THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Measurement, Modeling, and Characterization for Energy-Efficient Computing

BHAVISHYA GOEL

Division of Computer Engineering
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2016

Measurement, Modeling, and Characterization for Energy-Efficient Computing
Bhavishya Goel
ISBN 978-91-7597-411-8

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie Nr 4092
ISSN 0346-718X

Technical Report 129D
Department of Computer Science and Engineering
Computer Architecture Research Group

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 GÖTEBORG, Sweden
Phone: +46 (0)31-772 10 00

Author e-mail: `goelb@chalmers.se`

Chalmers Reposervice
Göteborg, Sweden 2016

Measurement, Modeling, and Characterization for Energy-Efficient Computing

Bhavishya Goel

Division of Computer Engineering, Chalmers University of Technology

ABSTRACT

The ever-increasing ecological footprint of Information Technology (IT) sector coupled with adverse effects of high power consumption on electronic circuits has increased the significance of energy-efficient computing in the last decade. Making energy-efficient computing a norm rather than an exception requires that system designers and programmers understand the energy implications of their design and implementation choices. This necessitates a detailed view of system's energy expenditure and/or power consumption. We explore this aspect of energy-efficient computing in this thesis through power measurement, power modeling, and energy characterization.

First, we present a quantitative comparison between power measurement data collected for computer systems using four techniques: a power meter at wall outlet, current transducers at ATX power rails, CPU voltage regulator's current monitor, and Intel's proprietary RAPL (Running Average Power Limit) interface. We compare them for accuracy, sensitivity and accessibility.

Second, we present two different methodologies to model processor power consumption. The first model estimates power consumption at the granularity of individual cores using per-core performance events and temperature sensors. We validate the methodology on six different platforms and show that our model estimates power consumption with high accuracy across all platforms consistently. To understand the energy expenditure trends across different frequencies and different degrees of parallelism, we need to model power at a much finer granularity. The second power model addresses this issue by estimating static and dynamic power consumption for individual cores and the uncore. We validate this model on Intel's Haswell platform for single-threaded and multi-threaded benchmarks. We use this power model to characterize energy efficiency of frequency scaling on Haswell microarchitecture and use the insights to implement a low overhead DVFS scheduler. We also characterize the energy efficiency of thread scaling using the power model and demonstrate how different communication parameters and microarchitectural traits affect application's energy when it scales.

Finally, we perform detailed performance and energy characterization of Intel's Restricted Transactional Memory (RTM). We use TinySTM software transactional memory (STM) system to benchmark RTM's performance against competing STM alternatives. We use microbenchmarks and STAMP benchmark suite to compare RTM an STM performance and energy behavior. We quantify the RTM hardware limitations and identify conditions required for RTM to outperform STM.

Keywords: power estimation, energy characterization, power-aware scheduling, power management, transactional memory, power measurement

Preface

Parts of the contributions presented in this thesis have previously been published in the following manuscripts.

- ▷ **Bhavishya Goel**, Sally A. McKee, Roberto Gioiosa, Karan Singh, Major Bhadauria, Marco Cesati, "Portable, Scalable, Per-Core Power Estimation for Intelligent Resource Management" in *Proceedings of the 1st International Green Computing Conference*, Chicago, USA, August, 2010, pp.135-146.
- ▷ **Bhavishya Goel**, Sally A. McKee, Magnus Själander, "Techniques to Measure, Model, and Manage Power," in *Advances in Computers 87*, 2012, pp.7-54.
- ▷ **Bhavishya Goel**, Ruben Titos-Gil, Anurag Negi, Sally A. McKee, Per Stenstrom, "Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell" in *Proceedings of 28th IEEE International Parallel and Distributed Processing Symposium*, Phoenix, USA, May, 2014, pp.615-624

The following manuscript has been accepted but is yet to be published.

- ▷ **Bhavishya Goel**, Sally A. McKee, "A Methodology for Modeling Dynamic and Static Power Consumption for Multicore Processors" in *Proceedings of 30th IEEE International Parallel and Distributed Processing Symposium*, Chicago, USA, May, 2016

The following manuscripts have been published but are not included in this work.

- ▷ **Bhavishya Goel**, Magnus Själander, Sally A. McKee, "RTL Model for Dynamically Resizable L1 Data Cache" in *Swedish System-on-Chip Conference*, Ystad, Sweden, 2013
- ▷ Magnus Själander, Sally A. McKee, **Bhavishya Goel**, Peter Brauer, David Engdal, Andras Vajda, "Power-Aware Resource Scheduling in Base Stations" in *19th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Singapore, Singapore, July, 2012, pp.462-465.

Acknowledgments

I will like to thank following people for supporting me during my PhD and contributing to this thesis directly or indirectly.

▷ Professor Sally A. McKee for giving me the opportunity to work with her, for giving me the confidence boost during the lows and for celebrating my highs. She always trusted me to choose my own research path but was always there whenever I needed her for counseling. She is also an excellent cook and her brownies and palak paneers have got me through various deadlines.

▷ Magnus Själander for helping me with my research during the early years of my PhD and letting me bounce ideas off him. His unabated enthusiasm for research is very contagious.

▷ Madhavan Manivannan for being a great friend and colleague; and for spending loads of his precious time to review my papers and thesis.

▷ Professor Lars Svensson for helping me with practical issues during his time as division head and for giving me critical feedback to improve the technical and non-technical aspects of this thesis.

▷ Professor Per Larsson-Edefors for teaching some of the most entertaining courses I have taken at Chalmers and for his valuable feedback from time to time.

▷ Professor Per Stenström for being an inspiration in the field of computer architecture.

▷ Professor Lieven Eeckhout for evaluating my licentiate thesis and for his insightful comments on my research.

▷ Anurag Negi for being an accommodating roommate and a helpful co-author.

▷ Ruben Titos for helping me in getting started with the basics of transactional memory, for doing his part in the paper we co-authored and for cheering me up with his positive vibes.

▷ Karan Singh, Vincent M. Weaver and Major Bhadauria for getting me started with the power modeling infrastructure at Cornell and answering my naïve questions patiently.

▷ Roberto Gioiosa and Marco Cesati for helping me run experiments for my first paper.

▷ Jacob Lidman for being an amazing study partner, for sharing office space with me and putting up with my rants and bad jokes.

▷ Alen, Angelos, Bapi, Dmitry, Kasyab, Mafijul, Waliullah, Gabriele, Vinay, Jochen, Fatemeh, Behrooz, Chloe, Petros, Risat, Stavros, Alirad, Ahsen, Viktor, Miquel, Vassilis, Yiannis and Ivan for being wonderful colleagues and friends.

▷ Peter, Rune and Arne for proving invaluable IT support.

▷ Rolf Snedsböl for managing my teaching duties.

▷ Eva, Lotta, Tiina, Jonna, and Marianne for providing excellent administrative support during my time at Chalmers.

▷ All the colleagues and staff at Computer Science and Engineering department for creating a healthy work environment.

▷ Anthony Brandon for giving me valuable support when I was working with ρVEX processor for ERA project.

▷ European Union for funding my research.

▷ My family for motivating me to push for my goals and being there as a backup whenever I needed them.

Bhavishya Goel
Göteborg, May 2016

Contents

List of Figures

List of Tables

1 Introduction

In the last decade, the Information Technology (IT) sector has aggressively pursued the goal of making computers and peripherals more environment friendly. This has resulted in the term *Green Computing* becoming an increasingly common buzzword in the IT community. It can be defined as the practice of putting greater emphasis on reducing the ecological footprint of computing devices at all the stages of the product life cycle. This includes manufacturing, operation and disposal. In this thesis, we mainly focus on the operational aspect of green computing. Various contributing factors have encouraged the system designers and researchers to focus on the operational aspect of green computing. The list below is an attempt at identifying these factors and explaining their significance.

- **Global Warming:** One of the biggest reasons that has accelerated the push towards green computing is the increasing consensus among scientific community [1] that the global average temperatures are rising and that this climate change is anthropogenic in nature. This realization calls upon all industries to introspect the ecological footprint of their respective sectors. The increasing usage of IT in the everyday lives of people means that the ecological footprint of IT sector is significant and growing. As per a European Commission press release in 2013 [2], IT products and services are responsible for 8-10% of the European Union's electricity consumption and up to 4% of its carbon emissions. Murugesan [3] notes that each personal computer in use in 2008 was responsible for generating about a ton of carbon dioxide per year. Data centers consumed 1.5-2% of global electricity in 2011 and this is growing at a rate of 12% per year [4]. These statistics underscore the importance of reducing the energy we expend to avail IT services.

- **Electricity Bills:** Another reason for emphasizing energy expenditure as first order design constraint is the electricity bills for operating computers. As per a study published by National Resources Defense Council [5], the data center electricity consumption cost US businesses almost 13 billion dollars in the year 2013. This combined with the electricity bills incurred by private consumers for operating personal computers and electronic goods at home is a strong reason to design energy-efficient computers.
- **Maintenance Costs:** The power consumed by computing resources is dissipated as heat. This heat dissipation can result in heat build-up around the electronic circuitry. This can in turn increase the temperature of the electronic circuits beyond their operating range and cause performance degradation or critical failures. To avoid this scenario, the computers need a mechanism to dissipate this heat efficiently. This can be done by ensuring adequate natural air flow around the circuits, utilizing heat sinks, fans, liquid coolants or a combination thereof. All these heat dissipation mechanisms cost resources in terms of real estate and/or equipment expenditure. Moreover, high temperatures also lead to the shortening of expected lifespan of the products. The best way to combat these maintenance costs is to design systems that are more energy-efficient.
- **Battery Life:** There has been a marked increase in the usage of hand held and portable electronic goods like smart phones, cameras, fitness trackers, electronic watches, and other wearable electronics. The consumers expect every new generation of these devices to provide more features and last longer than before. This makes the battery life a critical parameter for any portable consumer device. Unfortunately, the battery technology has not kept up with the pace of feature enhancements in these energy hungry consumer products increasing the importance of designing energy-efficient hardware and writing energy-efficient software.

Apart from energy efficiency, the practice of green computing includes other operational goals with subtle distinctions. Below we attempt to distinguish between these goals:

- Minimize **total energy expenditure**.
- Minimize **average power**.
- Minimize **peak power**.

- Minimize **energy-delay product** (EDP) or **energy-delay squared product** (ED^{2}P).

It is important to note that since energy is defined as product of average power and execution time, the reduction in average power may be offset by the increase in execution time resulting in overall increase in total energy expenditure. The systems whose design and/or operation is constrained by total energy expenditure are termed as **energy-aware** systems. The systems that are constrained by either the average or instantaneous power consumption values are called **power-aware** systems. In this thesis, we use the term *energy efficiency* as an umbrella term for the above listed goals unless otherwise specified.

The challenge of improving the energy efficiency of computing resources needs to be tackled at various abstraction layers. Below, we list these layers of abstraction and discuss energy/power saving techniques that can be employed at these layers. Some of these techniques can be isolated within a single layer of abstraction, while others require information exchange across multiple layers. The techniques discussed here just serve as examples and are by no means exhaustive.

- **CMOS Device** technology is the lowest layer of abstraction at which power saving techniques can be implemented. These techniques either target dynamic power consumption which results from transistor switching activity or static power consumption which results from leakage current. This discussion highlights some of the prominent examples of low power techniques for each of them.

  A common technique used to lower dynamic power is clock gating whereby an additional enable signal is added to the clock tree of CMOS circuits that can be pulled down when the circuit is not in use. This ensures that the flip-flops do not switch states when they are idle and consequently, do not consume dynamic power. Another common methodology to reduce dynamic power consumption is dynamic voltage and frequency scaling (DVFS). The dynamic power consumption depends on the product $C \times V^2 \times F$ where C is switching capacitance, F is frequency and V is supply voltage. The reduction in voltage results in quadratic reduction in dynamic power consumption. But voltage scaling is coupled with frequency scaling since reduced supply voltage limits the maximum frequency at which the device can operate. This can potentially result in linear increase in execution time but reduction in overall dynamic energy expenditure due to the quadratic relationship between dynamic power and voltage.

  The shrinking transistor feature size has resulted in increase in subthreshold current leakage which has increased the static power consumption. As a result, with each subsequent adoption of smaller technology node, the static power consumption, if left unchecked, is bound to become more prominent as the fraction of total chip power consumption. Various techniques have been proposed and implemented to reduce the static power consumption of the chip. The sleep transistors [6] have been widely used to power gate a block of circuits, execution units or the entire compute core. The dual threshold voltage techniques [7] are also an effective methodology to reduce the standby power consumption for achieving the low power goals. Other low power techniques like dynamic body biasing [8] and input vector control [9] have also proven to be highly effective to reduce leakage current.

- **Computer Architecture** is the next level of abstraction at which the energy efficiency of computers can be targeted for optimization. It spans three aspects of computer design — instruction set architecture (ISA), computer organization, and microarchitecture design [10]. The energy aware computing techniques can either be realized for any of these aspects individually or their combination thereof.

  The ISA can be simplified to reduce the total number of instructions (Reduced Instruction Set Computing — RISC) supported. This facilitates a simpler hardware design which can trade-off performance for achieving lower power consumption.

  The computer organization refers to high level aspects of the processor design. This includes decisions about number and type of cores on chip, processor pipeline, cache size and hierarchy, the interconnects, etc.

  Until early 2000s, the commodity and server processors mainly relied on instruction level parallelism (ILP) and higher frequencies for achieving performance gains with each new generation of processors. This, however, resulted in higher heat dissipation per unit of die area because of the quadratic relationship between the processor frequency and the dynamic power consumption. Eventually, the processor design hit power wall — it was no longer sustainable to keep increasing processor frequency because of the heat limitations on the reliability of semiconductor devices. As a result, the industry moved towards chip multiprocessors (CMP) in order to continue reaping performance gains made available through shrinking transistor size. The multi-core processors can be further optimized to save energy by incorporating cores of different sizes on the same die [11]. The more CPU-intensive tasks can then be run on bigger cores while tasks which do not require high processing capability can be run on smaller more energy-efficient cores.

