still, this is not a concern given that LLC active energy is low and idle banks are typically kept at lower voltages. Temperature variation is negligible in the LLC as its low activity keeps it at a mostly constant temperature.

Regarding the core, we note that DVFS becomes harder to use due to the need for decreased voltage for energy savings and increased minimum operating voltage to tolerate process variations [13]. As a consequence, the acceptable voltage range narrows down in each technology generation.

On the other hand, temperature variations in the core can occur. SEA can deal with voltage and temperature variations in both the core and the LLC by having as many energy constants (those that need to be provided by the chip vendor) as valid combinations of voltage and temperature ranges are allowed for the corresponding hardware block. For instance, if the processor can operate at 0.8V, 0.9V and 1.0V, and temperature ranges are discretized as 320K-330K, 330K-340K and 340K-350K degrees, then 9 sets of constants are required to update the energy accounted to the tasks depending on the current voltage and temperature. Conversely, the ATD (or SATD) and the logic to predict whether accesses would hit in cache do not need to be changed given that such information is voltage and temperature independent. Overall, the overhead of this approach is low as few hardwired constants need to be replicated.

Some Operating System (OS) support is needed to read energy accounting registers on a context switch. This issue is analogous to the case of PTEM. In particular, we must expose to software the EARs for each hardware thread so that on a context switch the OS can reset it when a task is scheduled in and read it when it is switched out, its value is aggregated to the corresponding task. On a context switch, the contents of the ATD (or SATD) will likely differ from those that would be had if the task was run to completion without being scheduled out. This might have some impact on SEA accuracy. However, we have verified empirically that tasks typically fetch their working set to different cache levels in less than 200,000 cycles, which is less than 0.1 ms in a processor operating at 2GHz. On the other hand, OS quanta vary from 4ms to 100ms for common Linux and Windows implementations, thus making context switch inaccuracy negligible – such inaccuracy falls below the inaccuracy of SEA method itself –. Moreover, many tasks are not scheduled out on a context switch, thus further reducing such inaccuracy.

The actions performed by the OS working on behalf of a given task (e.g., on a system call) are assumed to be part of such task, so the OS accounts such energy to that task. The energy accounted to other OS activities (i.e. 'housekeeping' activities) can be evenly distributed across all running tasks, although any other policy can be followed to distribute OS energy based on the EAR registers exported by SEA.

With such OS support, applying SEA to multi-threaded applications is simple since no additional hardware change is required. In fact, the OS can implement different mechanisms to account the energy to multi-threaded applications by reading EARs and interpret the values in different ways. We illustrate some of these choices with a simple example: let us assume a N-thread multi-threaded application running on a N-core CMP, where only the LLC is shared. In this case, we account each thread $E_{LLC}^{\frac{1}{N}}(t_i)$ as if the LLC is fairly shared across threads (cores) so that each one is given $\frac{1}{N}$ of the LLC. Upon the completion of one thread, the OS can choose to read EAR of that thread and add its value to the total energy accounted to the application. Then, the OS can keep accounting the remaining threads in the same way until they all finish. Alternatively, the OS can read the EAR values of all active threads upon the completion of one thread, and add those values to the application's accounted energy. Then, the OS can account the remaining threads until another one finishes by assuming that they have extra LLC space to use. For instance, when the first thread finishes each of the remaining threads will be accounted for $E_{LLC}^{\frac{1}{N-1}}(t_i)$ of the LLC space until another one finishes. The later approach is feasible as long as the thread completion and populating frequency do not exceed the OS quanta.

6.5 Evaluation

In this section we assess the accuracy of SEA estimations for the shared LLC and SMT cores. We also compare SEA with other intuitive methods that could be used to account LLC energy consumption, such as *ES* and *PTA* as introduced in Section 2.4, and PTEM in Chapter 4. The experimental setup is introduced in Section 3.4, while the benchmark suite and workload generation strategy is introduced in Section 3.4.3.

Metrics In order to evaluate the accuracy of SEA, we use as a reference the actual energy consumption of a benchmark when it runs alone with the corresponding resource fraction. For instance, if we aim to estimate the LLC energy of a benchmark when it has only half of the LLC ways, the reference is a single-core processor setup with an LLC with half of the cache ways where the benchmark runs alone. Hence, in each experiment, we measure the *prediction error* of each model with respect to the actual energy consumed

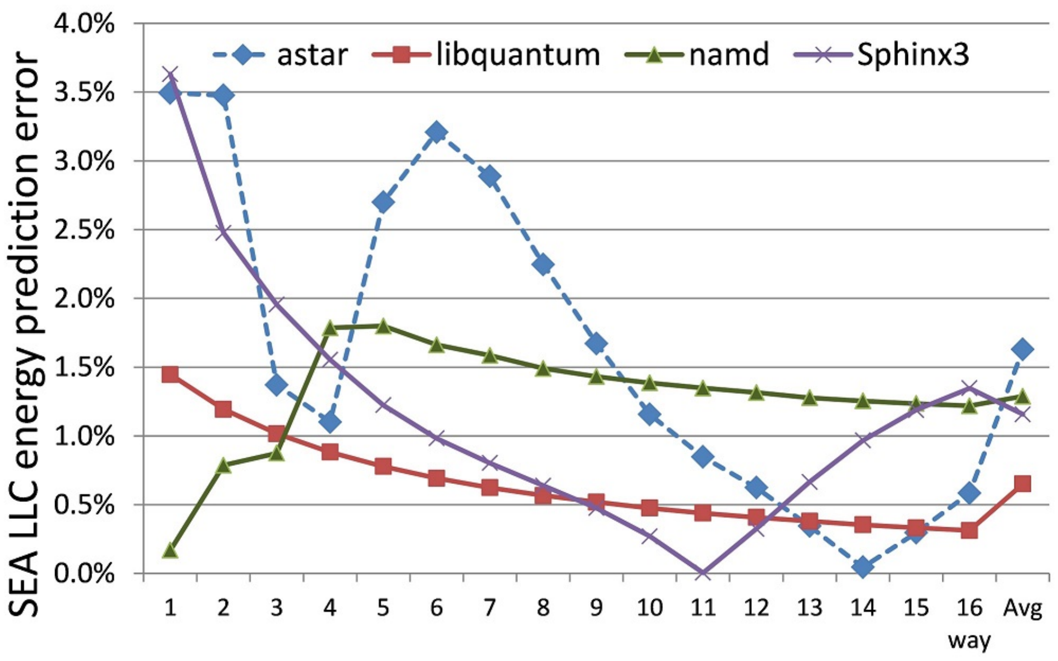


Figure 6.2: SEA_{LLC} prediction error for a workload consisting of benchmarks *astar*, *libquantum*, *namd*, and *sphinx3* in a 16-way associative LLC

when one task runs with the specified fraction($\frac{M}{N}$) of resources alone, which is computed with Equation 3.1.

6.5.1 SEA_{LLC} Accuracy Evaluation

In our multicore architecture with single-threaded cores the main sources of inter-task interferences are the LLC and the shared bus. Our results show that the latter has negligible consumption in our architecture so we do not consider it for SEA as it does not pay off the extra hardware requirements.

We start analyzing SEA results for a given 4-task workload consisting of the following benchmarks: *namd* that has few LLC accesses regardless of the space available; *astar* that accesses LLC often and whose LLC misses increase sharply when LLC space is decreased; *sphinx3* that also has frequent accesses to LLC, but its LLC misses mildly increase when LLC space decreases; and *libquantum* has large amount of LLC accesses but barely reuses the data in LLC , so it is highly insensitive to the available LLC space and produces constant evictions.

From a single run of these benchmarks, SEA is able to obtain predictions of the energy that each benchmark would consume running in isolation under any partition of the cache.

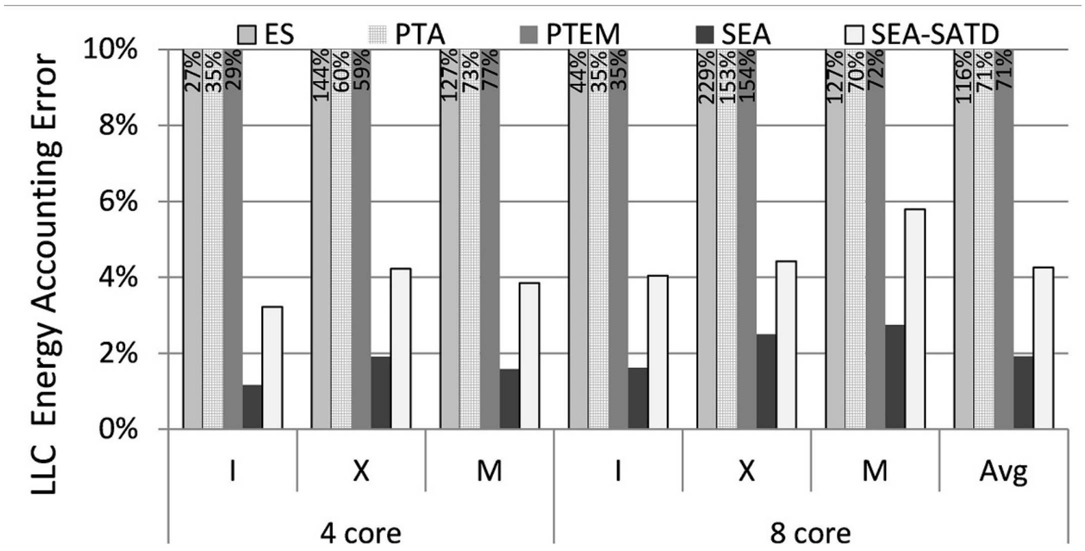


Figure 6.3: LLC energy accounting accuracy, under CMP 4, 8 cores setup, using I, X, M types workloads

Table 6.4: LLC SEA-SATD prediction error standard deviation.

	I	X	M
4 cores	3.5%	4.3%	3.7%
8 cores	4.8%	4.2%	6.1%

We evaluate SEA accuracy by comparing those predictions with the actual consumption each task has under each cache partition setup, see Figure 6.2. We can see that the error of SEA, which is computed as shown in Equation 3.1, is low for all cache partitions with a deviation of up to 4% and an average error always below 1.8%. In general, the prediction inaccuracy of SEA mainly comes from two sources: the estimation of the number of cache accesses by sampling the ATD and performance accounting based on estimating the number of extra cache misses with a given cache size and conflict misses incurred by co-runners. Some benchmarks show higher accuracy for a different cache partition. For instance, *namd* and *libquantum*, whose miss counts barely change with their varied given cache size, obtain highly accurate estimations across all cache sizes. Somewhat, higher variations are observed for those benchmarks that are more sensitive to the space available, such as *astar* and *sphinx3* with no particular trend w.r.t. the number of cache ways. Oscillations for different numbers of cache ways are mainly caused due to the fact that active, maintenance and leakage energy are estimated separately, which may compensate or aggregate estimation errors depending on whether each source of energy consumption is overestimated or underestimated for a given number of cache ways. Still,

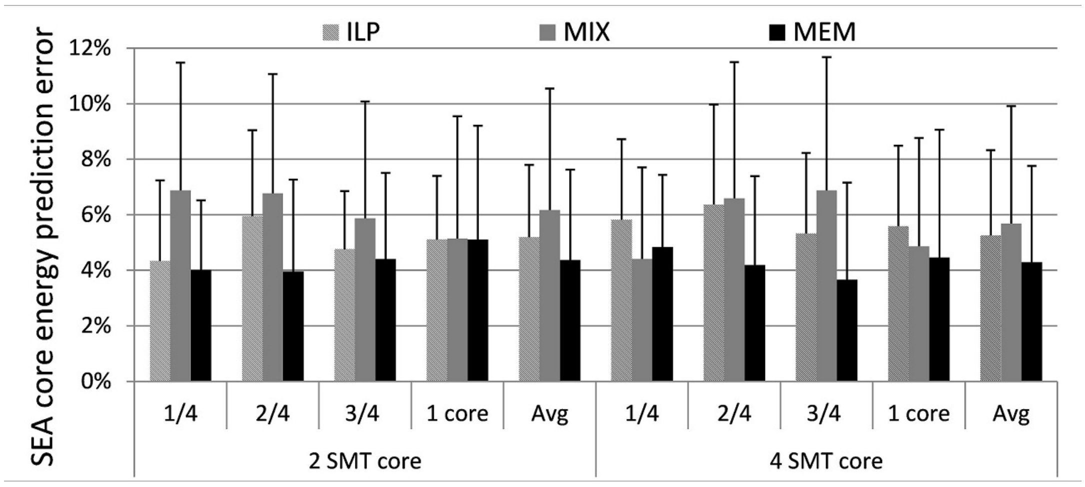


Figure 6.4: SEA_{core} prediction accuracy, under 2, 4 SMT cores setup, using I, X, M types workloads

prediction error is rather low.