  Within a processor core, various trade-offs can be made between performance and power. For example, in an out-of-order processor, the size of various microarchitectural features like Reservation Station, Re-order Buffer, load/store queue, integer/floating point register file, etc. can be increased to improve performance or decreased to save power at the cost of performance. These decisions also depend on the characteristics of the expected workload since increase in execution time can also lead to increase in total energy expenditure. Another area where power-performance trade-off comes into play is the cache hierarchy. The on-chip caches are critical in hiding the memory latency but they are also one of the biggest consumers of power among all on-chip microarchitectural features. The on-chip cache size can be increased to improve workload performance, but it will result in increase in both static and dynamic power. The cache associativity is another cache feature that can be increased to maximize performance gains but if not chosen properly, can result in overall increase in power consumption.

The dynamic power consumed by on-chip interconnects is growing in significance [12] due to greater number of on-chip compute units and the high-level of switching activity in the interconnects. The system architects need to consider the impact of interconnect power consumption while making decisions about increasing on-chip bandwidth.

For a similar computer organization, a plethora of power saving techniques can be implemented at lower-levels of hardware design. For example, hardware designers need to decide the level of clock gating and power gating to balance energy efficiency with increased latency. Most of the modern processors implement some kind of sleep states which are activated when the processor is idle. The sleep states reduce power consumption by turning off clocks, scaling down voltage and shutting down parts of processor through power gating. There are multiple levels of sleep states. The shallower levels are less aggressive in saving power but require fewer cycles to be woken up from. The deeper levels target more aggressive power saving but require more cycles while waking up. The mechanism to select an appropriate sleep state for current idle period can either be taken fully by hardware or by a mechanism that is a combination of hardware and software.

Apart from the techniques to save power when CPU is idle, a plethora of techniques exist to save power when CPU is actively executing a workload. Significant power savings can be achieved by using DVFS technique described above. DVFS implementation details like number of distinct voltage-frequency operating points and the number of separate voltage planes are taken at the level of microarchitectural design. These decisions need to consider the trade-off between design complexity and potential power savings. The cache hierarchy can also be optimized for saving power during active states by techniques like dynamic cache reconfiguration or dynamic cache resizing. In the first technique, the caches are reconfigured at runtime to use the available cache blocks more efficiently [13–16]. In the second technique, the parts of cache are either fully turned-off [17, 18] or put in low-power state [19] during low cache occupancy periods to save power.

- **System** components like CPU chip, fan, disk, graphics card, etc. have their individual power states and hence, a system-level power management scheme is required to handle the complex interface between these components. An example of such a scheme is *Advanced Configuration and Power Interface*(ACPI) [20], a standardized interface that can be used by operating systems to manage the power states of computer peripherals. For large-scale system installations like datacenters, the systems need to be energy proportional to ensure energy efficiency at all levels of system utilization [21]. This requires designing system components like power supply units, memory, disk, etc. with a flat efficiency curve. Alternatively, installation of heterogeneous nodes can improve energy proportionality such that the job scheduler can schedule jobs to appropriately sized nodes based upon the compute requirements of the job [22].
- **Software** has a major impact on the overall energy expenditure of the system. The software power optimization strategies can be implemented at compile time or runtime. Most of the energy savings from compiler optimizations are side effect of performance and memory optimizations. The compiler optimizations that have a significant impact on energy expenditure include instruction scheduling, loop optimizations, reducing cache miss rates and efficient register assignment [23]. However, the research on energy-aware compilers is still in infancy.

  At the operating system level, DVFS management policies play a big role in saving energy. The DVFS schedulers can detect the slack in tasks that are executed periodically and finish before their deadlines. These slacks can be exploited to eliminate idle periods by reducing system frequency and hence save energy without degrading performance. Alternatively, the operating system can take advantage of the discrepancy between CPU and memory frequencies by scaling down CPU frequency during the memory bound phase of the application. The operating system can also implement power capping. The OS can monitor system power consumption in real time [24, 25] to make sure that it does not breach a pre-defined power envelope. If the breach is detected, the OS can reduce system power consumption by throttling performance of individual tasks or entire system. Energy-aware schedulers can be used for scheduling tasks within a system [26, 27] or for scheduling jobs across nodes on datacenters [28–31].

Both power-aware and energy-aware systems generally require some kind of feedback from the system about the power and/or energy usage, either online or offline. The focus of this thesis is how to get this information through power measurement, power modeling, and energy characterization.

1.1 Power Measurement

The first-step towards power-aware systems is to measure power consumption. Readings from power measurement infrastructure can be used to identify power inefficiencies in hardware and software, which in turn can identify energy inefficiencies. A setup for measuring system/processor power consumption should have following desirable attributes:

- High accuracy;
- High sampling rate;
- High sensitivity;
- Non-interference with the system-under-test;
- Low cost; and
- Ease of setup and use.

To measure the power consumption of the entire system, an off-the-shelf power meter can be employed which is connected between the wall outlet and the input of the power supply unit of the system. The high-end server systems (like the ones used in datacenters) sometimes have in-built power measurement mechanisms [32] that can report the system power consumption over network.

Apart from wall outlet, power measurement can also be done on the ATX power rails that supply power from the power supply unit to the motherboard and other system components like fans, disks and optical drives. The power measurements done on ATX rails hide the energy wasted in AC to DC conversion in power supply unit. But they are useful in isolating the power consumption of individual system components. ATX power measurements can be done using equipments like clamp ammeters [33], sense resistors [34, 35] and current transducers.

To get the most accurate power measurement of processor chip, the measurement needs to be done directly at the processor supply voltage to hide the inefficiency of onboard voltage regulator's DC-DC conversion. Most modern voltage regulators have a current monitor pin which outputs voltage proportional to the instantaneous current consumed by the processor chip. The signal from this output can be sampled using a digital multimeter [36] or a data acquisition unit to measure the processor power consumption.

Research studies have made use of live power measurement to schedule tasks and allocate resources at the level of individual processors [34, 36–40] and large data centers [28–31, 41–43]

1.2 Power Modeling

Depending on the requirements and costs, it may not be possible or desirable to use actual power measurements. For example, on-chip power management software may require detailed breakdown of the power consumption of various components of the chip. Moreover it may be too costly to deploy the power measurement setup at multiple machines, or the response time of the power measurement setup may not be fast enough for a particular power-aware technique. An alternative to actual power measurement is to estimate the power consumption using models that provide power estimates based on performance events in the system. They can be implemented in either hardware or software. Power models should have following properties:

- Small delay between input and output;
- Low overhead — low CPU usage if software model and low hardware area if hardware model;
- High accuracy;
- High sensitivity;
- Desired power consumption breakdown of individual components (like cores or microarchitectural components).

Based on the requirements of the power-aware technique and trade-offs among costs, accuracy, and overhead, the system designer must decide whether to implement hardware or software power model. Intel introduced a hardware power model in its Core i7 processors starting with the Sandy Bridge microarchitecture [44]. The values from this power model are available to the software using Model Specific Registers (MSR) through the Running Average Power Limit (RAPL) interface. Similar model-based power estimates are available for AMD processors through their Application Power Management (APM) Interface [45], and on NVIDIA GPU processors through NVIDIA Management Library (NVML) [46]. In addition to these relatively new power models implemented in hardware by chip manufacturers, researchers have proposed software power models derived from performance events [34, 42, 47–54].

The information gained by power modeling can be used by resource management software to make power-aware decisions. Power/energy models have been used in many research studies to propose power-aware strategies for controlling DVFS policies [36, 37], task scheduling in chip multiprocessors [41] and data centers [28, 29], power budgeting [34, 39], energy-aware accounting and billing [55], and avoiding thermal hotpsots in chip multiprocessors [56, 57].

1.3 Energy Characterization

Energy characterization analyzes the system's energy expenditure by varying workload characteristics. This can be used to devise software and hybrid power-aware strategies, optimize workloads for energy efficiency, and optimize system design. Some of the challenges for energy characterization are:

- **Choosing representative workloads.** Workloads selected for characterization studies should be representative of those that are likely to be run on the system. The workloads should also stress the full spectrum of identified system characteristics to cover all possible corner cases.
- **Choosing good metrics.** The selection of the metrics to characterize and compare system energy expenditure depends on the type of characterization study and the emphasis that the researchers wants to put on delay versus energy expenditure. Possible metrics include total energy, average power, peak power, dynamic power, energy-delay product, energy-delay-squared product, power density, MIPS/watt (Million instructions per second per watt) and FLOPS/watt (Floating point operations per second per watt).
- **Setting up appropriate power metering/modeling infrastructure.** Any energy characterization study requires a means to measure/estimate and log system power consumption. Depending on the type of study, measurement factors like accuracy and temporal granularity must be considered. It may be useful to be able to decompose power consumption figures for different system components but that support may or may not exist. Researchers need to consider these factors and decide between measuring actual power versus modeling the power estimates.

Researchers have used energy characterization to understand energy-efficiency of mobile platforms [58–60] and desktop/server platforms [42, 61–63]. Energy characterization can also be used to analyze the energy efficiency of specific features of the system [64, 65]. The energy behaviors thus characterized can be used, for example, to identify software inefficiencies [58, 59, 61], manage power [42], analyze power/performance trade-offs [66], and compare energy efficiency of competing technologies [64]. Apart from actual power measurement, power models can prove to be useful for energy characterization [64, 66, 67].

1.4 Contributions

This thesis makes following contributions:

Power Measurement (Chapter 2). We develop an infrastructure to measure power consumption at three points in voltage supply path of the processor: at the wall outlet, at the ATX power rails and at the CPU voltage regulator. We do a qualitative comparison for the measurements sampled from the three points for accuracy and sensitivity. We discuss the advantages and disadvantages of each sampling point. We show that sampling power at ATX power rails provides the best trade-off between accuracy and accessibility. We test Intel's digital power model (available to the software through Running Average Power Limit (RAPL) interface) for accuracy, overhead and temporal granularity. We demonstrate that the RAPL model estimates power with good mean accuracy but suffers with infrequent high deviations from the mean.

Per-core Power Modeling (Chapter 3). We build upon the power model developed by Singh et al. [52, 68] that estimates power consumption of individual cores in chip multiprocessors. We port and validate the model across multiple platforms, improve model accuracy by augmenting microbenchmark and counter selection methodology, provide analysis for model estimation errors, and show the effectiveness of the model for the meta-scheduler that uses multiple DVFS operating points and process suspension to enforce power budget.

Static and Dynamic Power Modeling (Chapter 4). We present a methodology to estimate the static and dynamic power consumption of individual cores and those of processor uncore. We identify the contributions of activity in private L2 cache and shared L3 cache to dynamic power consumption of the chip.

Energy Characterization of Frequency and Thread Scaling (Chapter 4). We use our power models to characterize energy efficiency of dynamic frequency scaling and thread scaling on Haswell microarchitecture. We show that lowering system frequency does not guarantee reduction in energy expenditure and that the frequency at which lowest energy expenditure is achieved depends on memory bound fraction of the application. We identify communication parameters that affect the thread scaling energy trends, namely serial fraction, core-to-core communication overhead, bandwidth contention, and locking mechanism overhead and demonstrate the effect of each of these parameters on the energy expenditure. We identify that uncore energy, especially uncore static energy, is a major source of energy inefficiency on Haswell microarchitecture.

Characterization of Intel's Restricted Transactional Memory (Chapter 5). We present a detailed evaluation of Intel's Restricted Transactional Memory (RTM) performance and energy expenditure. We compare RTM behavior to that of the TinySTM, a software transactional memory system, first by running microbenchmarks, and then by running the STAMP benchmark suite. We quantify the RTM hardware limitations concerning its read/write-set size, duration and overhead. We find that RTM performs well when transaction working-set fits inside cache. RTM also handles high contention workloads better than TinySTM.

1.5 Thesis Organization

The rest of this thesis is organized as follows:

- In **Chapter 2**, we explain different techniques to measure power consumption of the system, compare their intrusiveness, and give experimental results to show their accuracy and sensitivity. We test Intel's hardware power model for accuracy, sensitivity, and update granularity and discuss the results.
- In **Chapter 3**, we present a per-core, portable, scalable power model and show the validation results across multiple platforms. We show the effectiveness of the model by implementing it in an experimental meta-scheduler power-aware scheduling.
- In **Chapter 4**, we present a methodology to model static and dynamic power consumption of core and uncore components of multi-core processors. We use the power model for energy characterization of frequency scaling on Haswell processor and use the characterization data for implementing a low overhead on-line DVFS scheduler. Similarly, we use the model for energy characterization of thread scaling and show how different communication parameters affect the energy expenditure trends for core and uncore.
- In **Chapter 5**, we present performance and energy expenditure characterization results for Restricted Transactional Memory (RTM) implementation on the Intel Haswell microarchitecture. We use microbenchmarks and the STAMP benchmark suite to compare the performance and energy efficiency of RTM to TinySTM — a software transactional memory implementation.
- In **Chapter 6** we present our concluding remarks for the work presented in this thesis.

2 Power Measurement Techniques

Designing intelligent power-aware computing technologies requires an infrastructure that can accurately measure and log the system power consumption and preferably that of the system's individual resources. Resource managers can use this information to identify power consumption problems in both hardware (e.g., hotspots) and software (e.g., power-hungry tasks) and to address those problems (e.g., through scheduling tasks to even out power or temperature across the chip) [49,69,70]. A measurement infrastructure can also be used for power benchmarking [71,72], power modeling [42,49,50,69] and power characterization [61,73,74]. In this chapter, we compare different approaches to measuring actual power consumption on the Intel Core^{TM} i7 platform. We discuss these techniques in terms of their invasiveness, ease of use, accuracy, timing resolution, sensitivity, and overhead. The measurement techniques demonstrated in this chapter can be applied to other platforms, subject to some hardware support.

In the absence of techniques to measure actual power consumption, model-based power consumption estimation is also a viable alternative. Intel's Running Average Power Limit (RAPL) interface [44], AMD's Application Power Management (APM) interface [45] and NVIDIA's Management Library (NVML) [46] interface make model-based energy estimates available to the operating system and user applications through

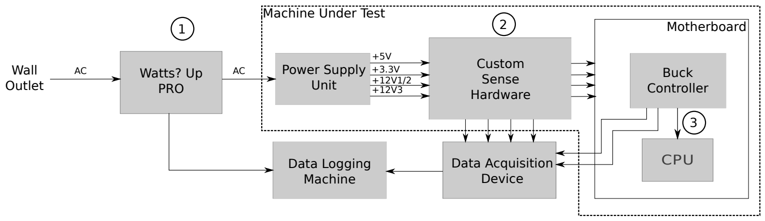


Figure 2.1: *Power Measurement Setup*

model-specific registers, thereby enabling the software to make power-aware decisions.