For the next experiment we focus on the case in which $fhr = 1/N$, i.e. SEA predicts when each benchmark receives a fair share of the LLC. Figure 6.3 shows the prediction error of the different models under 4-core and 8-core CMP setups: ES, PTA and PTEM. Two versions of SEA are evaluated: with full ATD and with SATD.

As we can observe from the figure, *ES*, *PTA* and *PTEM* fail to accurately predict the energy to account to each task. This is expected as those models do not capture inter-task interferences that impact energy consumed and how energy consumption for a task deviates from the reference. ES, PTA and PTEM have prediction errors above 25% across all workload types and core counts and, on average, all of them produce deviations above 70% on average. On the other hand, SEA has consistent prediction accuracy which has error below 3% across all workload types and core counts, thus showing the excellent improvement of the method. When using SEA-SATD, whose hardware cost is lower, the error only grows to 4%. For the sake of completeness Table 6.4 shows the standard deviation for SEA-SATD. As shown, the variation of the prediction error across the whole set of workloads is moderate. Overall, SEA-SATD is highly accurate and far more better than any state-of-the-art method.

6.5.2 SEA_{core} Accuracy Evaluation

In this section, we evaluate the accuracy of SEA approach in SMT cores. In order to account for the error of the core model, we discount the effect of the shared LLC in this

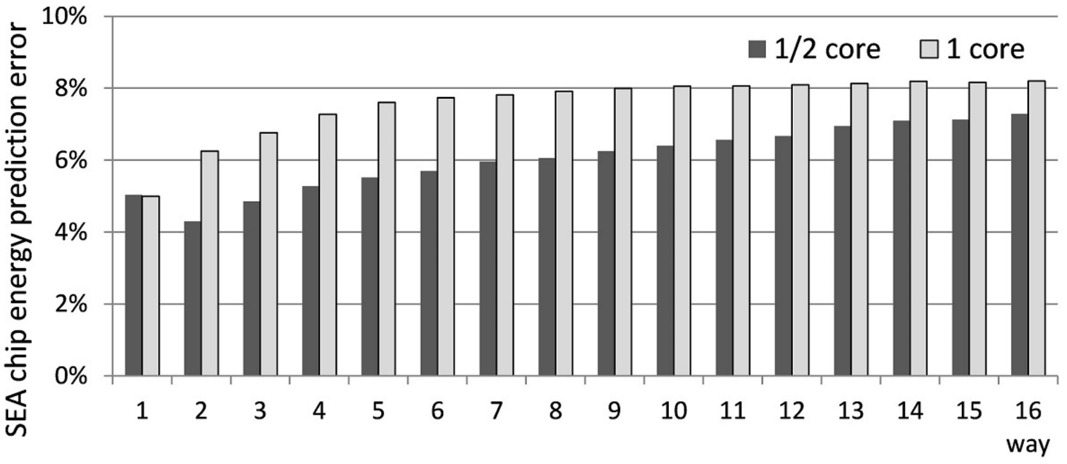


Figure 6.5: SEA_{chip} prediction error for a 4 SMT core setup and 16-way LLC

experiment. In particular, the LLC energy accounted to a given tasks is obtained assuming that the full LLC space has been allocated to it. Therefore, energy variations can only come from the error of the core energy model.

We consider 2- and 4-way SMT core setups. Analogously to the LLC, the ES and PTEM models lack of the flexibility and adequate accuracy to predict the energy one task has with a fraction of the core, so we do not show them in the chart. On average, ES model has over 38% prediction error, while PTEM has over 27% prediction error, when comparing their output with the energy one task should have consumed with the full core.

The prediction error for SEA is shown in Figure 6.4. We observe that, across all setups and types of workloads, SEA has stable prediction accuracy. For X type workloads, the average prediction error is rather higher than the others. We have also shown the standard deviation of SEA prediction error in the figure. While X type workloads have also higher variation than the others, such variation remains rather low for all workloads and setups. Nevertheless, SEA accuracy is still very high.

6.5.3 SEA_{chip} Accuracy Evaluation

In this section, we combine the SEA in LLC and in core. Actually SEA_{chip} is flexible with different combinations of SEA_{LLC} for $\frac{M}{N}$ of LLC and SEA_{core} for $\frac{\hat{M}}{\hat{N}}$ of core.

We analyze all configurations where each task is accounted for half (*1/2 core*) or all (*1 core*) core resources, and for any number of cache ways between 1 and 16. Average off-estimation is shown in Figure 6.5 across the different configurations. The x-axis corresponds to the different number of cache ways (from 1 to 16). It can be seen that error is

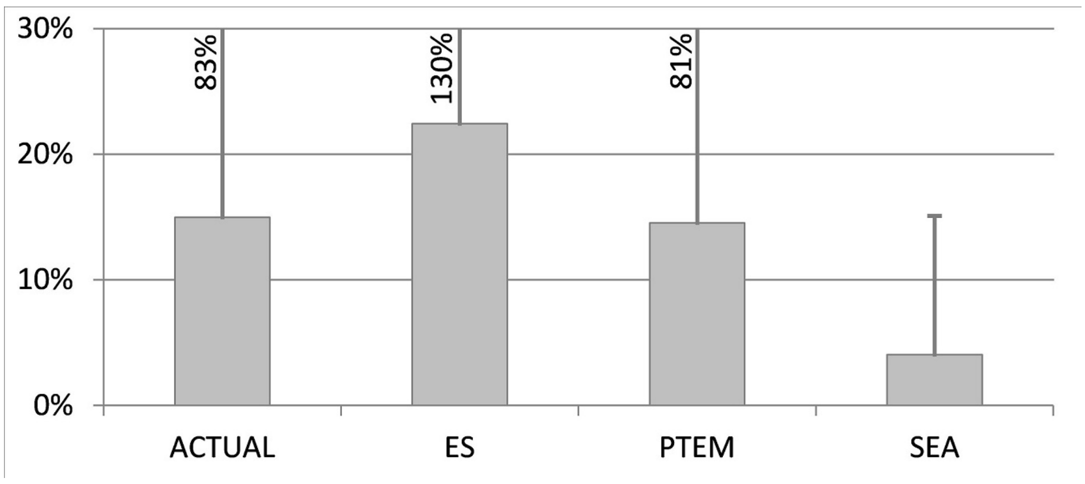


Figure 6.6: The deviation of mispredicted energy account to tasks running in 8-task workloads under 4-core SMT setup and 16-way LLC

in the range 4%-8% on average. In general, higher accuracy is attained when accounting energy for *1/2 core* given that accuracy for the LLC is higher than for the SMT core, and the total energy to be accounted to the core under the *1/2 core* setup is lower. We also observe that higher accuracy is achieved for lower cache ways counts. This occurs because miss rates are normally higher when fewer LLC ways are allocated, and thus, increase the portion of active energy. Although the extra misses lead to more inaccuracies to the execution time prediction, fewer LLC ways contribute low maintenance and leakage power so less impact when compared with the increased but accurately estimated active energy.

Overall, SEA achieves very high accuracy estimating energy consumption under a given fraction of resources despite the fact that it is estimated under workloads where many resources are shared in many different ways.

6.6 Case Study

6.6.1 Energy Accounting Using ES, PTEM and SEA

In order to illustrate the main conceptual differences between ES, PTEM and SEA, in this section we analyze the variation in terms of energy consumed and in terms of misprediction w.r.t. the energy that should be accounted. As for the actual energy, we make use of the ideal per-task energy metering model proposed in Chapter 4, which stands as a oracle version of PTEM that disregards the cost to measure energy. We consider that the per-task energy measured by this model is the best approximation to the actual energy consumed by tasks, thus, labeled as *ACTUAL* in the plot. Since all solutions compared (ES, PTEM and SEA) have negligible energy impact in practice, the actual energy consumed is essentially the same, so we just plot one column for *ACTUAL*. Note that accounting for an homogeneous share of the resources across tasks is the only case where ACTUAL, ES and PTEM can attain some degree of accuracy. In contrast, SEA is able to account energy for arbitrary fractions of the shared resources. Therefore, for comparison purposes here we only consider an homogeneous share of the resources for each task.

In particular we analyze the energy accounted to task T_i running in an SMT core of a 4-core 16-way LLC, when half of the core resources and 2 ways of the LLC are accounted to it. In other words, T_i is accounted for exactly 1/8 of the resources of the processor, as it is able to run up to 8 tasks simultaneously. Figure 6.6 shows the average and maximum energy prediction errors. In particular, we obtain for each benchmark its range of variation (maximum minus minimum energy) w.r.t. to its energy consumption when running alone with 1/8 resources, and then we report in the figure the average and maximum value across benchmarks.

We observe that the actual consumed energy has an average 15% prediction error across benchmarks and the maximum error reaches 83%. When using ES model for energy accounting, we observe that variations are significant. On average prediction error is 22%, while the maximum for one benchmark reaches 130%. This would mean that users would get 22% variations in the bills on average and those variations could reach 130% for the very same task. In the case of using PTEM, results of the actual implementation are very similar to those of the ideal PTEM model. On average the prediction error is around 14% and in some cases it may be as high as 84%. This reflects the fact that many tasks may significantly overuse/underuse the resources w.r.t. a fair share of them. This affects their own energy consumption and co-runners consumption. In contrast, SEA reduces the average error down to 4%, and maximum is 19% for one benchmark. These prediction errors are far lower than those of ES and PTEM and can be hidden from end users to some extent by the fact that the cost per Watt also varies along time. SEA is able to accurately predict the energy consumed with a fair share of the resources with negligible cost, as shown before, and allowing tasks to freely share resources.

In addition, when we account one workload with the energy accounted to $fhr = 1/8$ resources of all its tasks, comparing with its actual energy consumption, we found the actual energy saves on average 7.7% across all workloads because of resources sharing. Thus, on one hand, datacenter operators can leverage the use of SEA to further reduce the actually consumed energy by finding an optimal point to co-locate tasks like we show in Section 6.6.2. On the other hand, SEA can qualitatively applying the energy saving as discount to end users as mutual benefits.

6.6.2 Energy Oriented LLC Allocation Using SEA

In this section we present a case study that shows how to use SEA as a powerful mechanism enabling energy savings. Similar approaches have been proved effective for performance optimizations [80, 81, 107]. Those approaches show that the performance gain could be significant when performance can be accurately accounted. By tracking the tasks running in a workload, SEA accurately estimates the energy consumed by each task under each number of allocated LLC ways, thus enabling efficient LLC space allocation algorithms with no need to run all programs under all configurations. In this section, we use a simplified scenario to show the potential on energy saving if we can choose the most optimal resource allocation scheme for tasks in a multi-benchmark workload regardless of the system throughput and per-task performance. In this case, we assume a CMP architecture with non-shared LLC, in which each task accesses its allocated LLC space exclusively. In this experiment, we have included the energy consumption of the memory. The memory system is simulated using DRAMsim2 [98], which is connected to our processor simulator. The power model in it is obtained from MICRON data sheets [84]. Memory energy accounting is not in place and decisions regarding the most convenient cache partition are performed only based on core and LLC energy accounting. Thus, if memory energy accounting was in place there would be potential for identifying better cache way partitions to further increase the energy saving. Sensible memory energy accounting would need a specific technique, which is part of our future work. Based on the fact that per-task memory energy metering has already been proposed [71] and SMT core and LLC energy accounting has been proved doable on top of energy metering, we do not expect any impediment in devising accurate memory energy accounting techniques.

At first, based on PTEM measurements, we can observe that benchmarks have various energy profiles with different number of allocated LLC ways. For some benchmarks, their consumed energy increases with more LLC ways. This is due to the correspondingly increased LLC power overlaps the reduction on execution time benefit from more LLC space. In contrast, the energy consumption of some benchmarks decreases with more allocated LLC ways. Analogously, this happens because their LLC misses reduce sharply with more cache space allocated, which significantly improves their performance. Also, there are several benchmarks with varying behavior. For those benchmarks, till a given point, allocating more LLC ways pays off because the energy saved due to the reduction

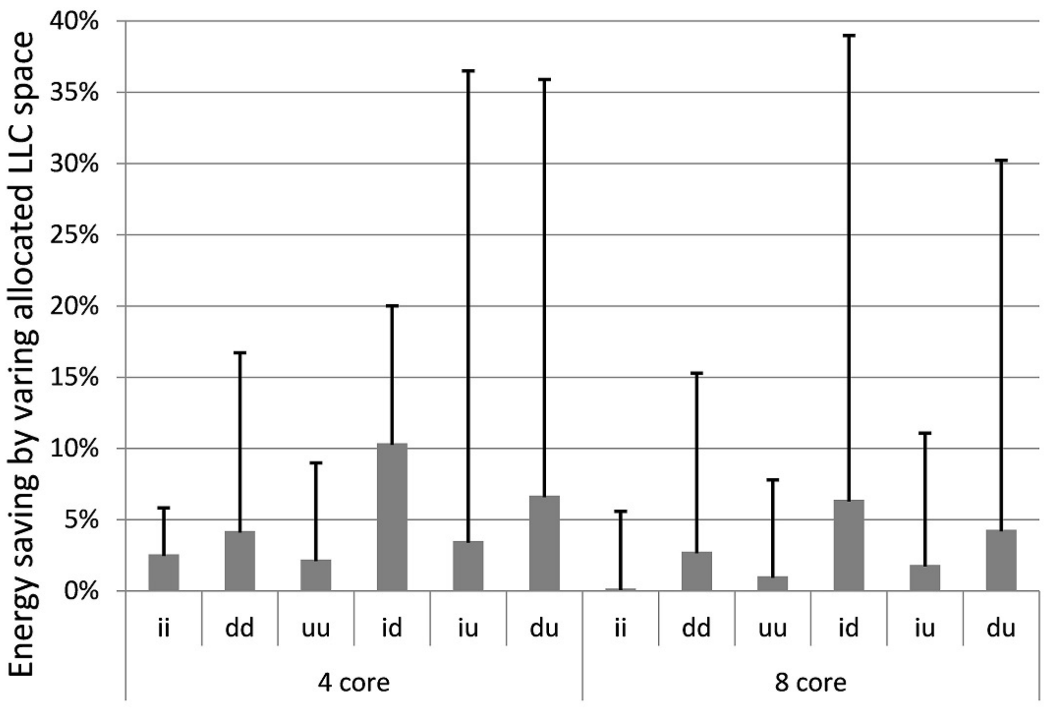


Figure 6.7: Energy saving with varied LLC space allocation, comparing with fair allocation

in misses is higher than the extra energy consumed by those ways. Beyond that point, their LLC misses do not further significantly decrease and then, the energy consumed is increased.

Therefore, in this section we classify benchmarks differently from what we showed in Section 3.4.3, since this helps to better understand the different characteristics across benchmarks. In particular we divide programs into 3 categories: those whose energy increases as LLC space increases (i), those whose energy decreases as space increases (d), and the remaining ones that have a U-shape trend (u). i programs do not make efficient use of the cache space, so increasing LLC space will simply increase their maintenance and leakage energy. They all have minimized energy consumption when only 1 LLC way is allocated. In contrast, d programs exploit LLC space efficiently, so they minimize their energy consumed when they are allocated all LLC ways. Finally, u programs minimize their energy consumption with a number of ways larger than 1 and smaller than the whole cache space.

We compare the energy savings with the best LLC allocation with a fair share allocation where each task gets the same number of cache ways. In Figure 6.7, bars show average energy saving across workloads in a particular category while the lines on top of them show the maximum savings. Workloads are built by combining half of the benchmarks of one type and half of them of another type.

As shown, the lowest average energy savings correspond to the cases where all benchmarks are of type i (ii case) or of type u (uu case). This is expected as i type benchmarks have a near constant active energy consumption, and the optimal maintenance and leakage energy remain roughly constant regardless of how space is split. In the case of uu workloads, the baseline space distribution is already close to the optimal one as each program needs a fraction of cache space somehow in the central part of the distribution. In other cases it is easy to find some benchmarks with different sensitivities to the amount of cache space, so there are workloads with energy savings between 10% and 40%. This results confirm how SEA can be used to enable other energy saving techniques.

6.7 Summary

The advent of CMPs allows running many tasks simultaneously, thus allowing resources to be shared and, generally, optimizing energy efficiency. Unfortunately, the energy consumed by a given task strongly depends on the set of co-runners, which create different inter-task interferences. Therefore, energy consumption of a given task with a given set of inputs can change noticeably across different executions. If energy is used for billing, it is hard to defend charging end users largely different energy costs for the very same service.

This chapter develops the concept of Sensible Energy Accounting (SEA) from theoretical point of view. SEA allows accurate estimation of the energy that would be consumed by a given task if it was running with a given fraction of the resources, despite the fact that the task shares resources in a multi-task workload. SEA, thus, opens the door to stable billing as well as energy optimizations in CMPs. Our results show that SEA provides highly accurate estimations for on-chip resources – as needed for billing – and can be used for scheduling purposes achieving up to 39% energy savings.

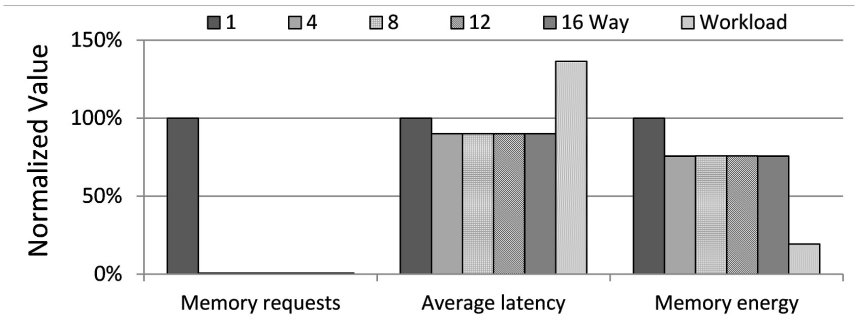
7

Sensible Energy Accounting for the DRAM Memory System

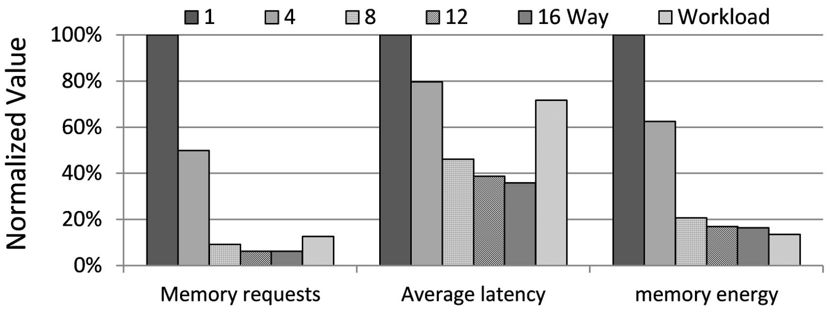
7.1 Introduction

The memory wall still limits performance, so many techniques have been devised to hide the long memory latency by allowing multiple memory requests access the memory system in parallel, such as the non-blocking cache, out-of-order instruction issue, speculative execution, etc. Furthermore, the pervasive use of the multi-core and many-core design paradigms puts more pressure on the memory system because multiple tasks can send in parallel their memory requests. In this scenario, the execution of one task can be severely interfered by other co-running tasks due to memory access contention. Thus, in order to efficiently use memory resources, modern DRAM controllers implement complicated scheduling policies to issue the memory requests from the processor to the memory system. However, while the overall performance has been generally improved, the energy usage of each task in memory has not been deeply analyzed. This fact is highly relevant given that in modern computing systems the power of the memory system is already as significant as the processor socket [14].

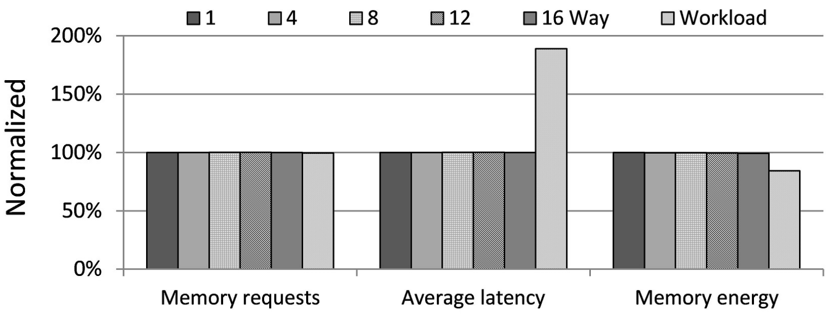
In Chapter 5, we have introduced our techniques to meter the memory energy that is actually consumed by each task during their execution in a workload running in a multi-core system. However, that mechanism cannot tackle the issue of sensibly accounting for a task the energy it would consume in the DRAM DDR2/3 memory system when it has an arbitrary fraction of processor resources to use alone, as formalized in Section 1.1.2.



(a) 453.povray



(b) 473.astar



(c) 433.milc

Figure 7.1: The number of memory requests, the average latency of these requests and the incurred energy of SPEC CPU 2006 benchmarks *453.povray*, *473.astar* and *433.milc*, when they run alone with different LLC ways and in the workload, are shown respectively in 7.1a, 7.1b and 7.1c

This is a difficult challenge since heterogeneous applications nowadays run simultaneously in the same computing system, and their consumed energy varies when they run with different co-runners. In order to fairly account for their energy usage, it is needed to have a consistent energy profile. The outcome of this work will be beneficial for several applications, such as datacenter billing, scheduling policy, resource partitioning, tasks co-location, etc.

To elaborate on this need, we ran three SPEC CPU 2006 benchmarks *453.povray*, *473.astar* and *433.milc*, alone in a single core system, connected to 16-way 4MB LLC and 8GB DRAM memory system, the detailed setup can be found in Section 3.2. Each benchmark has been run for 16 times, each time with a different number of LLC cache ways (from 1 to 16), while the rest of resources remain the same. During the execution, we monitor the number of memory requests generated, the average latency of the memory requests from their generation till their completion, and the memory energy consumed by the task. In addition, we ran each of these three benchmarks individually in 4 different 4-task workloads that have been randomly generated. Each workload has other three benchmarks, running on a 4-core multicore system with the same but shared 16-way LLC and memory setup, during which, we obtain the same runtime statistics. In particular, the energy of each task is derived through the ideal model we have introduced in 5, known as the *actual* energy. The results are shown in Figure 7.1 for the 3 benchmarks respectively. For the sake of clarity, we consider the case when only 1-way of LLC is used as the baseline and all results are normalized w.r.t. this case.

We observe that the 3 benchmarks exhibit heterogeneous behavior. In 7.1a, we can see for *453.povray*, which is an ILP bound benchmark, its generated memory requests are few and stable as long as it has at least 4 LLC ways. With just 1 way its data do not fit in the LLC and the number of memory requests is much higher. On the other hand, the average latency of each request and the energy consumed only decrease slightly. For 4 or more ways the number of memory requests decreases down to 1% of those with 1 way, the average latency is 90%, and the energy around 75%. Conversely, when this benchmark runs with other tasks in the workloads, the average request latency increases to 140% due to the contention in accessing memory resources, but the energy drops to 20% because the (high) background energy is shared with other benchmarks. In the case of *473.astar* in Figure 7.1b, its number of memory requests is more sensitive to different number of LLC ways allocated. When it has 4 ways, its generated memory requests decrease to 50% compared to the 1 way baseline. After further increase to 8 ways, the number drops to around 8%. Its average memory latency and energy variation follow the same but decreasing less. When this benchmark runs in the workloads, although it only generates 10% of the memory requests, the average latency stays over 75%, but the energy is around 15%. Benchmark *433.milc* has a completely different behavior since its generated memory requests remains roughly constant across all scenarios. Due to contention, its average memory latency rises to almost 180% in the workloads. However, its metered energy is around 90% that of the cases when it uses the memory system alone.

From these results, we can conclude that neither the number of accesses nor access latency directly reflect the energy consumption. In order to sensibly account the energy a task would have consumed in a particular configuration, some specific information needs to be correlated to deliver accurate estimates.

In this chapter, we introduce our approach for SEA in the memory system. Our proposed techniques accurately estimate, during runtime, the energy consumption each task would have in the DRAM DDR2/3 memory system when they run alone with an arbitrary fraction of resources on the processor, with efficient implementations. The energy is accounted sensibly to tasks with their predicted behavior depending on the particular fraction of the resources allocated. This is done by analyzing their runtime behavior when they run within an arbitrary workload. Overall, the contributions of this work are as follows:

- We propose an ideal sensible energy accounting model for DRAM memories, based on the assumption that the memory behavior a task has when it run with a particular fraction of resources in isolation can be fully known during the runtime in a workload. To the best of our knowledge, it is the first reference model against which energy accounting mechanisms in the DRAM memories can be compared to.

- We propose techniques to predict for the task the activities, the time invoking the memory system and execution time when it runs with a fraction of the processor resources in isolation. Based on which, we devise SADEA, a practical implementation of the ideal model.

- We compare SADEA with existing energy measurement techniques, such as ES, PTA and `DReAM`. We show that SADEA extends the concept of energy accounting in computing systems, in which SADEA provides consistent and accurate estimates for the fraction of processor resources allocated.

The rest of this chapter is organized as follows. Section 7.2 provides background on memory controller scheduling policies and memory interference delay analysis. Section 7.3 presents our approach to perform ideal SEA in DRAM memory systems. SADEA, our efficient hardware implementation of the ideal model, is described in Section 7.4. SADEA accuracy is evaluated in Section 7.5. Section 7.6 draws the main conclusions of this chapter.