In the rest of this chapter, we first describe the methodology of three techniques to measure actual power consumption, discuss their advantages and disadvantages, and collect experimental results on an Intel Core^{TM}i7 870 platform to compare their accuracy and sensitivity. We then compare one of these measurement techniques — reading power measurement from the ATX (Advanced Technology eXtended) power rails — to Intel's RAPL implementation on Core^{TM}i7 4770 (Haswell).

2.1 Power Measurement Techniques

Power consumption can be measured at various points in a system. We measure power consumption at three points, as shown in Fig. 2.1:

1. The first and least invasive method for measuring the power of an entire system is to use a power meter like the *Watts up? Pro* [75] plugged directly into the wall outlet;
2. The second method uses custom sense hardware to measure the current on individual ATX power rails; and
3. The third and most invasive method measures the voltage and current directly at the CPU voltage regulator.

2.1.1 At the Wall Outlet

The first method uses an off-the-shelf (*Watts up? Pro*) power meter that sits between the system under test and the power outlet. Note that to prevent data logging activity from disturbing the system under test, we use a separate machine to collect measurements for all three techniques, as shown in Fig. 2.1. Although easy to deploy and non-invasive, this meter delivers only a single system measurement, making it difficult to separate the power consumption of different system components. Moreover, the measured power values are inflated compared to actual power consumption due to inefficiencies in the system power supply unit (PSU) and on-board voltage regulators. The acuity of the measurements is also limited by the (low) sampling frequency of the power meter: one sample per second (here on referred to as Sa/s) for the *Watts up? Pro*. The accuracy of the system power readings depends on the accuracy specifications provided by the manufacturer ($\pm 1.5\%$ in our case). The overall accuracy of measurements at the wall outlet is affected by the mechanism converting between alternating current (AC) to direct current (DC) in the power supply unit. When we discuss measurement results below, we examine the accuracy effects of the AC-DC conversion done by the PSU.

This approach is suitable for studies of total system power consumption instead of individual components like the CPU, memory, or graphics cards [76,77]. It is also useful in power modeling research, where the absolute value of the CPU and/or memory power consumption is less important than the trends [69].

2.1.2 At the ATX Power Rails

The second methodology measures current on the supply rails of the ATX motherboard's power supply connectors. As per the ATX power supply design specifications [78], the power supply unit delivers power to the motherboard through two connectors, a 24-pin connector that delivers +5.5V, +3.3V, and +12V, and an 8-pin connector that delivers +12V used exclusively by the CPU. Table 2.1 shows the pinouts of these connectors. Depending on the system under test, the pins belonging to the same power region may be connected together on the motherboard. In our case, all +3.3 VDC pins are connected together, as are all +5 VDC pins and +12V3 pins. Apart from that, the +12V1 and +12V2 pins are connected together to supply current to the CPU. Hence, to measure the total power consumption of the motherboard, we can treat these connections as four logically distinct power rails — +3.3V, +5V, +12V3, and +12V1/2.

For our experiments, we develop custom measurement hardware using current transducers from LEM [79]. These transducers use the Hall effect to generate an output voltage in accordance with the changing current flow. The top-level schematic of the hardware is shown in Fig. 2.2, and Fig. 2.3 shows the manufactured board. Note that when designing such a printed circuit board (PCB), care must be taken to ensure that the current capacity of the PCB traces carrying the combined current for the ATX power rails is sufficiently high and that the on-board resistance is as low as possible. We use a PCB with 105 micron copper instead of the more widely used thickness of 35 microns. Traces carrying high current are at least 1 cm wide and are backed by thick-stranded wire connections, when required. The current transducers need +5V supply voltage, which is provided by the +5VSB (stand by) rail from the ATX connector. Using +5VSB for the

(a) 24-pin ATX Connector Pinout

Pin	Signal	Pin	Signal
1	+3.3 VDC	13	+3.3 VDC
2	+3.3 VDC	14	-12 VDC
3	COM	15	COM
4	+5 VDC	16	PS_ON
5	COM	17	COM
6	+5 VDC	18	COM
7	COM	19	COM
8	PWR OK	20	Reserved
9	5 VSB	21	+5 VDC
10	+12 V3	22	+5 VDC
11	+12 V3	23	+5 VDC
12	+3.3 VDC	24	COM

(b) 8-pin ATX Connector Pinout

Pin	Signal	Pin	Signal
1	COM	5	+12 V1
2	COM	6	+12 V1
3	COM	7	+12 V2
4	COM	8	+12 V2

Table 2.1: *ATX Connector Pinout*

transducer's supply serves two purposes. First, because the +5VSB voltage is available even when the machine is powered off, we can measure the base output voltage from the current transducers for calibration purposes. Second, because the current consumed by the transducers themselves (∼28mA) is drawn from +5VSB, it does not interfere with our power measurements. We sample and log the analog voltage output from the current transducers using a data acquisition (DAQ) unit from National Instruments (NI USB-6210 [80]).

As per the LEM datasheet, the base voltage of the current transducer is 2.5V. Our experiments indicate that the current transducer produces an output voltage of 2.494V when zero current is passed through its primary turns. The sensitivity of the current transducer is 25mV/A, hence the current can be calculated as in Eq. 2.1.

$$I_{out} = \frac{V_{out} - BASE\_VOLTAGE}{0.025} \tag{2.1}$$

We verify our current measurements by comparing against the output from a digital multimeter. The power consumption can then be calculated by simply multiplying the current with the respective voltage. Apart from the ATX power rails, the PSU also provides separate power connections to the hard drive, CD-ROM, and cabinet fan. To calculate the total PSU load without adding extra hardware, we disconnect the I/O devices and fan, and we boot our system from a USB memory device powered by the motherboard. The total power consumption of the motherboard can then be calculated

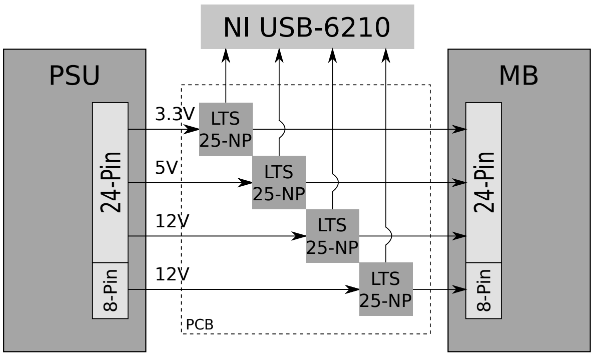


Figure 2.2: *Measurement Setup on the ATX Power Rails*

as in Eq. 2.2.

$$P = I_{3.3V} * V_{3.3V} + I_{12V3} * V_{12V3} + I_{5V} * V_{5V} + I_{12V1/2} * V_{12V1/2} \tag{2.2}$$

The theoretical current sensitivity of this measurement infrastructure can be calculated by dividing the voltage sensitivity of the DAQ unit (47μV) by the current sensitivity of the LTS-25NP current transducers from LEM (25mV/A). This yields a current sensitivity of 2mA.

This approach improves accuracy by eliminating the complexity of measuring AC power. Furthermore, the approach enjoys greater sensitivity to current changes (2mA) and higher acquisition unit sampling frequencies (up to 250000 Sa/s). Since most modern motherboards have separate supply connectors for the CPU(s), this approach facilitates distinguishing CPU power consumption from that of other motherboard components. Again, this improvement comes with increased cost and complexity: the sophisticated DAQ unit is priced an order of magnitude higher than the power meter, and we had to build a custom board to house the current transducer infrastructure.

2.1.3 At the Processor Voltage Regulator

Although measurements taken at the motherboard supply rails factor out the power supply unit's efficiency curve, they are still affected by the efficiency curve of the on-board voltage regulators. To eliminate this source of inaccuracy, we investigate a third approach. Motherboards that follow Intel's processor power delivery guidelines (Voltage Regulator-Down (VRD) 11.1 [81]) provide a load indicator output (IMON/Iout) from the processor voltage regulator. This load indicator is connected to the processor for

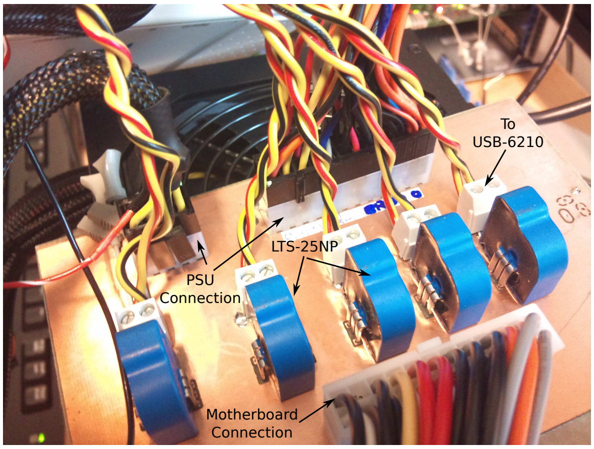


Figure 2.3: *Our Custom Measurement Board*

use by the processor's power management features. This signal provides an analog voltage linearly proportional to the total load current of the processor. We use this current sensing pin from the processor's voltage regulator chip (CHL8316 [82], in our case) to acquire real-time information about total current delivered to the processor. We also use the voltage output at the V_CPU pin of the voltage regulator, which is directly connected to the CPU voltage supply input of the processor. We locate these two signals on the motherboard and solder wires at the respective connection points (the resistor/capacitor pads connected to these signals). We connect these two signals and the ground point to our DAQ unit, logging the values read on the separate test machine. This current measurement setup is shown in Fig. 2.4.

The full voltage swing of the IMON output is 900mV for the full-scale current of 140A (for the motherboard under test). Hence, the current sensitivity of the IMON output comes to about 6.42mV/A. The theoretical sensitivity of this infrastructure depends on the voltage sensitivity of the DAQ unit (47μV) and its overall sensitivity to current changes comes to 7mA. This sensitivity is less than that for measuring current at the ATX power rails, but the sensitivity may vary for different voltage regulators on different motherboards. This method provides the most accurate measurements of absolute current feeding the processor, but it is also the most invasive, as it requires soldering wires on the motherboard. Moreover, these power measurements are limited to processor power consumption (we get no information about other system components). For example, for memory-intensive applications, we can account for power consumption effects of the external bus transactions triggered by off-chip memory accesses, but this method provides no means of measuring power consumed in the DRAMs. The accuracy

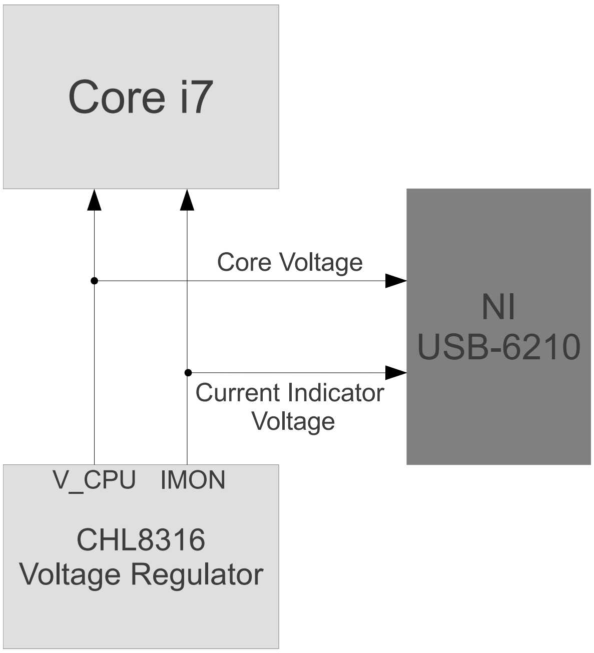


Figure 2.4: *Measurement Setup on CPU Voltage Regulator*

of the IMON output is specified by the CHL8316 datasheet to be within $\pm7\%$. This falls far below the 0.7% accuracy of the current transducers at the ATX power rails[1].

2.1.4 Experimental Results

We use an Intel Core^{TM}i7 870 processor to compare power measurement readings at the wall outlet, at the ATX power rails, and directly on the processor's voltage regulator.

The *Watts Up? Pro* measures power consumption of the entire system at the rate of 1 Sa/s, whereas the data acquisition unit is configured to capture samples at the rate of 40000 Sa/s from the four effective ATX voltage rails (+12V1/2, +12V3, +5V and +3.3V) and the V_CPU and the IMON outputs of the CPU voltage regulator. We choose this rate because the combined sampling rate of the six channels adds up to 240K Sa/s, and the maximum sampling rate supported by the DAQ is 250K Sa/s. To remove background noise, we average the DAQ samples over a period of 40 samples, which effectively

[1]The accuracy specifications of the processor's voltage regulator may differ for different manufacturers.

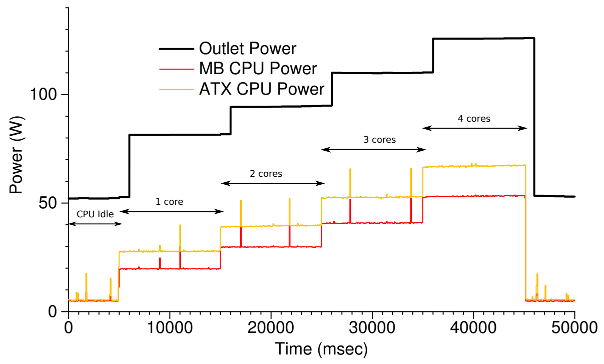


Figure 2.5: *Power Measurement Comparison When Varying the Number of Active Cores*

gives 1000 Sa/s. We use a CPU-bound test workload consisting of a 32×32 matrix multiplication in an infinite loop.

Coarse-grain Power Variance. Fig. 2.5 shows power measurement results across the three different points as we vary the number of active cores. Steps in the power consumption are captured by all measurement setups. The low sampling frequency of the wall-socket power meter prevents it from capturing short and sharp peaks in power (probably caused by background OS activity). The power consumption changes we observe at the wall outlet are at least 13 watts from one activity level to another. At such coarse-grained power variation, the power readings at wall outlet are strongly correlated with power readings at the ATX power rails and CPU voltage regulator.