7.2 Background

7.2.1 Performance Accounting

As introduced in Section 6.2, SEA comprises two main blocks: PTEM techniques and performance accounting techniques. In this respect, in order to build the SEA for the memory system, we need a PTEM for the DRAM memory system, such as `DReAM` – see Chapter 5 – and a mechanism for performance accounting in the memory system.

Due to the significant amount of techniques used to hide memory latency, the latency of memory requests will impact the execution of the task in a non-obvious way. For example, let us assume a scenario where a task has many memory requests during a particular time interval when running in isolation. If it runs as part of a workload and due to LLC interferences it has an extra LLC miss, such extra memory request may be overlapped with other requests, thus leading to null or negligible performance impact. However, if the task performs roughly no requests during such time interval, the extra request is unlikely to be overlapped and it may easily impact performance.

Therefore, in order to sensibly account the memory energy, we need to accurately account performance based on the varying memory behavior. However, it is difficult to do it in today's sophisticated memory controllers and DRAM devices.

7.2.2 DRAM Memory Controller Scheduling Policies

In a modern DRAM memory system, operations strictly follow the JEDEC DDR standard [52]. In the DRAM memory organization that has been detailed in Section 5.2.2, memory requests from the chip are dispatched to the memory system by the memory controller. During this process, several specifications have been considered, such as the I/O timing parameters of the DRAM devices, address mapping scheme, row-buffer management policies, etc. Therefore, the memory controller needs the scheduler issuing the internal commands according to their time constraints, so that each operation can be performed correctly. In addition, different policies can be used to optimize the performance and energy efficiency of the memory system. We have inherited the memory controller model from DRAMsim2, which uses a typical scheduling policy, known as first-ready first-come-first-serve (FR-FCFS), which prioritizes the ready commands over the non-ready commands, the old commands over the newly arrived commands, and the column access commands over the row access commands when open-page buffer management policy is used. Note that the ready commands can be sent to the DRAM devices immediately with no constraint. By applying such policy, the commands issuing order will not follow their arriving order exactly.

7.2.3 Memory Interference Delay Analysis

Recently, there has been an increasing interest in analyzing the delay caused by memory interferences, mainly in real-time system domain [62,118]. These works focus on estimating the Worst Case Execution Time (WCET) of a task, which is an upper-bound of the memory interferences that come from the other co-running tasks. For this purpose, based on the memory behavior a task has during a single run, authors create high synthetic interference by generating memory requests from the other cores, and estimate the WCET each task has under such extreme conditions. In that respect, authors have made several simplifications, such as: any increase in memory latency is additive to the task's execution time; all the commands generated from a memory request suffer contention due to interference disregarding the bank-level accessing parallelism in DRAM devices.

Conversely, in our work we cannot follow these principles. Instead, we monitor the memory behavior of a task during runtime when it co-runs with many other arbitrary tasks, based on which we analyze the memory behavior each task should have when it runs alone with an arbitrary fraction of the processor resources. As far as we know of, such a model does not exist.

7.3 Ideal SEA for DRAM Memory System

In this section, first of all, based on the SEA definition we have in Section 1.1.2, we introduce how SEA can be ideally applied in a DRAM DDR2/3 memory system. Although the core resource partition cause variations in terms of LLC accesses, and in turn LLC misses, its influence does not fall outside the scope of LLC space variation. Thus, in the main line of this work we ignore the different core resources allocation, since memory behavior mainly depends on the LLC space allocated. Nevertheless, we comment how to extend the proposal to cover core resources variation in Section 7.4.1.

Our target is to account to a given task T_i, when running in a workload in a multicore, the energy it would consume with a given (arbitrary) fraction of the LLC space. Given that the LLC has N ways, the possible number of ways of the LLC that T_i could use ranges from $1 \ldots N$, and we use n to refer to the particular number of ways in the LLC used by T_i for its accounting.

Following the classifications we have introduced in Section 5.2.2, the energy consumption in the DRAM memory system is broken down into three components: active, refresh and background. We show how different components of the energy should be ideally accounted to a task.

Active Energy

Active energy in the DRAM memory system corresponds to the energy spent to perform task activities, such as the energy incurred by the commands that relate to the memory request sent by a task. Therefore, in order to account the active energy in the memory to a task for a particular LLC allocation, n ways, it is needed to account the activities it would have incurred under that condition.

When the bank buffer management chooses close-page policy, this is relatively easy because every request will have the following activities: first, a row activate command (ACT) is sent from the memory controller to load a specific row of data in the data arrays to the row-buffer sense amplifiers; upon its completion, a column read/write (READ/WRITE) command arrives to read/write the data from/to the row-buffer; finally, a precharge (PRE) command is triggered to restore the data from the row-buffer to the arrays on the row where they were stored. In this case, the activities performed by a task are directly correlated to its generated memory requests.

For the open-page policy, the activities cannot be directly mapped to the memory requests, since ACT and PRE commands are not always needed for memory requests in this case. When this policy applies, the row-buffer will remain open after the former READ/WRITE command is done, without sending a PRE command to restore the data. This occurs because, if there are pending requests on the row that is already in the row-buffer, the memory controller can send them immediately. Therefore, the response speed of the row-buffer hit requests is increased, since they only have to pay the READ/WRITE latency, thus avoiding the delay due to the PRE command of the former request and the ACT command of its own request.

The selection of the row-buffer management policy is beyond the scope of this thesis. For both policies, the active energy should account following the same principle: to account T_i the useful activities it would incur with its own generated memory request. Thereby, in this ideal model, we assume that the number of internal commands from each task is known, so that we can directly account the active energy based on the command counts. Thus, given a task T_i that uses n ways in the LLC, the active energy accounted to it is calculated as follows:

$$
\begin{aligned}
EA_n^{dyn}(T_i) = {} & Num_n^{ACT}(T_i) \times E^{ACT} + Num_n^{READ}(T_i) \times E^{READ} \\
& + Num_n^{WRITE}(T_i) \times E^{WRITE} + Num_n^{PRE}(T_i) \times E^{PRE}
\end{aligned}
\tag{7.1}
$$

where $Num_n^{ACT}(T_i)$, $Num_n^{READ}(T_i)$, $Num_n^{WRITE}(T_i)$ and $Num_n^{PRE}(T_i)$ stand for the number of commands of each type that belong to task T_i when it uses n LLC ways. E^{ACT}, E^{READ}, E^{WRITE} and E^{PRE} stand for the energy consumed by each command.

Background Energy

Background energy includes the maintenance and leakage energy, which correspond to the energy consumed due to *useless* activities not triggered by the programs being run and the energy wasted due to the imperfection of the technology used to implement the circuit that is detailed in Section 2.1. Note that the background power of the memory system has different levels corresponding to different states of the DRAM device: power down (P), standby (S) and active (A). The power in P state is the lowest across all states and it is incurred when the memory system clock is disabled. After enabling clocking, the DRAM device enters S state, which largely rises the background power but can quickly respond to the requests. When executing the ACT command, the background power rises to A state. Such A state is also needed to perform READ/WRITE commands. After the PRE command precharging the open row in the DRAM device, the background power returns to S state. Therefore, the background energy that should be accounted to task T_i when it uses n ways in LLC, is determined by the time DRAM devices spend in each state. Given that in an ideal model the time information can be known, we can calculate the background energy to account to task T_i as follows:

$$
EA_n^{BG}(T_i) = T_n^A(T_i) \times P^A + T_n^S(T_i) \times P^S + T_n^P(T_i) \times P^P \tag{7.2}
$$

where $T_n^A(T_i)$, $T_n^S(T_i)$ and $T_n^P(T_i)$ stand for the time task T_i induced the DRAM device to remain in A, S and P states respectively when it has been allocated n ways in LLC. P^A, P^S and P^P stand for the background power under A, S and P states respectively.

Refresh Energy

Refresh energy corresponds to the energy consumed to refresh periodically all memory contents, which is consistent in a time interval that is adequately long according to the JEDEC standard [52], for example, 40 μs in our used configuration. Therefore, accounting refresh energy to one task based on its execution time with n ways of the LLC, is done as follows:

$$EA_n^{REF}(T_i) = T_n^{Exe}(T_i) \times P^{REF} \tag{7.3}$$

where $T_n^{Exe}(T_i)$ stands for the execution time of T_i with n ways LLC and P^{REF} stands for the refresh power.

7.4 SADEA, an Implementable Approach of SEA

Due to the increasing core count and the sophisticated modern memory controller design, tracking all the profiles that are needed by the ideal SEA is virtually impossible. Therefore, we propose SADEA, a simple to implement yet accurate model that follows the same methodology of the ideal model.

T_i has different behaviors in the DRAM memory system when it runs alone with a fixed number of LLC ways, and when it runs as part of a workload with several co-runners. This occurs because in the workload: 1) other tasks may evict some cache lines of T_i, which will probably cause T_i suffering from extra cache misses, known as inter-task misses; 2) in a particular n way configuration where $n < N$, T_i may suffer some cache misses due to the limited capacity, but those accesses may be hits when running as part of a workload; 3) the memory controller receives more memory requests due to co-runners, and they have to be scheduled, thus causing potentially significant delays on T_i memory requests.

Active Energy

In order to account the active energy, the main difficulty lies in estimating the number of memory requests for task T_i running in isolation with n LLC ways. Therefore, accurately estimating the inter-task misses and the capacity misses due to using n ways in LLC holds the key.

To this end, we use a similar technique to those used by SEA in the LLC as detailed in Section 6.4.3. In SEA we have used the Auxiliary Tag Directory (ATD) proposed by Qureshi and Patt [96], which focuses on a least recently used (LRU) replacement policy, and it is used to estimate the LLC accesses for T_i when n ways of LLC are allocated. The ATD is used to keep a local copy of the tag directory for each task, which keeps track of what would be the LLC contents of a task with any arbitrary number of LLC ways in the case of LRU. In this way, if the LLC implements LRU, one can predict whether an access would hit in the LLC for any number of cache ways n lower or equal to the total number of LLC ways (N) as explained in Section 6.4.3. Also, as explained before, we use the Sampled ATD (SATD) instead of the ATD to keep overheads low. And the SATD can also be used for pseudo-LRU caches with negligible impact on accuracy, so the same reasoning applies to SADEA.

Given that LLC misses are directly mapped to memory requests, we can rely on the access count estimation derived with the SATD to estimate the active energy in the memory system. In close-page policy, the number of memory requests can be mapped exactly into the number of commands in the memory system. For open-page policy, instead, the estimate needs to correlate with the timing and data locality information of the memory requests. However, in existing memory controller designs, the implementation of open-page policy is different from the theory. In practice, since the background power in state A is very high (comparable with the commands power), the banks are only allowed to stay open for a short time interval to lower power consumption. Therefore, the hit-in-page rate reduces significantly w.r.t. that expected hit rate in theory. In our case we have found that the difference between open-page and close-page is negligible in practice, so we use the close-page model in both cases:

$$EA_n^{dyn}(T_i) = (E^{ACT} + E^{PRE} + \frac{E^{READ} + E^{WRITE}}{2}) \times Num_n^{MemAcc}(T_i) \tag{7.4}$$

where $Num_n^{MemAcc}(T_i)$ stands for the estimated LLC misses of task T_i under LLC fraction n derived with the SATD. E^{ACT}, E^{PRE}, E^{READ} and E^{WRITE} represent the energy needed by each command that is provided by the hardware manufacturer, such as in [84]. In our case we average the energy of read and write operations since it is typically very similar and allows us not having to track each type of event individually. If different types of accesses have different energy consumptions (e.g., read and write operations), then different counters need to be kept for each operation type per task so that each access updates the counter corresponding to its type. With current DRAM technology, read and write operation in general have less than 10% difference [84].

Background Energy

In order to sensibly account the background energy, the number of memory requests and their impact on the execution time and the time invoking the memory system need to be correlated. Following Chapter 5, we split the background energy into 2 parts: 1) the energy consumed under P state would always be consumed; 2) the extra background energy is only consumed when the power state of DRAM devices is raised to S or A states.

Table 7.1: A synthetic case of two tasks T_0 and T_i accessing 2 banks in parallel

Requests(T_0)	R_0	R_1	R_2	R_3									
Requests(T_1)	R_0	R_1	R_2	R_3									
$Bank_0$	$R_0(T_0)$			$R_1(T_0)$			$R_2(T_0)$			$R_3(T_0)$			...
$Bank_1$	$R_0(T_1)$			$R_1(T_1)$			$R_2(T_1)$			$R_3(T_1)$			...
Cycles	20	20	20	20	20	20	20	20	20	20	20	20	...

Table 7.2: A synthetic case of two tasks T_2 and T_3 accessing 2 banks in an intervealed fashion

Requests(T_2)	R_0	R_1	R_2	R_3									
Requests(T_3)	R_0	R_1	R_2	R_3									
$Bank_0$	$R_0(T_2)$			$R_1(T_3)$			$R_2(T_2)$			$R_3(T_3)$			...
$Bank_1$	$R_0(T_3)$			$R_1(T_2)$			$R_2(T_3)$			$R_3(T_2)$			...
Cycles	20	20	20	20	20	20	20	20	20	20	20	20	...

Table 7.3: Task T_0 with multiple read requests accessing 1 bank

Requests(T_0)	R_0	R_1	R_2	R_3									
$Bank_0$	$R_0(T_0)$			$R_1(T_0)$			$R_2(T_0)$			$R_3(T_0)$			...
Cycles	20	20	20	20	20	20	20	20	20	20	20	20	...

To open up the analysis on such effects, we use the following three metrics: 1) given a memory request that experiences no contention when it is served, its latency will be a fixed value that is specified by the JEDEC standard [52] and hardware vendor implementations, namely *DL* (Default Latency). 2) The second metric we use corresponds to the count of all cycles that a task spends with at least one memory request inflight in the memory system, which we name as *MC* (Memory Cycles). *MC* represents the time a task induces the memory system to be in a high power-consuming state. 3) For each memory request, we monitor from the cycle it is dispatched to the memory controller, till the cycle its data is returned to the LLC. This metric represents the whole penalty this particular LLC miss suffers, denoted as *LLCMP*. During the period of the *LLCMP*, the execution of the task in the processor pipeline may stall for a certain time till the miss is handled.

We use some synthetic examples to illustrate how we account background energy based on those metrics. We assume that tasks $T_0 \dots T_3$ all have 4 memory request bursts. In Table 7.1, only T_0 and T_1 are active, and each one accesses a different bank. Conversely, in Table 7.2, only T_2 and T_3 are active and access two banks in an interleaved manner. In both cases, regardless of the data and command bus conflicts, although the tasks have very different bank access patterns, their timing behavior is very similar. All tasks spend 240 cycles invoking the memory system as *MC*. Their *LLCMP*s is also the same, $\frac{60+120+180+240}{4} = 150$ cycles. Thus, their performance and metered energy are identical. However, they behave differently when they access the full memory system alone.

Table 7.3 shows the case when T_0 runs with the full memory system alone. All its

Table 7.4: Task T_2 with multiple read requests accessing 2 banks

Requests(T_2)	R_0	R_1	R_2	R_3				
$Bank_0$	R_0			R_2			...	
$Bank_1$	R_1			R_3			...	
Cycles	20	20	20	20	20	20	20	20

memory requests access sequentially bank 0, which leads to the same performance as when it runs with T_1. Conversely, in Table 7.4, T_2 has its memory requests scheduled accessing 2 banks in parallel, and its MC reduces down to 120 cycles. Therefore, in order to estimate the extra background energy in the memory, which is determined by MC, an efficient approach is to detect the Bank Level Parallelism (BLP) of a task's memory requests. This basically translates into measuring how long a task has inflight requests in memory. For this purpose, we setup a counter for each task in the memory controller. It increments when a command belonging to task T_i is sent to the memory system, and decrements when one command completes. This counter is sampled every 1,000 cycles, so that we can obtain the average BLP_i for T_i. We use such BLP_i estimation to compute the MC with following formula:

$$MC_i = \frac{Num_n^{MemAcc}(T_i)}{BLP_i} \times \frac{(DL_{READ} + DL_{WRITE})}{2} \tag{7.5}$$

In essence, we multiply the number of accesses by the average minimum accessing latency, which would give the minimum latency experienced if all accesses occurred sequentially. Then we divide the value obtained by the bank level parallelism to account for the fact that accesses may happen in parallel. As a result, we estimate the background energy of a task in A and S states as follows:

$$EA_n^{ExBG}(T_i) = (P^A \times (L^{ACT} + L^{READ/WRITE}) + P^S \times L^{PRE}) \times MC_i \tag{7.6}$$

where L^{ACT}, $L^{READ/WRITE}$ and L^{PRE} stand for the latency of each command.

Note that, although a task may generate different amount of memory requests when it has different ways of the LLC allocated, the memory space it is allocated will be the same, thus BLP_i is still an accurate estimate of the range of banks T_i can access in the memory system.

To carry out the estimation on the P state background energy, we need to estimate T_i execution time when it uses n ways of the LLC. For that purpose, we rely on the performance accounting techniques (ITCA) proposed by Luque et al. [77]. ITCA uses the SATD to infer the number of LLC hits and misses that the task would experience when running alone with all LLC cache ways. For an access that hits in the SATD but misses in the LLC, ITCA defines it as an inter-task miss. Such miss will be a hit if T_i has the full LLC. Authors use specific logic to estimate the miss penalties due to those misses to obtain the cycles that need to be truly accounted to a task. For that purpose, the events considered include rename register stalls, reorder buffer full, etc. Analogously, for accesses that miss in the SATD, but hit in the LLC, the same logic is used to add the corresponding miss penalties to the task. In our work, we use this technique to calculate the execution time T_i has when it runs with n ways of LLC alone in the system, with several extensions. For one access to the LLC, we predict if it is a hit or a miss when n ways of the LLC are used. For either case, we account the cycles caused by the hit or miss following the methodology used in ITCA. However, ITCA uses a fixed memory latency to estimate the miss penalty. Instead, we rely on both the $LLCMP$ estimation when T_i runs with n ways of the LLC alone and the $LLCMP$ T_i has in the workload, and also the BLP_i. Note that the average $LLCMP$ in Table 7.3 and 7.4, are 150 and 90 cycles. Thus, we derive the formula to calculate it as follows:

$$LLCMP^{ISO} = LLCMP_i^{WL} - \frac{(DL_{READ} + DL_{WRITE})}{2} \times (BLP_i - 1) \tag{7.7}$$

where $LLCMP^{WL}$ stands for the $LLCMP$ task T_i has when it runs in a workload. Therefore, the execution time of T_i in isolation with n ways of the LLC, $ExeTime_n(T_i)$, obtained with our adapted version of ITCA, increases the estimated execution time in isolation using $LLCMP^{ISO}$ latency to account for the cost of the spatial misses, and decreases the estimation using $LLCMP_i^{WL}$ to account for the cost of the inter-task misses in the workload.

For the power-down state background energy, which is proportional to the execution time T_i would have when it runs with n ways of LLC alone, thus, we calculate it with the execution time estimation $ExeTime_n(T_i)$ made from our extended ITCA mechanism using following formula:

$$EA_n^{PBG}(T_i) = P^P \times ExeTime_n(T_i) \tag{7.8}$$

Refresh Energy

For refresh energy, since it is directly correlated with the execution time and refresh power, we calculate it in the same way as in Equation 7.3, but with the estimated execution time $ExeTime_n(T_i)$.

7.4.1 Putting it All Together

Integrating SEA_{chip} presented in Chapter 6 and SADEA is mostly straightforward. In general, there is no conflict from the mechanisms of SEA_{chip} and SADEA since both approaches rely on the same input: the fraction of the processor resources that need to be accounted. In these works, we focus on the fractions of two resources to perform SEA: the core and the LLC.

Both on chip and memory SEA relies on the fraction of LLC allocated to the task under analysis. Thus, both approaches require the same hardware support to account for the execution time, LLC accesses and LLC misses when one task has n ways of the LLC in isolation. Thus, they can be integrated seamlessly. In the context of single-threaded cores, no further integration is needed.

In contrary, SEA_{core} uses MIBTA [77] to sample one task at a time per core periodically if cores are multi-threaded. During MIBTA phases, the information collected related to memory accounting with SADEA needs to be aware of MIBTA behavior. In principle, we inherit the methodology that is used to integrate SEA_{LLC} and SEA_{core} as introduced in Section 6.4.5: during the running of a workload in a multi-core multi-threaded system, the tasks on each core are concurrently sampled in separate MIBTA phases (one in each core invoking at the same frequency) in a round-robin process. MIBTA phases, as explained before, allow estimating accurately L1 cache behavior, and so LLC access and miss frequency of each task. The resulting LLC access and miss frequency obtained during the corresponding MIBTA phase is, therefore, assumed constant until the next MIBTA phase. During MIBTA phases, the LLC is still shared across cores, so SEA_{LLC} and SADEA need to remain working since the interactions in LLC exist across cores.

SADEA hardware support and overhead. SADEA builds upon the SATD that is already used in SEA_{chip}. Details on the overheads incurred have already been introduced in Section 6.4.5 (around 0.7% area overhead for the LLC [77, 78, 96]). In addition to that, few registers and little logic are needed by DReAM and ITCA with negligible area overhead also due to SEA_{chip}. Additionally, for SADEA itself, a 6-bits counter is needed for each task to record the number of their inflight memory requests, as well as a register to sample access counts every 1,000 processor cycles. The estimated energy is stored in a register for each task, called Memory Energy Accounting Register (MEAR). Thus, the incurred area and energy overheads are little.

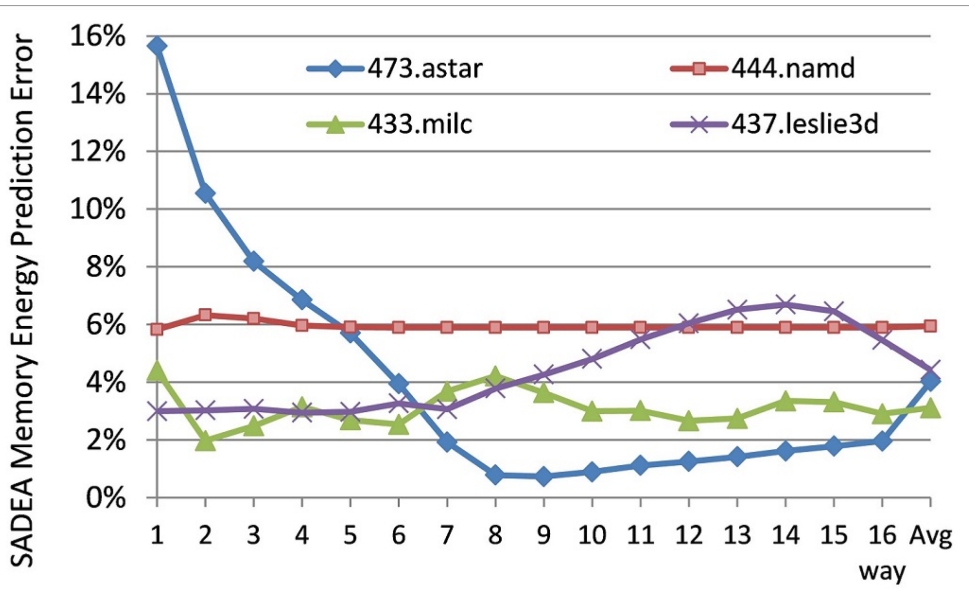


Figure 7.2: Prediction error to account DRAM memory energy to benchmarks *473.astar*, *444.namd*, *433.milc* and *437.leslie3d* in a workload running in 4-core CMP system.

7.5 Evaluation

In this section we assess the SADEA prediction accuracy. We also compare SADEA with other intuitive methods that could be used to account DRAM energy consumption, such as *ES* and *PTA* as introduced in section 2.4, and DReAM in Chapter 5. The experimental setup is introduced in section 3.4. The benchmark suite and workload generation strategy are introduced in section 3.4.3.