Fine-grain Power Variance. Fig. 2.6 depicts measurement results when we vary CPU frequency every five seconds from 2.93 GHz to 1.33 GHz in steps of 0.133 GHz. The power measurement setup at the ATX power rails and the CPU voltage regulator capture the changes in power consumption accurately, and apart from the differences in absolute values and the effects of the CPU voltage regulator efficiency curve, there is not much to differentiate measurements at the two points. However, the power measurements taken by the power meter at the wall outlet fail to capture the changes faithfully, even though its one-second sampling rate is enough to capture steps that last five seconds. This effect is even more visible when we introduce throttling (at eight different levels for each CPU frequency), as shown in Fig. 2.7. Here, each combination of CPU frequency and throttling level lasts for two seconds, which should be long enough for the power meter to capture steps in the power consumption. But the power meter performs worse as power consumption decreases. We suspect that this effect is due to the AC to DC conversion circuit of the power supply unit, probably due to the smoothing effect of the capacitor in the PSU. These effects are not visible between measurement points at

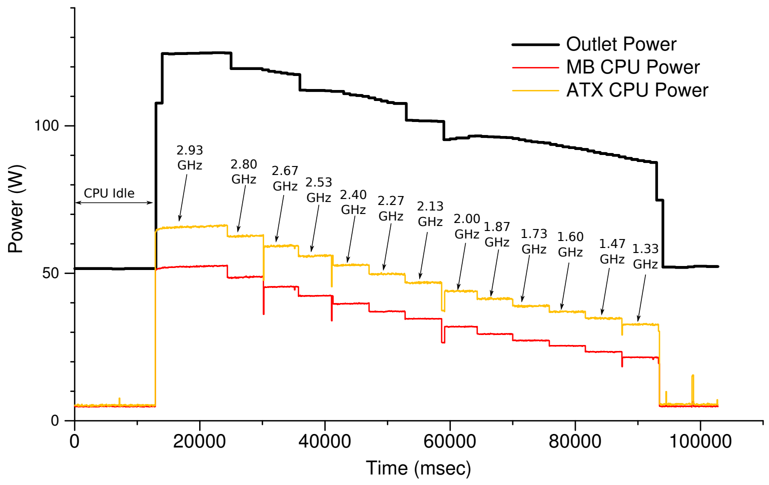


Figure 2.6: *Power Measurement Comparison When Varying Core Frequency*

the ATX power rails and CPU voltage regulator.

Fig. 2.8 shows the efficiency curve of the CPU voltage regulator at various load levels. The voltage regulator on the test system employs dynamic phase control [82] to adjust the number of phases with varying load current to try to optimize the efficiency over a wide range of loads. The voltage regulator switches to one-phase or two-phase operation to increase the efficiency at light loads. When the load increases, the regulator switches to four-phase operation at medium loads and six-phase operation at high loads. The sharp change in efficiency visible in Fig. 2.8 is likely due to adaptation in phase control. Fig. 2.9 shows the obtained efficiency curve of the cabinet PSU against total power consumption calculated on the ATX power rails. The total system power never goes below 30W, and the efficiency of the PSU varies from 60% to around 80% in the output power range from 30W to 100W.

Fig. 2.10 shows the changes in CPU and main memory power consumption while running *gcc* from SPEC CPU2006. Power consumption of the main memory varies from around 7.5W to 22W across various phases of the *gcc* run. Researchers and practitioners who wish to assess main memory power consumption will at least want to measure power at the ATX power rails.

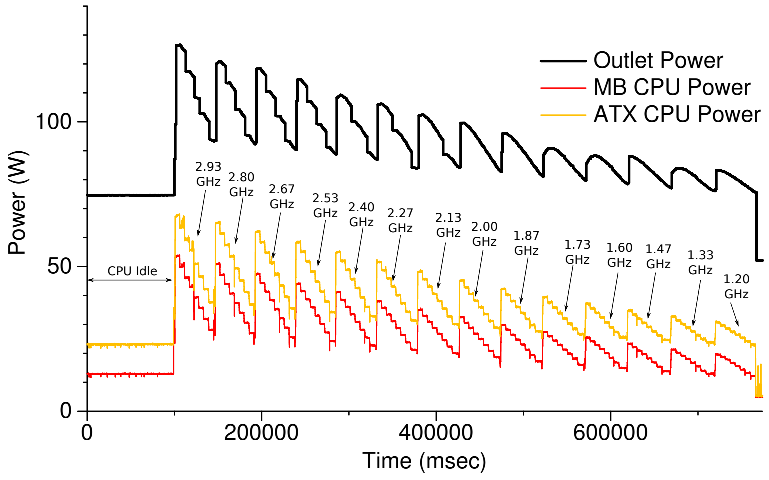


Figure 2.7: *Power Measurement Comparison When Varying Core Frequency Together with Throttling Level*

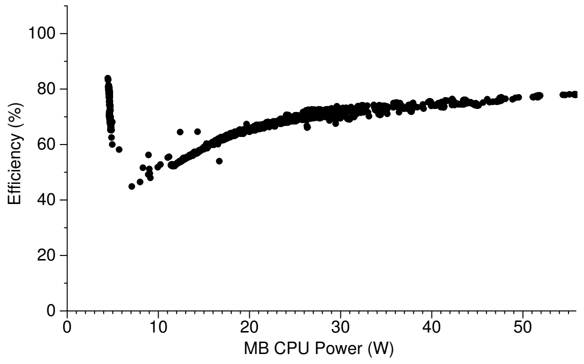


Figure 2.8: *Efficiency Curve of CPU Voltage Regulator*

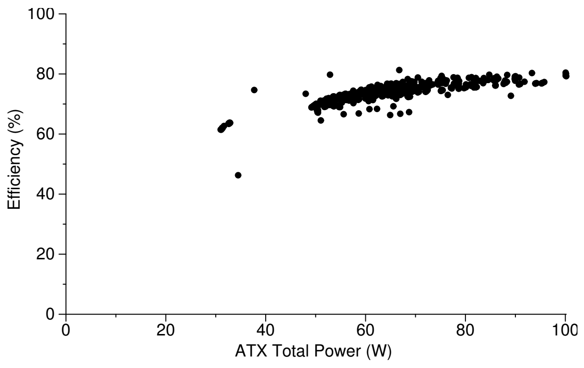


Figure 2.9: *Efficiency Curve of the PSU*

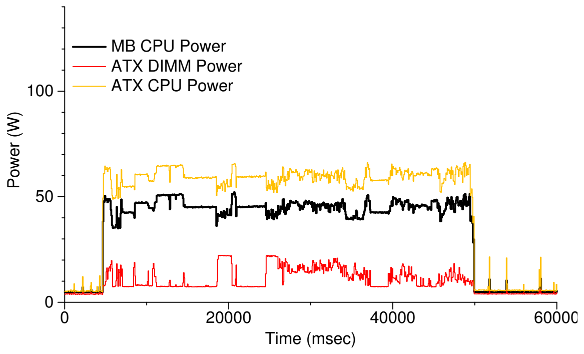


Figure 2.10: *Power Measurement Comparison for the CPU and DIMM (Running gcc)*

2.2 RAPL power estimations

2.2.1 Overview

Intel introduced Running Average Power Limit (RAPL) interface on the Sandy Bridge microarchitecture. The programmers can use the RAPL interface for enforcing power consumption constraint on the processor. This interface includes non-architectural Model-specific registers (MSRs) for setting the power limit and reading the energy consumption status of the processor power domains like processor die and DRAM[2]. The energy expenditure information provided by the initial implementations of RAPL interface was not based on actual current measurement from physical sensors, but a power model based on performance events and probably other inputs like temperature and supply voltage [44]. The later RAPL implementations, especially in server grade desktops use actual measurement to report energy expenditure [83]. However, this distinction between modeling and measurement for RAPL is not publicly available for each individual processor model. In this section, we test the accuracy of RAPL values without assuming one way or another. In the following sections, we compare power measurement readings from RAPL and the ATX power rails.

2.2.2 Experimental Results

We use Intel Core$^{\text{TM}}$i7 4770 processor to compare power measurement results at the ATX power rails and Intel's RAPL energy counter. For reading the RAPL energy counter, we use the x86-MSR driver to read the RAPL energy counter MSR called `MSR_PKG_ENERGY_STATUS`. We set up a real-time timer that raises `SIGALRM` at configured intervals to read the MSR. We divide the RAPL energy values by the sampling interval in seconds to report average power consumption over the sampling interval. The ATX power measurement infrastructure is the same as described for Intel Core$^{\text{TM}}$i7 870 processor. Below we describe our experiments to validate the RAPL energy counter readings.

RAPL Overhead. One of the major differences between measuring power using RAPL and other techniques shown in Figure 2.1 is that the RAPL measurements are done on the same machine that is being tested. As a result, reading the RAPL energy counter at frequent intervals adds some overhead to the system power. To test this, we run our RAPL counter reading tool at the sampling frequency of 1 Sa/s, 10 Sa/s and 1000 Sa/s and monitor the change in system power consumption on the ATX CPU power rails. The results from this experiment are shown in Fig. 2.11. The spikes in

[2]The DRAM energy status counter is only available on server platforms.

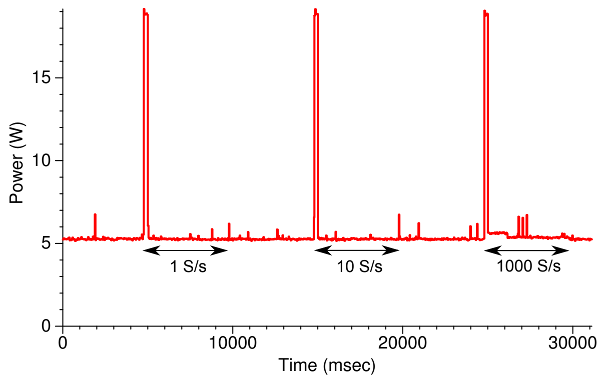


Figure 2.11: *Power Overhead Incurred While Reading the RAPL Energy Counter*

power consumption visible in the figure indicate the starting of the RAPL reading utility. These spikes mainly result from loading dynamic libraries and can be ignored, assuming that the RAPL utility is started before running any benchmark under test. These spikes however act as useful synchronization points to overlap readings from RAPL and the ATX power rails. For calculating the RAPL power overhead, we concentrate on the CPU power consumption after the initial power spike. As seen from the figure, reading the RAPL MSR every second and every 100ms adds no discernible power consumption to the idle system power, but reading the RAPL counter every 1ms adds 0.1W of power overhead. However, this small power overhead comes mainly from generating `SIGALRM` every millisecond and not from reading the MSR. Hence, if the reading of RAPL energy counter is done as part of the existing software infrastructure like a kernel scheduler, this will not add discernible overhead to CPU power consumption.

RAPL Resolution. As per the Intel specification manual [84], the RAPL energy counter is updated at the interval of *approximately* 1ms. To test this update frequency, we run the matrix multiplication application described in Section 2.1.4 while reading the RAPL energy status counter every 1ms. We sample the ATX power rails every 10μS and average over 100 samples. The results from this experiment are shown in Fig. 2.12. Although the RAPL readings follow the same curve as those from the ATX power rails, two things are of note here. First, the RAPL readings show more deviation from the mean than the ATX readings. Second, there are huge deviations from the mean at periodic intervals of around 1.6-1.7 seconds. Hähnel et al. [74] observe that the RAPL counter is not updated exactly at 1ms. As per their experiments, there can be a deviation of a few tens of thousands of cycles in the periodic interval at which the RAPL counter is updated on a given platform. This explains the small deviations from the mean we

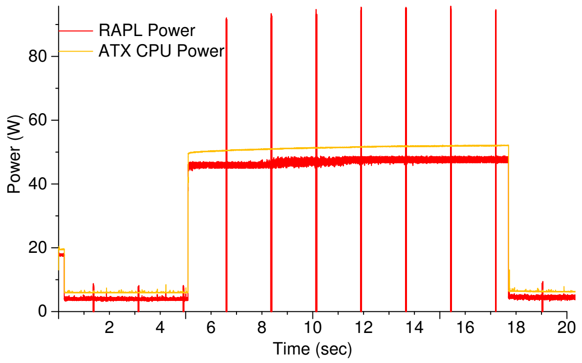


Figure 2.12: *Power Measurement Comparison for the ATX and RAPL at the RAPL Sampling Rate of 1000 Sa/s*

see in our RAPL readings. They also observe that when the CPU switches to System Management Mode (SMM), the RAPL update is delayed until after the CPU comes out of SMM mode. This results in the RAPL energy counter showing no increment between two successive readings, creating a negative deviation. The next update to the counter increments the energy value expended by the processor in last 2ms instead of 1ms, creating a positive deviation. On our system, the CPU switches to SMM every 1.6-1.7 seconds, which causes inaccurate energy readings.

RAPL Sensitivity. We test the sensitivity of the RAPL readings to coarse-grained and fine-grained changes in CPU power consumption. We first repeat the DVFS test from Fig. 2.6 in Section 2.1.4 and take the power measurement readings from the ATX power rail and RAPL. Each DVFS operating point is maintained for five seconds. The ATX power is sampled at 10000 Sa/s while the RAPL counter is sampled at 10 Sa/s. The results from this test are shown in Fig. 2.13. The RAPL measurement curve follows the ATX measurement curve faithfully, although their y-axis scales are necessarily different. This test verifies the RAPL sensitivity to coarse-grain changes in power consumption.

To test the sensitivity of RAPL to fine-grain changes, we use a microbenchmark which writes to a buffer in a loop, and every 20 seconds, the buffer size is increased to increase the cache miss rate. This results in gradual increase in power consumption every 20 seconds. We run this microbenchmark at two fixed frequencies: 3.4 GHz and 800 MHz and compare the power consumption values from ATX CPU rail and RAPL. The results are shown in Figure 2.14 and Figure 2.15 for 3.4 GHz and 800 MHz respectively. At 3.4 GHz, the step increment in cache miss rate results in power consumption increment of 1W–1.2W as measured at ATX CPU power rail. As the results show, RAPL sensitivity is high enough to track power consumption changes in that range. But when

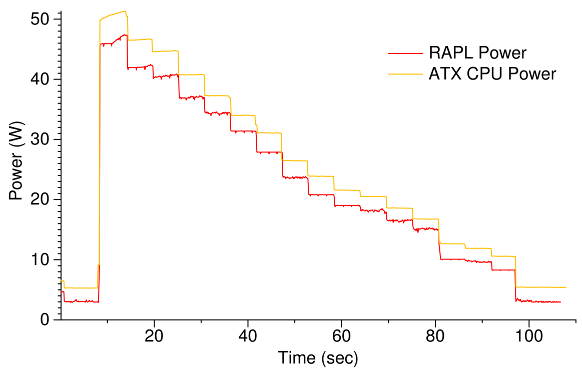


Figure 2.13: *Coarse-grain Sensitivity Test for ATX and RAPL: Frequency Change Test*

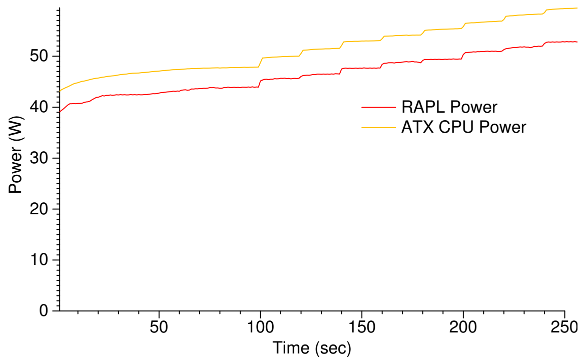


Figure 2.14: *Fine-grain Sensitivity Test for ATX and RAPL at 3.4 GHz*

we run the same experiment at 800 MHz, the power consumption increment is only 0.1W–0.2W. The total power consumption increment at ATX rail from the start to the end of the experiment is 0.99W, which is not detected by RAPL. This highlights the limitation of RAPL when detecting change in power consumption of smaller than 1W.

Next we test the sensitivity of RAPL power measurement to change in package temperature. The static power consumption of the CMOS circuit is dependent on supply voltage and temperature. If the voltage and the core activity is fixed, the change in package temperature should be directly correlated with the change in power consumption. To conduct this experiment, we fix the core and uncore voltage and frequency from BIOS and put all the cores in active C-state (C0: power management state where the CPU is fully turned on). We then use a hot air gun to increase package temperature

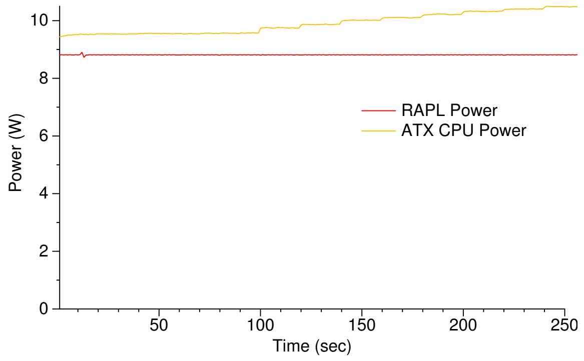


Figure 2.15: *Fine-grain Sensitivity Test for ATX and RAPL at 800 MHz*

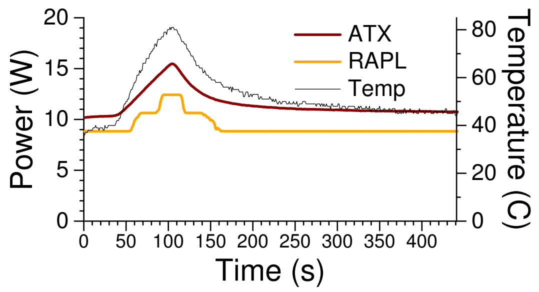


Figure 2.16: *Power Measurement Comparison for ATX and RAPL with Varying Temperature*

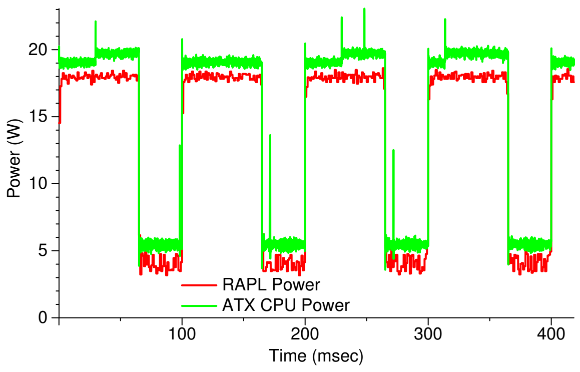
and sample the power measurement values at ATX and RAPL every second. The results from this experiment are depicted in Figure 2.16. As we can see from the results, because of lower sensitivity of RAPL, the correlation between power measurement at ATX and temperature is much stronger than the correlation between RAPL and temperature values. Statistically, the correlation coefficient ρ between ATX power values and temperature was measured to be 0.9857 while that between RAPL and temperature was 0.9127. These results highlight the limitation of RAPL in accurately detecting the change in power consumption due to temperature change.

Next, we test the ability of RAPL to detect changes power consumption at various levels of temporal granularity. We simulate a real-time application in which a task is started at fixed periodic intervals. We compare power measurement from RAPL and ATX while varying the task period. We set up a timer to raise `SIGALRM` signal, and a matrix multiplication task is started at every timer interrupt. The matrix multiplication loop is configured to occupy about 66% of the time period. We run this experiment for three different periods: 100ms, 10ms, and 1ms. We gather samples between two consecutive SMM switches. We read the RAPL counter every 1ms. We sample the ATX power rails every 5μS and average over 20 samples. The results from this experiment are shown in Fig. 2.17. For the 100ms and 10ms period, the RAPL readings closely follow the ATX readings, although they are off by 1ms when the power consumption changes suddenly. As expected, for the 1ms period test, RAPL is unable to capture the power consumption curve but accurately estimates the average power consumption.

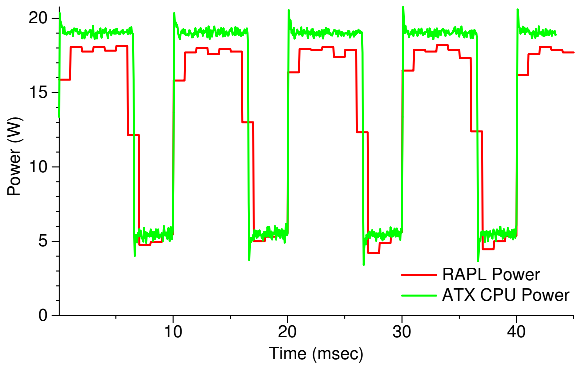
RAPL Accuracy: We perform an experiment to test the ability of RAPL to report accurate power measurement values while running a test benchmark that consists of integer operations, floating-point operations and memory operations in different phases. For this experiment, we sample the RAPL energy counter at 100Sa/S. We sample the ATX CPU power rails at 10000 Sa/s and then average over 100 samples. We also sample the ATX 5V power rails, which on our system mainly supplies the DRAM memory. We run this experiment at 3.4 GHz. The results are shown in Fig. 2.18. The deviations that we see in Fig. 2.12 due to CPU switching to SMM are visible in this Fig. 2.18 as well. Because we sample the RAPL counter every 10ms instead of 1ms, we only see 10% deviation from the mean instead of 100%. Apart from these deviations that occur every 1.7 second (effectively one out of 170 samples), the RAPL energy readings are accurate across the integer, floating-point and memory-intensive phases. To quantify the RAPL power estimation accuracy, we calculate Spearman's rank correlation coefficient for RAPL and ATX CPU readings for the microbenchmark run. The correlation coefficient is ρ=0.9858.

Sample values from the ATX show that DRAM power is significant, which is not captured by the RAPL package energy counter. Power-aware computing methodologies must take this into consideration. The RAPL interfaces on Intel processors for server platforms include an energy counter for the *DRAM domain* called `MSR_DRAM_ENERGY_STATUS`. This can be useful for assessing the power consumption of the DRAM memory.

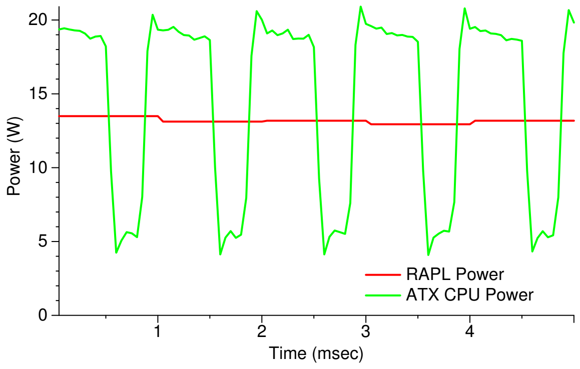
Based upon our experimental results, we conclude that the RAPL energy counter is good for estimating energy over a duration of more than few milliseconds. However, when the researcher is interested in instantaneous power trends, especially peak power trends, sampling the actual power consumption using an accurate measurement infrastructure (like ours) is a better choice.



(a) 100ms period test



(b) 10ms period test



(c) 1ms period test

Figure 2.17: *Power Measurement Comparison for ATX and RAPL with Varying Real-time Application Period*

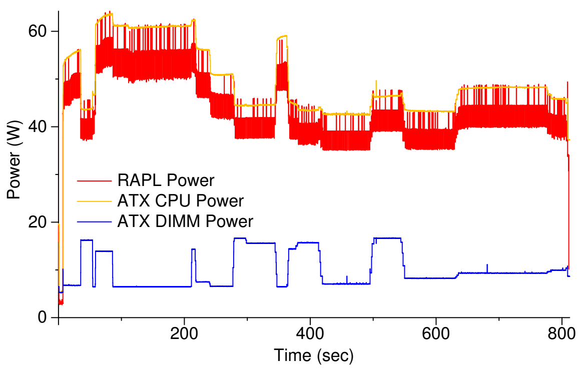


Figure 2.18: *RAPL Accuracy Test for Test Benchmark*

2.3 Related Work

Hackenberg et al. [85] compare several measurement techniques for signal quality, accuracy, timing resolution, and overhead. They use both Intel and AMD systems in their study. Their experimental results complement our results. There have been many interesting studies on power-modeling and power-aware resource management. These employ various means to measure empirical power. Rajamani et al. [34] use on-board sense resistors located between the processor and voltage regulators to measure power consumed by the processor. They use a National Instruments isolation amplifier and data acquisition unit to filter, amplify, and digitize their measurements. Isci and Martonosi [33] measure current on the 12V ATX power lines using clamp ammeters, that are hooked to a digital multimeter (DMM) for data collection. The DMM is connected to a data logging machine via an RS232 serial port. Contreras and Martonosi [51] use jumpers on their Intel XScale^{TM} development board to measure the power consumption of the CPU and memory separately. They feed the measurements to a LeCroy oscilloscope for sampling. Cui et al. [35] also measure the power consumption at the ATX power rails. They use current-sense resistors and amplifiers to generate sense voltages (instead of using current transducers), and they log their measurements using a digital multimeter. Bedard et al. [77] build their own hardware that combines the voltage and current measurements and host interface into one solution. They use an Analog Devices ADM1191 digital power monitor to sense voltage and current values and an Atmel$^{®}$ microcontroller to send the measured values to a host USB port. Hähnel et al. [74] experiment with the update granularity of the RAPL interface and report practical considerations for using the RAPL energy counter for power measurement. We find their results useful in explaining our experiments. Ge et al. [86] propose a framework to provide in-depth analysis of energy expenditure of individual components of the system like fans, disks, processors, DRAM, and motherboard. Their framework uses a combination of hardware resources (sensors, power meter, data acquisition unit) and software (drivers, instrumentation APIs and analysis tools).

2.4 Conclusions

In this chapter, we demonstrate different techniques that can be employed to measure power consumption: using off-the-shelf power meter at wall outlet, using current transducers at ATX power rails and by sampling the current monitor pin of CPU's onboard voltage regulator. We compare the different techniques in terms of accuracy, sensitivity and temporal granularity and discuss their advantages and disadvantages. We conclude that power measurement at ATX power rails provide the best balance between accuracy and accessibility. We test Intel's RAPL energy counter results for overhead, accuracy, and update granularity. We compare ATX power measurements with values reported by RAPL. We conclude that the RAPL package energy counter incurs almost no power overhead and is fairly accurate for estimating energy over periods of more than few tens of milliseconds. However, because of irregularities in the frequency at which the RAPL counter is updated, it shows deviations from the average power when it is sampled at a faster rate.

3 Per-core Power Estimation Model

Power measurement techniques described in Chapter 2 are essential for analyzing the power consumption of systems under test. However, these measurement techniques do not provide detailed information on the power consumption of individual processor cores or microarchitectural units (e.g., caches, floating point units, integer execution units). To develop resource-management mechanisms for an individual processor, system designers need to analyze power consumption at the granularity of processor cores or even the components within a processor core. This information can be provided by placing on-die digital power meters, but that increases the chip's hardware cost. Hence, support for such fine-grained digital power meters is limited by chip manufacturers. Even when measurement facilities exist, their viability for use in OS or user-level power-aware techniques is limited either because the measured values are exposed to the software or the power/energy breakdown is not detailed enough. Indeed, power sensing, actuation, and management support is more often implemented at the blade level with a separate computer monitoring output [87, 88].

An alternative to hardware support for fine-grained digital power meters is to estimate the power consumption at the desired granularity using software power models. Such models can identify various power-relevant events in the targeted microarchitecture and then track those events to generate a representative power consumption value. We can characterize software power estimation models by the following attributes:

- **Portability.** The model should be easy to port from one platform to another;
- **Scalability.** The model should be easy to scale across varying number of active cores and across different CPU voltage-frequency points;
- **CPU Usage.** The model's CPU footprint should be negligible, so as not to pollute the power consumption values of the system under test;
- **Accuracy.** The model's estimated values should closely follow the empirically measured power of the device that is modeled;
- **Granularity.** The model should provide power consumption estimates at the granularity desired for the problem description (per core, per microarchitectural module, etc.); and
- **Speed.** The model should supply power estimation values to the software at minimal latency (preferably within microseconds).

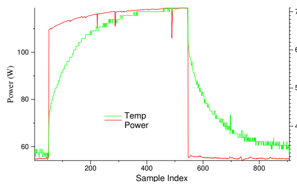
In this thesis, we develop a power model that uses performance counters and temperature to generate accurate per-core power estimates in real time, with no off-line profiling or tracing. We build upon the power model developed by Singh et al. [52, 68] and include some of their data for comparison. We validate their model on AMD K8 and Intel Core$^{\text{TM}}$i7 architectures. Below we explain our choice of using performance counters and temperature to develop the model.