Metrics . To evaluate the prediction accuracy of SADEA, we use as a reference the actual memory energy consumption of a benchmark when it runs alone with the corresponding processor resource fraction. For instance, if we aim to estimate the memory energy of a benchmark when it has only half of the LLC ways, the reference is a single-core processor setup with half of the cache ways in the LLC where the benchmark runs alone. Hence, in each experiment, we measure the *prediction error* of each model with respect to the actual energy consumed when one task runs with the specified fraction ($\frac{M}{N}$) of the resources alone, which is computed as in Equation 3.1:

SADEA prediction accuracy in a particular 4-task workload

Figure 7.2 shows the results in terms of prediction error for SADEA for a 4-task workload running on a 4-core CMP architecture, including the following SPEC CPU 2006 bench-

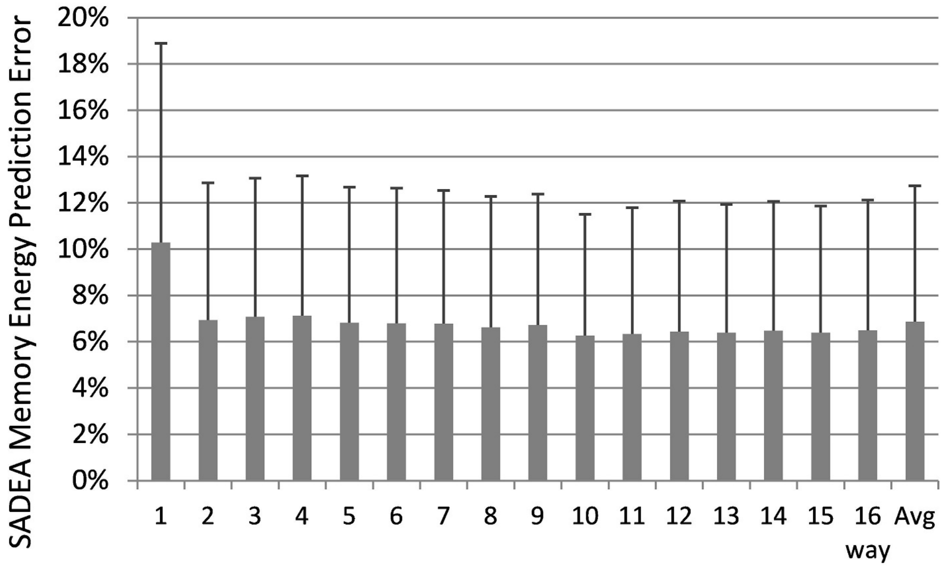


Figure 7.3: Prediction error to account DRAM memory energy to benchmarks for different LLC ways allocated running in 4-core CMP system.

marks: *astar*, *namd*, *milc* and *leslie3d*. The prediction error is mainly generated from (1) the prediction of the activities, (2) the execution time and (3) the time a task uses the memory system. Thus, as benchmark *astar* is sensitive to the LLC space, the number of its memory requests varies significantly with different LLC ways allocated. In particular, as the number of LLC ways allocated decreases, the prediction error increases due to the pathological combination of the 3 sources of error indicated before. The prediction error can be up to 15.7% since its memory behavior is hard to predict with the discrepancy between the actual number of memory requests and the number of accounted ones. The prediction is, instead, very accurate for the cases when it has more than 8 ways to use, since its behavior in the workload is not that different to the behavior that needs to be accounted for. Still, the average prediction error for *astar* is low (4.1%).

Conversely, the number of memory requests barely changes with different LLC ways allocated for *namd*. Thus, prediction accuracy is stable across all configurations. Benchmark *milc* has highly frequent memory request in all scenarios, so its prediction accuracy depends on the interferences it suffers from its co-runners and the speed at which memory requests are dispatched when it uses different number of LLC ways. On average, the prediction error is low (3.2%). For *leslie3d* with moderate number of memory request and moderate variation across different LLC way allocations, the prediction error is within 3-7%, and on average is 4.2%.

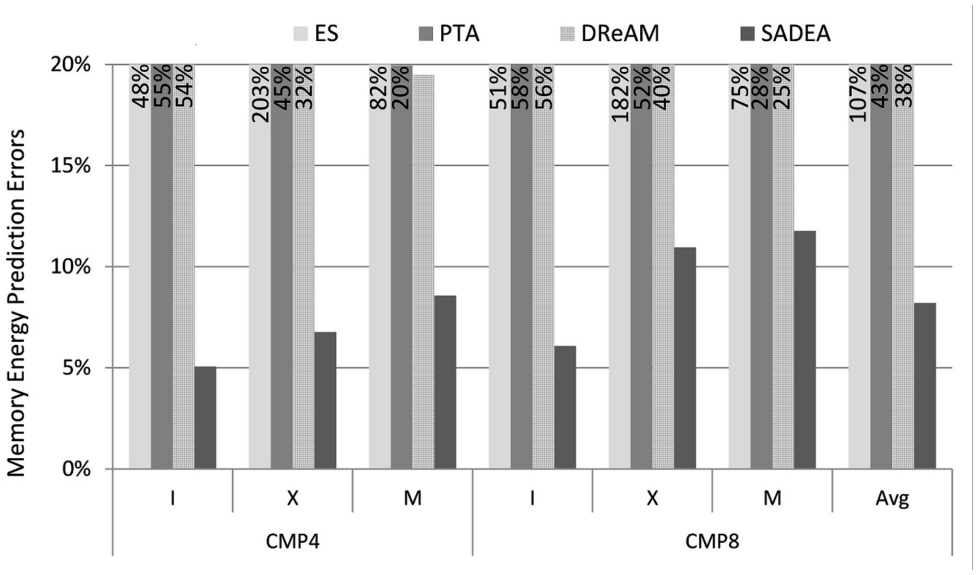


Figure 7.4: Prediction error to account DRAM memory energy for workloads running in a 4-core CMP system using models: ES, PTA, `DReAM` and SADEA.

SADEA evaluation in CMP systems

Next, we evaluate SADEA in a 4-core CMP architecture, with 24 randomly composed workloads using benchmarks of different LLC miss frequency level as classified in Table 3.8. We can observe in Figure 7.3 that SADEA delivers stable prediction across all 1-16 ways of the LLC. The average prediction error across all benchmarks is relatively low, generally under 7%, with their standard deviation under 13%, except for the case of 1-way. The reason that the deviation for 1-way is higher than others is because many benchmarks experience a drastically different number of LLC misses when only 1 LLC way is allocated. This huge variation in the number of LLC misses translates into a significantly different memory energy profile that is, in turn, very hard to predict. Nevertheless, the overall prediction is sufficiently accurate, 6.5% on average. Still, we believe there is room for improvement in the future.

Then, we compare SADEA in 4-core and 8-core scenarios, with *ES*, *PTA* and `DReAM`. The results are shown in Figure 7.4, where workloads are categorized into I, X and M, as described in 3.4.3. Note that in this figure, the outcome of *ES*, *PTA* and `DReAM` are compared with the cases where a task runs in isolation with a given fair share of the LLC, and this is the only case they can be compared with. For example, for N tasks running in a M ways LLC, each task is given $\frac{M}{N}$ ways of cache. We can observe from the figure that *ES*, *PTA* and `DReAM` fail for the purpose to sensibly account the memory energy to a task. This is expected since, at first, they lack of support for capturing inter-task

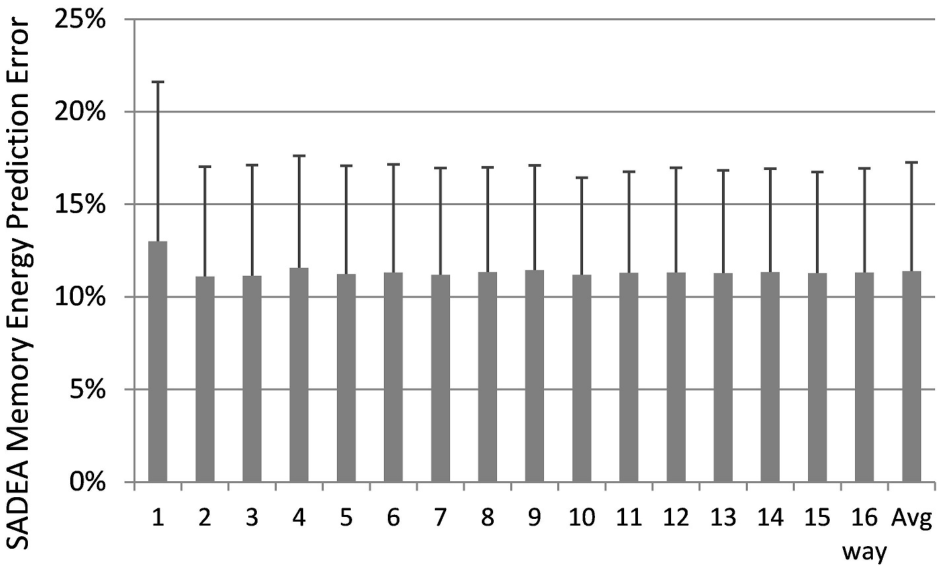


Figure 7.5: Prediction error to account DRAM memory energy to benchmarks for different LLC ways allocated running in a 2-way 2-core SMT/CMP system.

interferences, and secondly, lack of support for acknowledging the deviations of behaviors a task has in different scenarios. On average, *ES*, *PTA* and `DReAM` have prediction error over 38% across all workloads and setups. On the other hand, SADEA achieves an average 7.8% prediction error. In general, the predictions for *M* workloads and higher core-count scenarios are less accurate due to the higher interferences from co-running tasks, which are harder to eliminate in terms of energy accounting. In general, SADEA, while not being ideal, keeps inaccuracy low enough to make it usable in practice.

SADEA evaluation in SMT/CMP systems

In the CMP architecture, SADEA collects statistics during the whole execution time. Conversely, applying SADEA for a given fraction of processor resources including the core, one can only use the information collected from the separated MIBTA phases. Thus, we show the SADEA prediction error for a 2-way 2-core SMT architecture case in Figure 7.5, for configurations where each task has exactly $\frac{1}{2}$ of the core resources and in the range $1 - 16$ ways of the LLC. We observe that the prediction error in such scenario is higher than in CMP architectures. The average error exceeds 10% and the standard deviation reaches 15%. As for the CMP case, the prediction error for the 1-way LLC case is higher than for other LLC way allocations. Still, the average prediction error remains under 12%, making SADEA still a reasonable choice.

7.6 Summary

A number of mechanisms are in place to hide the (long) memory latency. Thus, when multiple tasks share the memory system, interferences can impact significantly their memory behavior, and in turn their performance and energy consumption. In this chapter we focus on sensibly accounting the memory energy to a task, which corresponds to the energy it consumes when it runs with a fraction of the processor resources in isolation. This requires discounting the effect of interference from co-runners and its own when it generates a different number of memory requests when its given resources vary.

As part of SADEA we have devised several mechanisms to predict the number of memory request, the execution time and the time using the memory system that a task should have when running in isolation with a given fraction of the processor resources. Our results show that SEA provides sufficiently accurate estimates for memory energy, yet with low-cost. Still, we see room for improvement given that SADEA accuracy is not as high as for the other mechanisms proposed in this thesis.

8

Conclusion and Future Directions

8.1 Thesis Conclusion

With the price keep growing, energy has arguably became the most expensive resource in computing systems across different computing domains. In the meantime, with processor chip integrating billions of transistors, providing several gigaflop computing power and several gigabytes of memory capacity to use, power wall has became a major stumbling block for the performance growth of computing systems. The establishment of multi-core architectures offers improved performance per Watt, by allowing many tasks to run simultaneously sharing the resources in the system. Unfortunately, in this scenario, the energy consumed by a given task becomes non-obvious, since it strongly depends on the set of co-runners which create inter-task interferences. Obtaining the power profiles of such a complex, highly-threaded system is a difficult challenge. There has already been large efforts invested on this topic. Different directions have been explored, such as refining the power measurement of the system, energy and power profiling in hardware and software, energy breakdown in hardware components and software blocks, etc.. However, these studies have all consider the hardware resources as a whole. Ignoring the fact that in current reference platform, the multicore processors, most of the hardware resource are shared by multiple tasks running in parallel. As far as we know of, no study or proposals have been made to support per-task energy measurement in the multicores.

In this Thesis, for the first time, we formalize the need for per-task energy measurement in multi-core systems by establishing a two-fold concept: per-task energy metering(PTEM) and sensible energy accounting(SEA). In the scenario where many tasks running in parallel in a multicore system. For each task, the target of PTEM is to provide estimate of the actual energy consumption at runtime based on its resource usage during execution; and SEA aims at providing estimates on the energy it would have consumed when running in isolation with a particular fraction of system's resources.

The differences between the technology and design of different components in the computer are large. For this reason, we separately apply PTEM and SEA to two main functional components of a computing system: the processor and memory system. In summary, the main contributions of this thesis are listed following:

PTEM for the Processor

First we make a case of PTEM for the processor. The model distributes the energy of the chip to the running tasks in an arbitrary workload based on the utilization of the on-chip components (e.g., cores, caches, etc.). By analyzing the impact of resources utilization on *active*, *maintenance* and *leakage* energy, we first propose an ideal PTEM model in the processor by means of tracking the activity and occupancy of all the resources in a per-task basis. This ideal model is complex and too expensive to implement. Thus, we propose an implementable and efficient design to perform PTEM in multicore processors by trading off the cost with accuracy. We illustrate how this method can accurately approach the ideal model, and thus obtain estimates of the actual energy each task consumes in the chip. State of the art models, such as Evenly Split (ES) and Proportional To Accesses (PTA) detailed in Section 2.3.4, are also evaluated and compared against the ideal model. PTEM achieves highly accurate estimates that greatly improve the state of the art.

In this thesis, we have shown how to apply PTEM to sequential and parallel applications. By deploying the proposed PTEM technique in a 2-way SMT core processor, we have seen that the metered energy for any SPEC CPU 2006 benchmark within different workloads can vary in the range of $[-25\%, 40\%]$, which sets the motivation for SEA.