Performance Counters. We use performance counters for model formation because the power models need a mechanism to track CPU activities with low overhead. Most modern processors are equipped with a Performance Monitoring Unit (PMU) providing the ability to count the microarchitectural events of the processor. PMCs are available individually for each core and hence can be used to create core-specific models. This allows programmers to analyze core performance, including the interaction between programs and the microarchitecture, in real time on real hardware, rather than relying on simplified and less accurate performance results from simulations. The PMUs provide a wide variety of performance events. These events can be counted by mapping them to a limited set of Performance Monitoring Counter (PMC) registers. For example, on Intel and AMD platforms, these performance counter registers are accessible as Model Specific Registers (MSRs). Also called Machine Specific Registers, these are not necessarily compatible across processor families. Software can configure the performance counters to select which events to count. The PMCs can be used to count events like cache misses, micro-operations retired, stalls at various stages of an out-of-order pipeline, floating point/memory/branch operations executed, and many more. Although, the counter values are not error-free [89,90] or even deterministic [91], if used correctly, the errors are small enough to make PMCs suitable candidates for estimating power consumption. The number and variety of performance monitoring events available for modern processors are increasing with each new architecture. For example, the number of traceable performance events available in the Intel Core$^{\text{TM}}$i7-870 processor is about ten times the number available in the Intel Core Duo processor [92]. This comprehensive coverage of event information increases the probability that the available performance monitoring events will be representative of overall microarchitectural activity for the purposes of performance and power analysis. This makes the use of performance counters to develop power models for hardware platforms very popular among researchers.

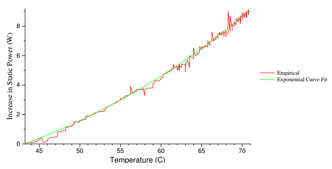
Temperature. Processor power consumption consists of both dynamic and static elements. Among these, the static power consumption is dependent on the core temperature. Eq. 3.1 shows that the static power consumption of a processor is a function of both leakage current and supply voltage. The processor leakage current is a summation of subthreshold leakage and gate-oxide leakage current: $I_{leakage} = I_{sub} + I_{ox}$ [93]. The subthreshold leakage current can be derived using Eq. 3.2 [93]. The V_θ component in the equation is thermal voltage, and it increases linearly with temperature. Since V_θ is in the exponents, subthreshold leakage current has an exponential dependence on temperature. With the increase in processor power consumption, processor temperature increases. This increase in temperature increases leakage current, which, in turn, increases static power consumption. To study the effects of temperature on power consumption, we ran a multithreaded program executing *MOV* operations in a loop on our Intel Core$^{\text{TM}}$i7-870 machine. The rate of instructions retired, instructions executed, pipeline stalls and memory operations remains constant over the entire run of the program. We also keep the CPU operating frequency constant and observe that CPU voltage remains constant as well. This indicates that the dynamic power consumption of the processor does not change over the run of the program. Fig. 3.1(a) shows that the total power consumption of the machine increases during the program's runtime, and it coincides with the increase in chip temperature. The total power consumption increases by almost 10%. To account for this increase in static power, it is necessary to include the temperature in power models.

$$P_{static} = \sum I_{leakage} * V_{core} \tag{3.1}$$

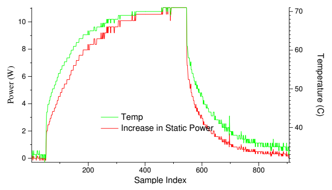
$$I_{sub} = K_1 W e^{-V_{th}/nV_\theta}(1 - e^{-V/V_\theta}) \tag{3.2}$$



(a) Temperature versus Power on Intel Core^{TM}i7-870



(b) Static Power Curve



(c) Temperature versus Static Power on Intel Core^{TM}i7-870

Figure 3.1: *Temperature Effects on Power Consumption*

where $I_{sub} = Subthreshold\ leakage\ current$

$W = Gate\ width$

$V_\theta = Thermal\ voltage$

$V_{th} = Threshold\ voltage$

$V = Supply\ voltage$

$K_1 = Experimentally\ derived\ constant$

$W = Experimentally\ derived\ constant$

As per Eq. 3.1, the static power consumption increases exponentially with temperature. We confirm this empirically by plotting the net increase in power consumption when the program executes at the higher temperature, as shown in the Fig. 3.1(b). The non-regression analysis gives us Eq. 3.3 and the curve fit shown in the Fig. 3.1(b) closely follows the empirical data points with determination coefficient $R^2 = 0.995$.

$$P_{staticInc} = 1.4356 \times 1.034^T, \quad \text{when } V_{core} = 1.09V \tag{3.3}$$

Plotting these estimates of the increments in static power consumption, as in Fig. 3.1(c), explains the gradual rise in total power consumption when the dynamic behavior of a program remains constant. Instead of using a non-linear function, we approximate the static power increase as a linear function of temperature. This is a fair approximation considering that the non-linear equation given in Eq. 3.3 can be closely approximated with the linear equation given in Eq. 3.4 with determination coefficient $R^2 = 0.989$ (for the range in which die temperature changes occur). This linear approximation avoids the added cost of introducing an exponential term in the power model computation.

$$P_{staticInc} = 0.359 \times T - 16.566, \quad \text{when } V_{core} = 1.09V \tag{3.4}$$

Modern processors allow programmers to read temperature information for each core from on-die thermal diodes. For example, Intel platforms report relative core temperatures on-die via Digital Thermal Sensors (DTS), which can be read by software through the MSRs or the Platform Environment Control Interface (PECI) [94]. This data is used by the system to regulate CPU fan speed or to throttle the processor in case of overheating. Third party tools like *RealTemp* and *CoreTemp* on Windows and open-source software like *lm-sensors* on Linux can be used to read data from the thermal sensors. As Intel documents indicate [94], the accuracy of temperature readings provided by the thermal sensors varies, and the values reported may not always match the actual core temperatures. Because of factory variation and individual DTS calibration, reading accuracy varies from chip to chip. The DTS equipment also suffers from slope errors, which means that temperature readings are more accurate near the T-junction max (the maximum temperature that cores can reach before thermal throttling is activated) than at lower temperatures. DTS circuits are designed to be read over reasonable operating temperature ranges, and the readings may not show lower values than 20°C even if the actual core temperature is lower. Since DTS is primarily created as a thermal protection mechanism, reasonable accuracy at high temperatures is acceptable. This affects the accuracy of power models that use core temperature. Researchers and practitioners should read the processor model datasheet, design guidelines, and errata to understand the limitations of their respective thermal monitoring circuits, and they should take corrective measures when deriving their power models, if required.

3.1 Modeling Approach

Our approach to power modeling is workload-independent and does not require application modification. To show the effectiveness of our models, we perform two types of studies:

- We demonstrate accuracy and portability on five CMP platforms;

- We use the model in a power-aware scheduler to maintain a power budget

Studying sources of model error highlights the need for better hardware support for power-aware resource management, such as fine-grained power sensors across the chip and more accurate temperature information. Our approach nonetheless shows promise for balancing performance, power, and thermal requirements for platforms from embedded real-time systems to consolidated data centers, and even to supercomputers.

In the rest of the chapter, we first present the details of how we build our model, and then we discuss how we validate them. Our evaluation analyzes the model's computation overhead(Section 3.3.1) and accuracy(Section 3.3.2). We then explain the meta-scheduler that we use as a proof-of-concept for showing the effectiveness of our model in Section 3.4.

3.2 Methodology

3.2.1 Counter Selection

Selecting appropriate PMCs to use is extremely important with respect to accuracy of the power model. We choose counters that are most highly correlated with measured power consumption. The chosen counters must also cover a sufficiently large set of events to ensure that they capture general application activity. If the chosen counters do not meet these criteria, the model will be prone to error. The problem of choosing appropriate counters for power modeling has been handled in different ways by previous researchers.

Research studies that estimate power for an entire core or a processor [34, 42, 51], use a small number of PMCs. Those that aim to construct decomposed power models to estimate the power consumption of sub-units of a core [48–50] tend to monitor more PMCs. The number of counters needed depends on the model granularity and the acceptable level of complexity. Also, most modern processors allow simultaneous counting of only a limited number (two/four/eight) microarchitectural events. Hence, using more counters in the model requires interleaving the counting of events and extrapolating the counter values over the total sampling period. This reduces the accuracy of absolute counter values but allows researchers to track more counters.

To choose appropriate performance counters for developing our power-estimation model, we divide the available counters into four categories and then choose one counter from each category based upon statistical correlation [52, 68]. This ensures that the chosen counters are comprehensive representations of the entire microarchitecture and are not biased towards any particular section. Caches and floating point units form a large part of the chip real estate, and thus PMCs that keep track of their activity factors would be useful additions to the total power consumption information. Depending on the platform, multiple counters will be available in both these categories. For example, we can count the total number of cache references as well as the number of cache misses for various cache levels. For floating point operations, depending upon the processor model, we can count (separately or in combination) the number of multiplication, addition, and division operations. Because of the deep pipelining of modern processors, we can also expect out-of-order logic to account for a significant amount of power. Stalls due to branch mispredictions or an empty instruction decoder may reduce average power consumption over a fixed period of time. On the other hand, pipeline stalls caused by full reservation stations and reorder buffers will be positively correlated with power because these indicate that the processor has extracted enough instruction-level parallelism to keep the execution units busy. Hence, pipeline stalls indicate not just the power usage of out-of-order logic but of the executions units, as well. In addition, we would like to use a counter that can cover all the microarchitectural components not covered by the above three categories. This includes, for example, integer execution units, branch prediction units, and Single Instruction Multiple Data (SIMD) units. These events can be monitored using the specific PMCs tied to them or by a generalized counter like total instructions/micro-operations (UOPS) retired/executed/issued/decoded. To construct a power model for individual sub-units, we need to identify the respective PMCs that represent each sub-unit's utilization factors.

To choose counters via statistical correlation, we run a training application while sampling the performance counters and collecting empirical power measurement values. Fig. 3.2 shows simplified pseudo-code for the microbenchmark [52] that we use to develop our power model. Here, different phases of the microbenchmark exercise different parts of the microarchitecture. Since the number of relevant PMCs will most likely be more than the limit on simultaneously monitored counters, multiple training runs are required to gather data for all desired counters.

Once the performance counter values and the respective empirical power consumption values are collected, we use a statistical correlation to establish the correlation between performance events (counter values normalized to the number of instructions executed) and power. This guides our selection of the most suitable events for use in the power model. Obviously, the type of correlation method used can affect the model accuracy. We use Spearman's rank correlation [95] to measure the relationship between each counter and power. The performance counters and power values can be linear or non-linear. Using the rank correlation, as opposed to using correlation methods like Pearson's, ensures that this non-linear relationship does not affect the correlation coefficient.

For example, Table 3.1 shows the most power-relevant counters divided categori-

```
for (i=0;i<interval*PHASE_CNT;i++)  {
        phase = (i/interval) % PHASE_CNT;
        switch(phase) {
                 case 0:
                        /* do floating point operations */
                  case 1:
                        /* do integer arithmetic operations */
                  case 2:
                        /* do memory operations with high
                            locality */
                  case 3:
                        /* do memory operations with low locality
                             */
                  case 4:
                        /* do register file operations */
                  case 5:
                        /* do nothing */
                  .
                  .
                  .
        }
}
```

Figure 3.2: *Microbenchmark Pseudo-Code*

(a) FP Operations

Counters	ρ
FP_COMP_OPS_EXE:X87	0.65
FP_COMP_OPS_EXE:SSE_FP	0.04

(b) Total Instructions

Counters	ρ
UOPS_EXECUTED:PORT1	0.84
UOPS_ISSUED:ANY	0.81
UOPS_EXECUTED:PORT015	0.81
INSTRUCTIONS_RETIRED	0.81
UOPS_EXECUTED:PORT0	0.81
UOPS_RETIRED:ANY	0.78

(c) Memory Operations

Counters	ρ
MEM_INST_RETIRED:LOADS	0.81
UOPS_EXECUTED:PORT2_CORE	0.81
UOPS_EXECUTED:PORT234_CORE	0.74
MEM_INST_RETIRED:STORES	0.74
LAST_LEVEL_CACHE_MISSES	0.41
LAST_LEVEL_CACHE_REFERENCES	0.36

(d) Stalls

Counters	ρ
ILD_STALL:ANY	0.45
RESOURCE_STALLS:ANY	0.44
RAT_STALLS:ANY	0.40
UOPS_DECODED:STALL_CYCLES	0.25

Table 3.1: *Intel Core^{TM}i7-870 Counter Correlation*

cally according to the correlation coefficients obtained from running the microbenchmark on the Intel Core$^{\text{TM}}$i7-870 platform. Table 3.1(a) shows that only `FP_COMP_OPS_EXE:X87` is a suitable candidate from the floating point (FP) category. Ideally, to get total FP operations executed, we should count both x87 FP operations(`FP_COMP_OPS_EXE:X87`) and SIMD (`FP_COMP_OPS_EXE:SSE_FP`) operations. The microbenchmark does not use SIMD floating point operations, and hence we see high correlation for the x87 counter but not for the SSE (Streaming SIMD Extensions) counter. Because of the limit on the number of counters that can be sampled simultaneously, we have to choose between the two. Ideally, chip manufacturers would provide a counter reflecting both x87 and SSE FP instructions, obviating the need to choose one. In Table 3.1(b), the correlation values in the total instructions category are almost equal, and thus these counters need further analysis. The same is true for the top three counters in the stalls category, shown in Table 3.1(d). Since we are looking for counters providing insight into out-of-order logic usage, the `RESOURCE_STALLS:ANY` counter is our best option. As for memory operations, choosing either `MEM_INST_RETIRED:LOADS` or `MEM_INST_RETIRED:STORES` will bias the model towards load- or store-intensive applications. Similarly, choosing `UOPS_EXECUTED:PORT1` or `UOPS_EXECUTED:PORT0` in the total instructions category will bias the model towards addition- or multiplication-intensive applications. We therefore omit these counters from further consideration.

Table 3.1 shows that correlation analysis may find counters from the same category with very similar correlation numbers. Our aim is to make a comprehensive power model using only four counters, and thus we must make sure that the counters chosen convey as little redundant information as possible. We therefore analyze the correlation among

(a) MEM versus INSTR Correlation

	UOPS_EXECUTED:PORT234	LAST_LEVEL_CACHE_MISSES
UOPS_ISSUED:ANY	0.97	0.14
UOPS_EXECUTED:PORT015	0.88	0.2
INSTRUCTIONS_RETIRED	0.91	0.12
UOPS_RETIRED:ANY	0.98	0.08

(b) FP versus INSTR Correlation

	FP_COMP_OPS_EXE:X87
UOPS_ISSUED:ANY	0.44
UOPS_EXECUTED:PORT015	0.41
INSTRUCTIONS_RETIRED	0.49
UOPS_RETIRED:ANY	0.43

(c) STALL versus INSTR Correlation

	RESOURCE_STALLS:ANY
UOPS_ISSUED:ANY	0.25
UOPS_EXECUTED:PORT015	0.30
INSTRUCTIONS_RETIRED	0.23
UOPS_RETIRED:ANY	0.21

Table 3.2: *Counter-Counter Correlation*

all the counters. To select a counter from the memory operations category, we analyze the correlation of `UOPS_EXECUTED:PORT234_CORE` and `LAST_LEVEL_CACHE_MISSES` with the counters from the total instructions category, as shown in Table 3.2(a). From this table, it is evident that `UOPS_EXECUTED:PORT234_CORE` is highly correlated with the instructions counters, and hence `LAST_LEVEL_CACHE_MISSES` is the better choice. To choose a counter from the total-instructions category, we analyze the correlation of these counters with the FP and stalls counters (in Table 3.2(b) and Table 3.2(c), respectively). These correlations do not clearly recommend any particular choice. In such cases, we can either choose one counter arbitrarily or choose a counter intuitively. `UOPS_EXECUTED:PORT015` does not cover memory operations that are satisfied by cache accesses, instead of main memory. The `UOPS_RETIRED:ANY` and `INSTRUCTIONS_RETIRED` counters cover only retired instructions and not those that are executed but not retired, e.g., due to branch mispredictions. A `UOPS_EXECUTED:ANY` counter would be appropriate, but since such a counter does not exist, the next best option is `UOPS_ISSUED:ANY`. This counter covers all instructions issued, so it also covers the instructions issued but not executed (and thus not retired).

We use the same methodology to select representative performance counters for all the machines that we evaluate. Table 3.4 shows the counters we select for the different platforms.

3.2.2 Model Formation

Having identified events that contribute significantly to consumed power, we create a formalism to map observed microarchitectural activity and measured temperatures to measured power draw. We re-run the microbenchmark sampling just the chosen PMCs, collecting power and temperature data at each sampling interval. We normalize each time-sampled PMC value, e_i, to the elapsed cycle count to generate an *event rate*, r_i.

We then map rise in core temperature, T, and the observed event rates, r_i, to core power, P_{core}, via a piece-wise model based on multiple linear regression, as in Equation 3.5. We apply non-linear transformations to normalized counter values to account for non-linearity, as in Equation 3.6. The normalization ensures that changing the sampling period of the readings does not affect the weights of the respective predictors. We develop a piecewise power model that achieves better fit by separating the collected samples into two bins based on the values of either the memory counter or the FP counter. Breaking the data using the memory counter value helps in separating memory-bound phases from CPU-bound phases. Using the FP counter instead of the memory counter to divide the data helps in separating FP-intensive phases. The selection of a candidate for breaking the model is machine-specific and depends on what gives a better fit. Regardless, we agree with Singh et. [52, 68] that piecewise linear models better capture processor behavior.

$$\hat{P}_{core} = \begin{cases} F_1(g_1(r_1), \cdots, g_n(r_n), T), & if \text{ condition} \\ F_2(g_1(r_1), \cdots, g_n(r_n), T), & else \end{cases} \tag{3.5}$$

where $r_i = e_i/(cycle\ count), T = T_{current} - T_{idle}$

$$F_n = p_0 + p_1 * g_1(r_1) + ... + p_n * g_n(r_n) + p_{n+1} * T \tag{3.6}$$

As an example, the piecewise linear regression model for the Intel Core^{TM}i7-870 is shown in Eq. 3.7. Here, r_{MEM} refers to the counter `LAST_LEVEL_CACHE_MISSES`, r_{INSTR} refers to the counter `UOPS_ISSUED`, r_{FP} refers to the counter `FP_COMP_OPS_EXE:X87`, and r_{STALL} refers to the counter `RESOURCE_STALLS:ANY`. The piecewise model is broken based on the value of the memory counter. For the first part of the piece-wise model, the coefficient for the memory counter is zero (due to the very low number of memory operations we sampled).

$$\hat{P}_{core} = \begin{cases} 10.9246 + 0 * r_{MEM} + 5.8097 * r_{INSTR} + \\ 0.0529 * r_{FP} + 6.6041 * r_{STALL} + 0.1580 * T, & if\ r_{MEM} < 1e-6 \\ 19.9097 + 556.6985 * r_{MEM} + 1.5040 * r_{INSTR} + \\ 0.1089 * r_{FP} + -2.9897 * r_{STALL} + 0.2802 * T, & if\ r_{MEM} \geq 1e-6 \end{cases} \tag{3.7}$$

3.3 Validation

We evaluate our models by estimating per-core power for the SPEC 2006 [96], SPEC-OMP [97, 98], and NAS [99] benchmark suites. In our experiments, we run multi-threaded benchmarks with one thread per core, and single-threaded benchmarks with an instance on each core. We use gcc 4.2 to compile our benchmarks for a 64-bit architecture with optimization flags enabled, and we run all benchmarks to completion. We use the *pfmon* utility from the *perfmon2* library [100] to access the hardware performance counters. Table 3.3 gives system details of the machines on which we validated our power model. System power is based on measuring the power supply's current draw from the power outlet when the machine is idle. When we cannot find published values for idle processor power, we sum power draw when powered off and power saved by turning off cdrom and hard disk drives, removing all but one memory DIMM, and disconnecting fans. We subtract idle core power from idle system power to get *uncore* (baseline without processor) power. Change in the uncore power itself (due to DRAM or hard drive accesses, for instance) is included in the model estimations. Including temperature as a model input accounts for variation in uncore static power. We always run in the active power state (C0).

We use the *sensors* utility from the *lm-sensors* library to obtain core temperatures, and we use the Watts Up? Pro power meter [75] described in Chapter 2 to gather power data. This meter is accurate to within 0.1W, and it updates once per second. Although measuring the power at wall outlet has limitations in terms of sensitivity to fine-grained changes in power consumption, we use it for our experiments because of its low cost, non-intrusive use, and easy portability. Moreover, we observe that for the benchmarks in our experiments, the power consumption of the system does not change at the granularity of less than one second, and hence sampling power consumption at the rate of 1 sample/second meets our requirement for this modeling approach. Our modeling approach can easily be adapted to other power measurement techniques, such as those described in Chapter 2.

We incorporate the power model into a proof-of-concept, power-aware resource manager (a user-level meta-scheduler of Singh et al. [52, 68]) designed to maintain a specified power envelope. The meta-scheduler manages processes non-invasively, requiring no modifications to the applications or OS. It does so by suspending/resuming processes and, where supported, applying dynamic voltage/frequency scaling (DVFS) to alter core clock rates. For these experiments, we degrade the system power envelope by 10%, 20%, and 30% for Core$^{\text{TM}}$i7-870 platform. Lower envelopes render cores inactive, and thus we do not consider them. Finally, we incorporate the model in a kernel scheduler, implementing a pseudo power sensor per core. The device does not exist

(a) Configuration Parameters for Intel Platforms

	Intel Q6600	Intel Xeon E5430	Intel Core^{TM}i7-870
Cores/Chips	4, dual dual-core	4	
Frequency (GHz)	2.4	2.0, 2.66	2.93
Process (nm)	65	45	45
L1 Instruction	32 KB 8-way	32 KB 8-way	32 KB 4-way
L1 Data	32 KB 8-way	32 KB 8-way	32 KB 8-way
L2 Cache	4 MB 16-way shared	6 MB 16-way shared	256 KB 8-way exclusive
L3 Cache	N/A	N/A	8 MB 16-way shared
Memory Controller	off-chip, 2 channel	off-chip, 4 channel	on-chip, 2 channel
Main Memory	4 GB DDR2-800	8 GB DDR2-800	16 GB DDR3-1333
Bus (MHz)	1066	1333	1333
Max TDP (W)	105	80	95
Linux Kernel	2.6.27	2.6.27	2.6.31
Idle System Power (W)	141.0	180.0	54.0
Idle Processor Power (W)	38.0	27.0	10.0
Idle Temperature ($^{\circ}$C)	36	45	30

(b) Configuration Parameters for AMD Platforms

	AMD Phenom^{TM}9500	AMD Opteron^{TM}8212
Cores/Chips	4	8, quad dual-core
Frequency (GHz)	1.1, 2.2	2.0
Process (nm)	65	90
L1 Instruction	64 KB 2-way	64 KB 2-way
L1 Data	64 KB 2-way	64 KB 2-way
L2 Cache	512 KB 8-way exclusive	1024 KB 16-way exclusive
L3 Cache	2 MB 32-way shared	N/A
Memory Controller	on-chip, 2 channel	on-chip, 2 channel
Main Memory	4 GB DDR2-800	16 GB DDR2-667
Bus (MHz)	1100, 2200	1000
Max TDP (W)	95	95
Linux Kernel	2.6.25	2.6.31
Idle System Power (W)	84.1	302.6
Idle Processor Power (W)	20.1	53.6W
Idle Temperature ($^{\circ}$C)	36	33

Table 3.3: *Machine Configuration Parameters*

(a) PMCs Selected for Intel Platforms

Category	Intel Q6600	Intel E5430	Intel Core^{TM}i7-870
Memory	L2_LINES_IN	LAST_LEVEL_CACHE_MISSES	LAST_LEVEL_CACHE_MISSES
Instructions Executed	UOPS_RETIRED	UOPS_RETIRED	UOPS_ISSUED
Floating Point	X87_OPS_RETIRED	X87_OPS_RETIRED	FP_COMP_OPS_EXE:X87
Stalls	RESOURCE_STALLS	RESOURCE_STALLS	RESOURCE_STALLS:ANY

(b) PMCs Selected for AMD Platforms

Category	AMD Phenom^{TM}9500	AMD Opteron^{TM}8212
Memory	L2_CACHE_MISS	DATA_CACHE_ACCESSES
Instructions Executed	RETIRED_UOPS	RETIRED_INSTRUCTIONS
Floating Point	RETIRED_MMX_AND_FP_INSTRUCTIONS	DISPATCHED_FPU:OPS_MULTIPLY
Stalls	DISPATCH_STALLS	DECODER_EMPTY

Table 3.4: *PMCs Selected for Power-Estimation Model*

Benchmark	baseline	model (10msec)	model (100msec)	model (1sec)
ep.A serial	35.68	35.57	36.04	35.59
ep.A OMP	4.84	4.89	4.77	4.74
ep.A MPI	4.77	4.72	4.73	4.75
cg.A serial	5.82	5.83	5.83	5.83
cg.A OMP	1.95	1.95	1.95	1.95
cg.A MPI	2.19	2.20	2.20	2.20
ep.B serial	146.53	146.52	145.52	146.77
ep.B OMP	19.45	19.33	19.35	19.54
ep.B MPI	18.95	19.41	19.12	19.18
cg.B serial	559.58	560.50	560.11	560.10
cg.B OMP	91.29	92.64	96.90	89.92
cg.B MPI	79.11	79.18	79.18	79.05

Table 3.5: *Scheduler Benchmark Times for Sample NAS Applications on the AMD Opteron^{TM}8212 (sec)*

in hardware but is simulated by the power model module. Reads to the pseudo-device retrieve the power estimate computed most recently.

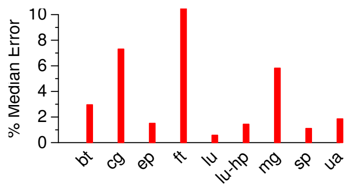
3.3.1 Computation Overhead

If computing the model is expensive, its use becomes limited to coarse timeslices. In this experiment we study the overhead of our power model to evaluate its use in an OS task scheduler. We use the scheduler developed by Boneti et al. [101] that is specifically tailored to High Performance Computing (HPC) applications that require that the OS introduce little or no overhead. In most cases, this scheduler delivers better performance with better predictability, and it reduces OS noise [101, 102]. The model's overhead depends on 1) the frequency with which the per-core power is updated, and 2) the complexity of the operations required to calculate the model. We calculate overhead by measuring execution time for our kernel scheduler running with and without reading the PMCs and temperature sensors. We vary the sample period from one minute to 10 msec. We time applications for five runs and take the average (differences among runs are within normal execution time variation for real systems). Table 3.5 gives measured times for the scheduler with and without evaluating the model. These data show that computing the model adds no measurable overhead, even at 10ms timeslices.

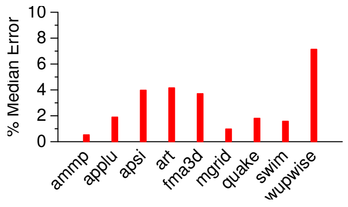
3.3.2 Estimation Error

We assess model error by comparing our system power estimates to power meter output (which our power meter limits to a one-second granularity). We estimate the power consumption per core, and then sum up the power consumption for all cores with uncore power to compare the estimated power consumption with measured values. Figure 3.3 through Figure 3.7 [1] show percentage median error for the NAS, SPEC-OMP, and SPEC

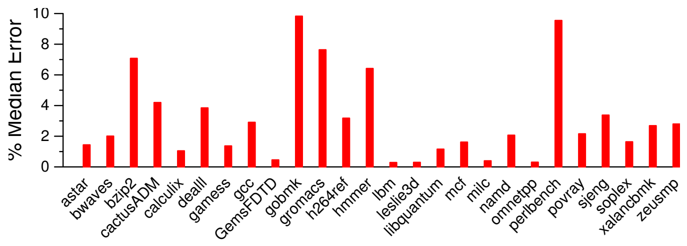
[1] Data for Intel Q6600



(a) NAS



(b) SPEC-OMP



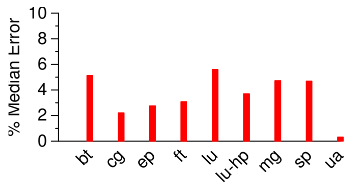
(c) SPEC 2006

Figure 3.3: *Median Estimation Error for the Intel Q6600 system*

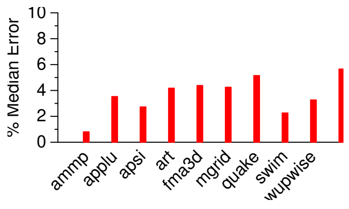
2006 applications on all systems. Figure 3.8 through Figure 3.12 show standard deviation of error for each benchmark suite. The occasional high standard deviations illustrate the main problem with our current infrastructure: instantaneous power measurements once per second do not reflect continuous performance counter activity since the last meter reading.