PTEM for the DRAM Memory System

Similarly in the memory system, the energy used by a task is correlated with its utilization of the DRAM devices. This includes the memory activities, the states of the memory banks, the execution time of the task, and in particular, the interaction with other tasks. We propose an ideal PTEM model for the DRAM memory system that tracks all the memory resources utilized by a task in every cycle. Based on such an ideal but complex model, we have proposed a practical model with low-cost, which relies on few counters and registers to be set up in the memory controller to estimate the memory energy. Such implementation can achieve accurate predictions with respect to the ideal model, largely improving the estimations obtained with ES and PTA models.

Next, we have shown that PTEM can help significantly improving the power efficiency in a multicore processor. Energy savings may depend on the tasks' memory access frequency and access patterns (i.e. bursty versus scattered behaviors). Also, we have proven that the energy impact of memory contention highly depends on the frequency of memory accesses: programs with frequent memory accesses decrease their power at the expense of increasing their energy.

SEA for the Processor

From the PTEM outcomes, we observe that the actual energy consumed by a task heavily depends on its co-running tasks due to their interaction in shared resources. The energy variation is huge for most benchmarks, even though the same benchmark always runs with the same input in the very same platform. Then, the principle of energy accounting suggests to consistently account a fixed amount of energy to a task independently of its dynamic behavior in the workload. This principle is inherited from the performance accounting principle defined by Luque et al. [76]. Thus, the energy consumed by a task depends on itself and the resources it uses to execute. In this thesis, we propose Sensible Energy Accounting (SEA), which accounts a task the energy it would consume when it runs in isolation with a certain fraction of the resources.

To make SEA feasible, we devise a low-cost hardware mechanism to obtain at runtime an estimate of the processor energy to account to a task when it co-runs with several other tasks in a multicore system. Our proposal achieves high prediction accuracy with regards to the reference model. When compared with other state of the art models such as ES, PTA and PTEM, SEA accounts a much more consistent and fair energy cost to a task. We have also proven that, by using SEA for scheduling purposes, significant energy savings can be obtained.

SEA for the DRAM Memory System

In this thesis, the concepts introduced by SEA are also applied to the DRAM memory system using SADEA. Thus, we propose to account the memory energy a task would consume when it runs in isolation with a certain fraction of processor resources. Since various techniques have been used to hide long memory latencies, the interferences from co-running tasks in the DRAM memory system are not obvious to identify. The key to achieve SADEA consists in predicting the memory behavior a task would have in isolation.

To attain this goal, we have to separately analyze several key metrics, such as the amount of activities, the time invoking the memory system and execution time. We have shown that, with few extra registers and logic to detect the bank level accessing parallelism, SADEA provides tight estimates on the memory energy to account. Finally, we have also shown that SADEA can be combined with SEA in the processor to build an integral energy accounting system for multicore processors.

8.2 Future Works and Impact

This thesis is the first attempt showing that per-task energy in resource-sharing multicore system can be quantified in an exact way. The work done in this thesis can become fundamental for several research lines, and they can impact different computing domains where multicores are used as the reference computing platform. Based on the proposals in this thesis, it is possible to enhance the understanding of energy savings in multicore system, and therefore, inspiring energy efficient studies from different perspectives.

Implementation in Real Systems

The fact that this thesis has been performed on simulators infers a long way for it to impact real systems available in the market. Therefore, implementing PTEM and SEA proposals in real systems would be the primary focus of the future work. Although the lack of hardware support in existing computing systems limits the applicability of PTEM and SEA, these proposals have shown that a reasonable tradeoff between predictions accuracy and hardware cost can be reached. Furthermore, the implementation of such models in existing systems can directly lead to energy-aware computing, so shifting processor designs towards energy-aware architectures.

Multicore Architecture Design

Current multicore processors are usually designed to have high throughput, with good tradeoff with per-task performance degradation. PTEM and SEA proposals not only can quantify the net effect on energy consumption of a particular design, but also can enable optional energy efficient designs to better deploy the multicore processors in energy sensitive environments.

Datacenter Billing

Nowadays, cloud-computing providers tend to provide services of virtualization, such as IaaS, PaaS and SaaS. As depicted in Section 6.3.1, SEA can be used by the datacenter owner to optimize the task co-location in order to reduce the operationl cost of their infrastructure. And clients can also benefit from the consistent and fair billing. That is, when they request the same computing power to run the same tasks with the same input, the same energy cost is accounted. Of course, in addition to the energy cost, the hardware and operational cost are also needed to be correlated. Nonetheless, maintaining this principle for billing is key to keep the bills consistent.

Energy Aware Scheduling

Linux kernel developers already start to research the appropriate scheduler for heterogeneous architectures, such as the big.LITTLE architecture by ARM [64]. In such scenario, scheduler is the best place to collect information on the past, current and future information on the energy profile of a task since it controls where to place various tasks. In this line, authors in [102] present a request-level OS mechanism to meter power consumption of requests in the servers. However, in this work, per-task energy estimates cannot be accurately obtained, this is why authors of this work call for finer hardware support. PTEM and SEA cover this gap, as PTEM provides information of the past and current information on the actual energy consumed by a task, and SEA predicts the future energy consumption of a task with different resources allocation(for example, running in big and LITTLE cores). The use of PTEM and SEA can enhance the energy aware scheduler designs, instead of using current limited information obtained from CPU frequency and idle state.

Resource Allocation

As we have illustrated in Chapters 6 and 7, when different fractions of resources are allocated to a task, the impact on its energy consumption could be huge. Thus, with the outcome of PTEM and SEA, one can choose for a task an optimal viable allocation of resources to minimize its energy usage. For this purpose, we have illustrated with a simple example in Section 6.6.2. Of course, to apply this in real systems it is needed more dedicated studies.

9

Publications

Q. Liu, M. Moretó, J. Abella, F. J. Cazorla, D.A Jimenez and M. Valero. Sensible Energy Accounting with Abstract Metering for Multicore Systems. In ACM Transactions on Architecture and Code Optimization (TACO), 2015 & HiPEAC 2016 11th International Conference on High-Performance Embedded Architectures and Compilers, Prague, January 2016.

Q. Liu, M. Moretó, J. Abella, F. J. Cazorla and M. Valero. DReAM: Per-Task DRAM Energy Metering in Multicore Systems. Euro-Par 2014 Parallel Processing international conference, Porto, August, 2014.

Q. Liu, V. Jimenez, M. Moretó, J. Abella, F. J. Cazorla and M. Valero. Per-task Energy Accounting in Computing Systems. In IEEE Computer Architecture Letters, Volume 13, Issue 2, 2014.

Q. Liu, V. Jimenez, M. Moretó, J. Abella, F. J. Cazorla and M. Valero. Hardware Support for Accurate Per-Task Energy Metering in Multicore Systems. In ACM Transactions on Architecture and Code Optimization (TACO), 2013 & HiPEAC 2014 9th International Conference on High-Performance Embedded Architectures and Compilers, Vienna, January 2014.

Q. Liu, M. Moretó, J. Abella and F. J. Cazorla. Online Performance Prediction in Processors with DVFS Capabilities. ACACES 2011, Poster. 7th International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems. Fiuggi (Italy), July 10-16 2011.

Q. Liu, M. Moretó, J. Abella, F. J. Cazorla, and M. Valero. DReAM: Per-Task DRAM Energy Metering in Multicore Systems. In ACM Transactions on Design Automation of Electronic Systems (TODAES). (a journal extension of the Euro-par conference paper, submitted)

Bibliography

[1] J. Abella and A. Gonzalez. On reducing register pressure and energy in multiple-banked register files. In *Proceedings of 21st International Conference on Computer Design, 2003.*, pages 14–20, Oct 2003.

[2] J. Abella, A. González, X. Vera, and M. O'Boyle. IATAC: a smart predictor to turn-off L2 cache lines. *ACM Trans. Archit. Code Optim.*, 2(1):55–77, Mar. 2005.

[3] C. Acosta, F. J. Cazorla, A. Ramirez, and M. Valero. The MPsim simulation tool. Technical Report UPC-DAC-RR-CAP-2009-15, in UPC, 2009.

[4] D. H. Albonesi, R. Balasubramonian, S. G. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, 36(12):49–58, Dec. 2003.

[5] L. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, Dec. 2007.

[6] C. Belady and C. Malone. Data center power projections to 2014. *Proceedings of the Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronics Systems*, pages 439–444, 2006.

[7] F. Bellosa. The benefits of event driven energy accounting in power-sensitive systems. In *9th ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, EW 9, pages 37–42, 2000.

[8] A. Beloglazov, R. Buyya, Y. C. Lee, A. Zomaya, et al. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in Computers*, 82(2), 2011.

[9] R. Bertran, Y. Becerra, D. Carrera, V. Beltran, M. Gonzílez, X. Martorell, N. Navarro, J. Torres, and E. Ayguadé. Energy accounting for shared virtualized environments under dvfs using pmc-based power models. *Future Gener. Comput. Syst.*, 28(2):457–468, Feb. 2012.

[10] R. Bertran, A. Buyuktosunoglu, M. S. Gupta, M. Gonzalez, and P. Bose. Systematic energy characterization of cmp/smt processor systems via automated micro-benchmarks. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, Washington, DC, USA, 2012. IEEE Computer Society.

[11] R. Bertran, M. Gonzàlez, X. Martorell, N. Navarro, and E. Ayguadé. Potra: A framework for building power models for next generation multicore architectures. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 427–428, New York, NY, USA, 2012. ACM.

[12] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade. Counter-based power modeling methods: Top-down vs. bottom-up. *The Computer Journal*, 56(2):198–213, 2013.

[13] J. P. Bickford, R. Rosner, E. Hedberg, J. W. Yoder, and T. S. Barnett. Sram redundancy - silicon area versus number of repairs trade-off. In *IEEE/SEMI Advanced Semiconductor Manufacturing Conference*, pages 387–392, 2008.

[14] W. Bircher and L. K. John. Complete system power estimation using processor performance events. *IEEE Transactions on Computers*, 61(4):563–577, 2012.

[15] S. Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[16] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 83–94, June 2000.

[17] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *USENIX annual technical conference*, pages 21–21, 2010.

[18] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically controlled resource allocation in smt processors. In *37th International Symposium on Microarchitecture, 2004.*, pages 171–182, 2004.

[19] K. Chandrasekar, B. Akesson, and K. Goossens. Improved power modeling of ddr sdrams. In *2011 14th Euromicro Conference on Digital System Design (DSD)*, pages 99–108, Aug 2011.

[20] Y.-F. Chung, C.-Y. Lin, and C.-T. King. ANEPROF: Energy profiling for android java virtual machine and applications. In *ICPADS*, pages 372–379, 2011.

[21] CloudSigma. Cloud server host. `https://www.cloudsigma.com`, 2016. Accessed: 2016-02-05.

[22] Q. Deng, D. Meisner, L. Ramos, T. Wenisch, and R. Bianchini. Memscale: active low-power modes for main memory. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[23] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings. 29th Annual International Symposium on Computer Architecture, 2002.*, pages 233–244, 2002.

[24] Digital Reality. White paper: Powering the green data center. `https://www.digitalrealty.com/information-library/white_papers/`, 2015. Accessed: 2016-02-05.

[25] J. Donald and M. Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *Proceedings of the International Symposium on Computer Architecture*, pages 78–88, 2006.

[26] ElasticHosts. Cloud server host. `https://www.elastichosts.com`, 2016. Accessed: 2016-02-05.

[27] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. *SIGARCH Comput. Archit. News*, 39(1):319–332, Mar. 2011.

[28] European Comission. Energy prices and costs in europe. `http://ec.europa.eu/energy/doc/2030/20140122_communication_energy_prices.pdf`, 2016. Accessed: 2016-02-05.

[29] S. Eyerman and L. Eeckhout. Per-thread cycle accounting in smt processors. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 44, pages 133–144, Mar. 2009.

[30] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate cpi components. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–184, New York, NY, USA, 2006. ACM.

[31] A. Fedorova, C. Small, D. Nussbaum, and M. Seltzer. Chip multithreading systems need a new operating system scheduler. In *11th ACM SIGOPS European workshop*, 2004.

[32] M. Floyd, M. Allen-Ware, K. Rajamani, B. Brock, C. Lefurgy, A. Drake, L. Pesantez, T. Gloekler, J. Tierno, P. Bose, and A. Buyuktosunoglu. Introducing the adaptive energy management features of the POWER7 chip. In *Proceedings of the 44th Annual International Symposium on Microarchitecture*, volume 31, 2011.

[33] D. Folegnani and A. González. Energy-effective issue logic. In *Proceedings of the International Symposium on Computer Architecture*, pages 230–239, 2001.

[34] S. Ganapathy, R. Canal, A. Gonzalez, and A. Rubio. Circuit propagation delay estimation through multivariate regression-based modeling under spatio-temporal variability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 417–422, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[35] S. Ganapathy, R. Canal, A. Gonzalez, and A. Rubio. Modest: A model for energy estimation under spatio-temporal variability. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 129–134, New York, NY, USA, 2010. ACM.

[36] B. Goel, S. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati. Portable, scalable, per-core power estimation for intelligent resource management. In *Green Computing Conference, 2010 International*, pages 135–146, Aug 2010.

[37] J. Gonzalez, J. Gimenez, M. Casas, M. Moretó, A. Ramírez, J. Labarta, and M. Valero. Simulating whole supercomputer applications. *Micro, IEEE*, 31(3):32–45, 2011.

[38] Green ICT: Sustainable Computing. Is PUE still above 2.0 for most data centers? `http://www.vertatique.com/no-one-can-agree-typical-pue`, 2015. Accessed: 2016-02-05.

[39] J. Hamilton. Internet-scale service infrastructure efficiency. In *Proceedings of the International Symposium on Computer Architecture*, pages 232–232, 2009.

[40] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W.-m. W. Hwu. Superblock formation using static program analysis. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, MICRO 26, pages 247–255, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[41] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[42] H. Homayoun, S. Pasricha, M. Makhzan, and A. Veidenbaum. Dynamic register file resizing and frequency scaling to improve embedded processor performance and energy-delay efficiency. In *45th ACM/IEEE Design Automation Conference.*, pages 68–71, June 2008.

[43] D. Howard, E. Gorbatov, U. Hanebutte, R. Khanna, and C. Le. RAPL: memory power estimation and capping. In *ISLPED*, pages 189–194, 2010.

[44] C. Hu, D. A. Jiménez, and U. Kremer. Efficient program power behavior characterization. In *Proceedings of the 2nd International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'07, pages 183–197, Berlin, Heidelberg, 2007. Springer-Verlag.

[45] C. Hu, D. A. Jiménez, and U. Kremer. Combining edge vector and event counter for time-dependent power behavior characterization. In *Transactions on High-Performance Embedded Architectures and Compilers II.* Springer Berlin Heidelberg, 2009.

[46] M. C. Huang, D. Chaver, L. Pinuel, M. Prieto, and F. Tirado. Customizing the branch predictor to reduce complexity and energy consumption. *Micro, IEEE*, 23(5):12–25, Sept 2003.

[47] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *30th International Symposium on Computer Architecture (ISCA 2003), 9-11 June 2003, San Diego, California, USA*, pages 157–168, 2003.

[48] W. Huang, C. Lefurgy, W. Kuk, A. Buyuktosunoglu, M. Floyd, K. Rajamani, M. Allen-Ware, and B. Brock. Accurate fine-grained processor power proxies. In *45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 224–234, 2012.

[49] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *J. Supercomput.*, 7(1-2):229–248, May 1993.

[50] Intel Corp. Intel 64 and ia-32 architectures software developer's manual. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, 2012. Accessed: 2016-02-05.

[51] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation - a pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. http://www.glue.umd.edu/ ajaleel/workload/, 2007. Accessed: 2016-02-05.

[52] JEDEC Solid State Technology Association. JEDEC DDR3 SDRAM standard. `https://www.jedec.org/standards-documents/docs/jesd-79-3d`, 2012. Accessed: 2016-02-05.

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

[54] L. John. Oprofile internals. `http://oprofile.sourceforge.net/doc/internals/index.html`, 2016. Accessed: 2016-02-05.

[55] R. Joseph, D. Brooks, and M. Martonosi. Live, runtime power measurements as a foundation for evaluating power/performance tradeoffs. In *In Workshop on Complexity Effectice Design WCED, held in conjunction with ISCA-28*, 2001.

[56] A. Joshi, L. Eeckhout, L. John, and C. Isen. Automated microprocessor stressmark generation. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 229–239, Feb 2008.

[57] T. Juang, S. Chen, and S. Li. A novel VLSI iterative divider architecture for fast quotient generation. In *IEEE International Symposium on Circuit and Systems(ISCAS)*, 2008.

[58] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd. POWER7: Ibm's next-generation server processor. In *Proceedings of the 43th Annual International Symposium on Microarchitecture*, pages 7–15, 2010.

[59] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)*, pages 39–50, 2010.

[60] K. Kedzierski, M. Moretó, F. J. Cazorla, and M. Valero. Adapting cache partitioning algorithms to pseudo-lru replacement policies. In *2010 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2010.

[61] G. Kestor, R. Gioiosa, D. Kerbyson, and A. Hoisie. Quantifying the energy cost of data movement in scientific applications. In *IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2013.

[62] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 145–154, April 2014.

[63] J. Koomey. Growth in data center electricity use 2005 to 2010. *Analytics Press*, Aug. 2011.

[64] A. Kucheria. Linux support for ARM big.LITTLE. `http://lwn.net/Articles/481055/`, 2012. Accessed: 2016-02-05.

[65] C. Kumar, B. Madhavi, and K. Kishore. Optimal designing approach to recursive coding in vlsi design. In *Advances in Mobile Network, Communication and its Applications (MNCAPPS), 2012 International Conference on*, pages 29–33, Aug 2012.

[66] R. Kumar, V. Zyuban, and D. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of the International Symposium on Computer Architecture*, pages 408–419, 2005.

[67] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code*

Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[68] C.-M. Lee, C.-K. Chen, and R.-S. Tsay. A basic-block power annotation approach for fast and accurate embedded software power estimation. In *Very Large Scale Integration (VLSI-SoC), 2013 IFIP/IEEE 21st International Conference on*, pages 118–123, Oct 2013.

[69] D. N. Lev Mukhanov and B. D. Supinski. Alea: Fine-grain energy profiling with basic block sampling. In *Proceedings of the 2015 24th International Conference on Parallel Architectures and Compilation Techniques*, 2015.

[70] Perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`, 2015. Url date: 2015-09-28.

[71] Q. Liu, M. Moreto, J. Abella, F. Cazorla, and M. Valero. Dream: Per-task dram energy metering in multicore systems. In *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 111–123. Springer International Publishing, 2014.

[72] Q. Liu, M. Moreto, V. Jimenez, J. Abella, F. J. Cazorla, and M. Valero. Hardware support for accurate per-task energy metering in multicore systems. *ACM Trans. Archit. Code Optim.*, 10(4):34:1–34:27, Dec. 2013.

[73] Y. Liu and H. Zhu. A survey of the research on power management techniques for high-performance systems. *Softw. Pract. Exper.*, 40(11):943–964, Oct. 2010.

[74] D. Lucanin, I. Pietri, I. Brandic, and R. Sakellariou. A cloud controller for performance-based pricing. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 155–162, June 2015.

[75] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[76] C. Luque, M. Moreto, F. J. Cazorla, R. G. A. Buyuktosunoglu, and M. Valero. CPU accounting in cmp processors. In *IEEE Comput. Archit. Lett. (CAL)*, volume 9, 2009.

[77] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero. Cpu accounting for multicore processors. *IEEE Trans. Comput.*, 161, 2012.

[78] C. Luque, M. Moreto, F. J. Cazorla, and M. Valero. Fair cpu time accounting in cmp&plus;smt processors. *ACM Trans. Archit. Code Optim.*, 9(4):50:1–50:25, Jan. 2013.

[79] J. L. Manferdelli. The many-core inflection point for mass market computer systems. `http://www.ctwatch.org/quarterly/articles/2007/02/the-many-core-inflection-point-for-mass-market-computer-systems/`, 2007. Accessed: 2016-02-05.

[80] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[81] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: Online contention detection and response. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 257–265, 2010.

[82] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta. Evaluating the effectiveness of model-based power characterization. In *USENIX annual technical conference*, pages 12–12, 2011.

[83] J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang. The weather research and forecast model: software architecture and performance. In *11th Workshop on the Use of High Performance Computing in Meteorology, Reading*, 2004.

[84] Micron. Calculating memory system power for DDR3. *Micron Technical Notes*, 2007.

[85] M. Monchiero, R. Canal, and A. Gonzalez. Power/performance/thermal design-space exploration for multicore architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 19(5):666–681, May 2008.

[86] M. Moreto, F. Cazorla, A. Ramirez, and M. Valero. MLP-aware dynamic cache partitioning. In *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, pages 337–352, 2008.

[87] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTi 6.0: A tool to understand large caches. *HP Tech Report HPL-2009-85*, 2009.

[88] S. Naffziger, B. Stackhouse, T. Grutkowski, D. Josephson, J. Desai, E. Alon, and M. Horowitz. The implementation of a 2-core multi-threaded itanium family processor. *IEEE Journal of Solid-State Circuits*, pages 182–183, 2005.

[89] U. Nawathe, M. Hassan, L. Warriner, K. Yen, D. Greenhill, A. Kumar, and H. Park. Implementation of an 8-core, 64-thread, power-efficient sparc server on a chip. *IEEE Journal of Solid-State Circuits,*, 43(1):6–20, 2008.

[90] Nokia. Energy profiler. `http://nokia-energy-profiler.en.softonic.com/symbian`, 2012. Accessed: 2016-02-05.

[91] P. R. Panda, S. Roy, S. Chandrasekaran, N. Sharma, J. Kaur, S. K. Kandalam, and N. N. High level energy modeling of controller logic in data caches. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, GLSVLSI '14, pages 45–50, New York, NY, USA, 2014. ACM.

[92] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *EuroSys*, pages 153–168, 2011.

[93] P. Petoumenos, G. Psychou, S. Kaxiras, J. Cebrian Gonzalez, and J. Aragon. MLP-aware instruction queue resizing: The key to power-efficient performance. In C. Meller-Schloer, W. Karl, and S. Yehia, editors, *Architecture of Computing Systems - ARCS 2010*, volume 5974 of *Lecture Notes in Computer Science*, pages 113–125. Springer Berlin Heidelberg, 2010.

[94] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2007.

[95] K. K. Pusukuri, D. Vengerov, and A. Fedorova. A methodology for developing simple and robust power models using performance monitoring events. In *Annual Workshop on the Interaction between Operating Systems and Computer Architecture*, 2009.

[96] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *39th International Symposium on Microarchitecture.*, pages 423–432, 2006.

[97] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4):65–79, July 2010.

[98] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16–19, Jan 2011.

[99] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power management architecture of the 2nd generation intel core microarchitecture, formerly codenamed sandy bridge. In *Hot Chip 23rd symposium*, 2011.

[100] E. Salminen, T. Kangas, V. Lahtinen, J. Riihimäki, K. Kuusilinna, and T. Hämäläinen. Benchmarking mesh and hierarchical bus networks in system-on-chip context. *J. Syst. Archit.*, 53(8), Aug. 2007.

[101] M. Santoro and M. Horowitz. SPIM: a pipelined 64*64-bit iterative multiplier. *IEEE Journal of Solid-State Circuits*, 24(2), 1989.

[102] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. Power containers: an os facility for fine-grained power and energy management on multicore servers. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 65–76. ACM, 2013.

[103] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, 2001.

[104] R. Singhal. Inside intel next generation nehalem microarchitecture. In *Intel Developer Forum*, 2008.

[105] D. C. Snowdon, S. M. Petters, and G. Heiser. Power measurement as the basis for power management. In *2005 Workshop of operating system platforms for embedded real-time applications*, 2005.

[106] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *IEEE Symposium on High Performance Computer Architecture*, pages 117–128, 2002.

[107] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of*

the 38th Annual International Symposium on Computer Architecture, pages 283–294, 2011.

[108] The Standard Performance Evaluation Corporation. SPEC cpu 2006 benchmark v1.2. `http://www.spec.org/cpu2006/`, 2011. Accessed: 2016-02-05.

[109] ThinkTank Energy Products Inc. Watts up? `https://www.wattsupmeters.com`, 2016. Accessed: 2016-02-05.

[110] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture*, pages 533–544, 1998.

[111] A. Udipi, N. Muralimanohar, and R. Balasubramonian. Towards scalable, energy-efficient, bus-based on-chip networks. In *IEEE Symposium on High Performance Computer Architecture*, 2010.

[112] C. H. K. van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1260–1265, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[113] V. Venkatachalam and M. Franz. Power reduction techniques for microprocessor systems. *ACM Comput. Surv.*, 37(3):195–237, Sept. 2005.

[114] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. FAME: Fairly measuring multithreaded architectures. In *International Conference on Parallel Architectures and Compilation Techniques*, 2007.

[115] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design. A Systems Perspective.* Addison-Wesley, 1988.

[116] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 84–97, 2003.

[117] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks. Quantifying sources of error in McPAT and potential impacts on architectural studies. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 577–589. IEEE, 2015.

[118] H. Yun. Parallelism-aware memory interference delay analysis for cots multicore systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE 21th*, 2015.