Estimation error ranges from 0.3% (*leslie3d*) to 7.1% (*bzip2*) for the Intel Q6600 system, from 0.3% (*ua*) to 7.0% (*hmmer*) for the Intel E5430 system, from 1.02% (*bt*) to 9.3% (*xalancbmk*) for the AMD Phenom^{TM}system, from 1.0% (*bt.B*) to 10.7% (*soplex*) for the AMD Opteron^{TM}8212 system and from 0.17% (*art*) to 9.07% (*libquantum*) for the Intel Core^{TM}i7-870 system. On Intel Q6600, only five (out of 45) applications exhibit median error exceeding 5%; on the Intel E5430, only six exhibit error exceeding 5%; on the AMD Phenom^{TM}, eighteen exhibit error exceeding 5%; and on the AMD Opteron^{TM}8212, thirteen exhibit error exceeding 5%. For the Intel Core^{TM}i7-870, median estimations for only seven applications exceed 5% error. Table 3.6 shows the summary of power estimation errors for our model across all platforms. These data suggest that our model works better on Intel machines compared to AMD machines.

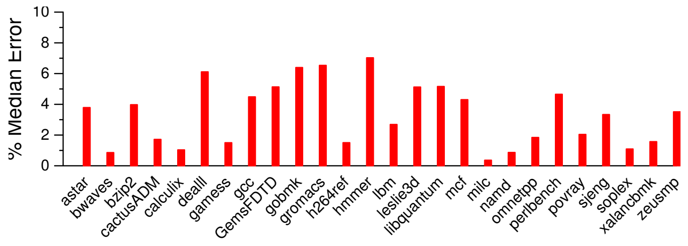
Figure 3.13 shows model coverage via Cumulative Distribution Function (CDF) plots for the suites. On Q6600, 85% of estimates have less than 5% error, and 98% have less than 10%. On E5430, 73% have less than 5% error, and 99% less than 10%. On Phenom^{TM}, 59% have less than 5% error, and 92% less than 10%. On 8212, 37%



(a) NAS

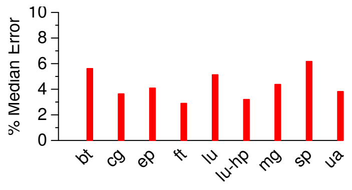


(b) SPEC-OMP

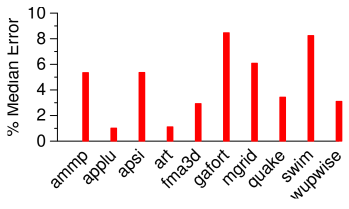


(c) SPEC 2006

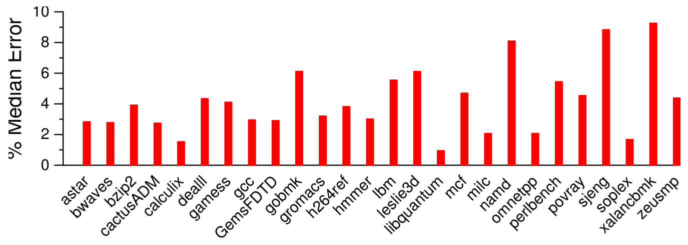
Figure 3.4: *Median Estimation Error for Intel E5430 system*



(a) NAS

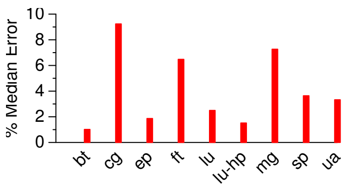


(b) SPEC-OMP

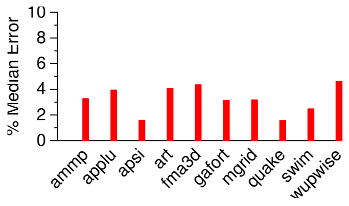


(c) SPEC 2006

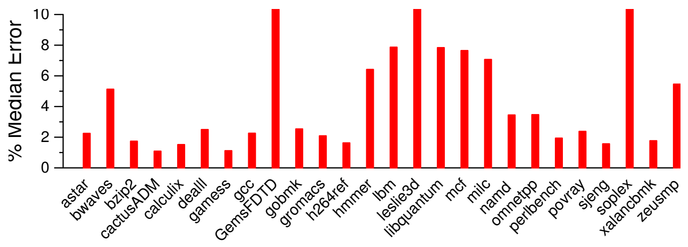
Figure 3.5: *Median Estimation Error for the AMD Phenom^{TM}9500*



(a) NAS

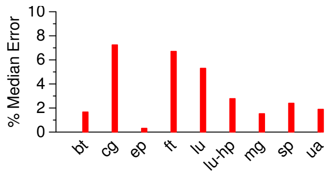


(b) SPEC-OMP

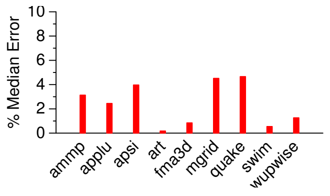


(c) SPEC 2006

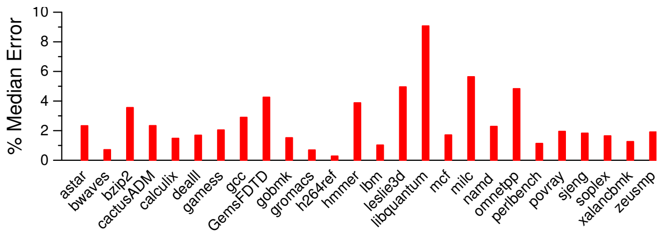
Figure 3.6: *Median Estimation Error for the AMD Opteron*^{TM}*8212*



(a) NAS

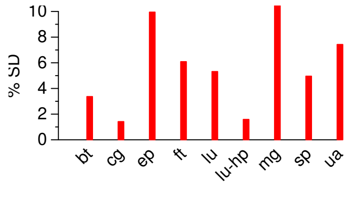


(b) SPEC-OMP

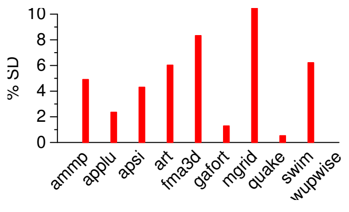


(c) SPEC 2006

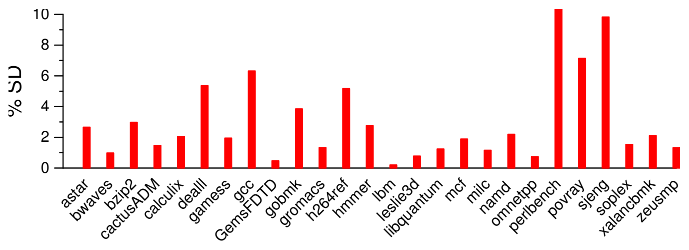
Figure 3.7: *Median Estimation Error for the Intel Core*^{TM}*i7*



(a) NAS

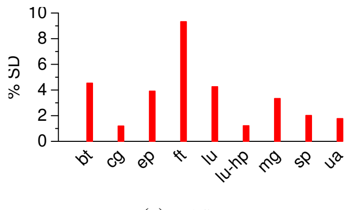


(b) SPEC-OMP

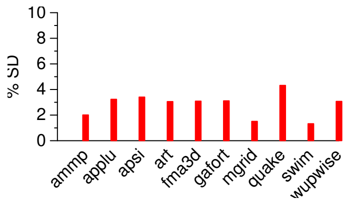


(c) SPEC 2006

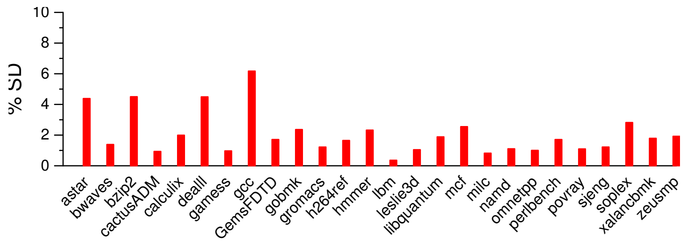
Figure 3.8: *Standard Deviation of Error for the Intel Q6600 system*



(a) NAS

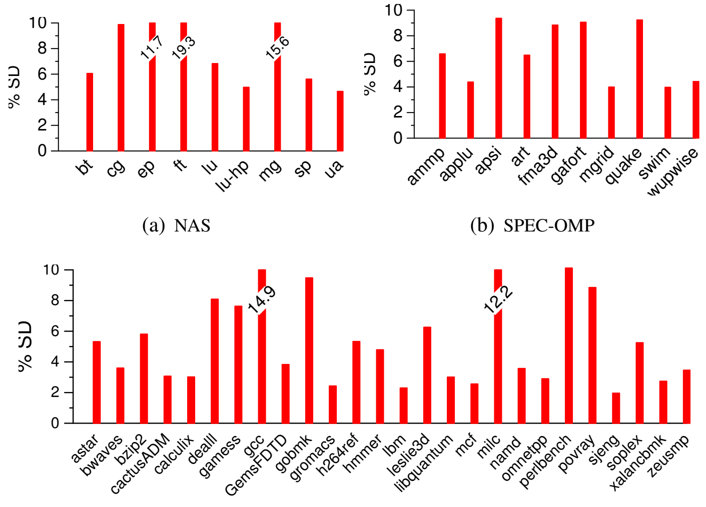


(b) SPEC-OMP



(c) SPEC 2006

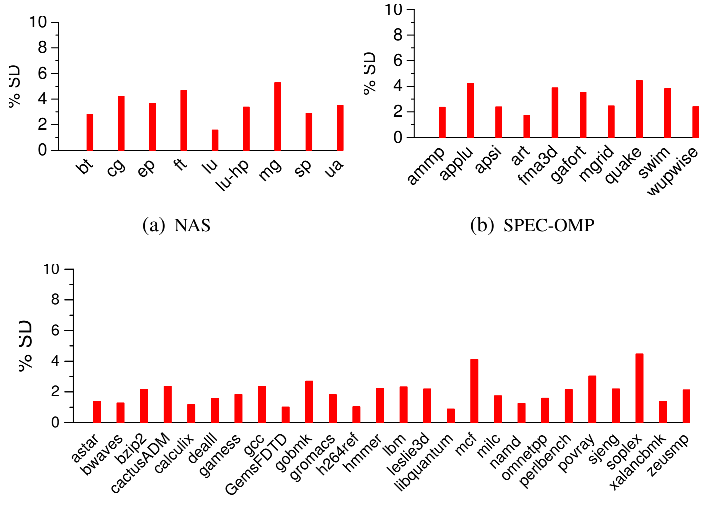
Figure 3.9: *Standard Deviation of Error for Intel E5430 system*



(a) NAS (b) SPEC-OMP

(c) SPEC 2006

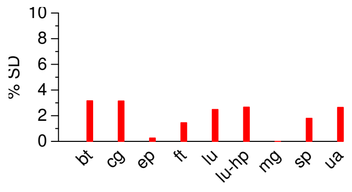
Figure 3.10: *Standard Deviation of Error for the AMD Phenom^{TM}9500*



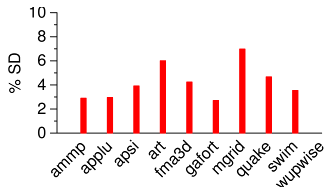
(a) NAS (b) SPEC-OMP

(c) SPEC 2006

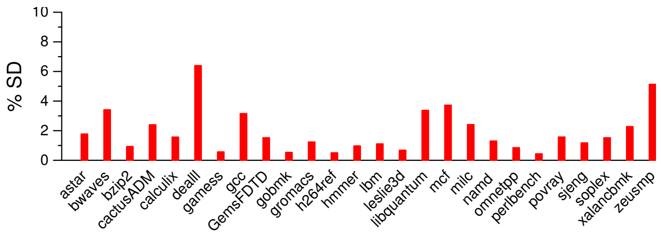
Figure 3.11: *Standard Deviation of Error for the AMD Opteron^{TM}8212*



(a) NAS



(b) SPEC-OMP



(c) SPEC 2006

Figure 3.12: *Standard Deviation of Error for the IntelCore^{TM}i7*

Benchmark	Intel Q660	Intel E5430	Intel Core^{TM}i7-870	AMD Phenom^{TM}	AMD 8212
SPEC 2006	1.1	2.8	2.22	3.5	4.80
NAS	1.6	3.9	3.11	4.5	2.55
SPEC OMP	1.6	3.5	2.02	5.2	3.35
Overall	1.2	3.6	2.07	3.8	4.38

Table 3.6: *Estimation Error Summary*

have less than 5% error, and 76% have less than 10%. For the Core^{TM}i7-870, 82% of estimates have less than 5% error and 96% have less than 10% error. The vast majority of estimates exhibit very small error.

These errors are not excessive, but lower is better. Prediction errors are not clustered, but are spread throughout application execution. Model accuracy depends on the particular PMCs available on a given platform. If available PMCs do not sufficiently represent the microarchitecture, model accuracy will suffer. For example, the AMD Opteron^{TM}8212 processor supports no single counter giving total floating point operations. Instead, separate PMCs track different types of floating point operations. We therefore choose the one most highly correlated with power. Model accuracy would likely improve if a single PMC reflecting all floating point operations were available. For processor models supporting only two Model Specific Registers for reading PMC values, capturing the activity of four counters requires multiplexing the counting of PMC-related events. This means that events are counted for only half a second (or half the total sampling period), and are doubled to estimate the value over the entire period. This approximation can introduce inaccuracies when program behavior is changing rapidly. The model estimation accuracy is also affected by accuracy of hardware sensors. For instance, the Opteron^{TM}8212 temperature sensor data suffers with low accuracy [103].

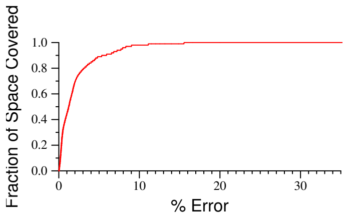
Similarly, even though the microbenchmark tries to cover all scenarios of power consumption, the resulting regression model will represent a generalized case. This is especially true for a model that tries to estimate power for a complex microarchitecture using limited number of counters. For example, floating point operations can consist of add, multiply, or divide operations, all of which use different execution units and hence consume a different amounts of power. If the application being studied uses operations not covered by the training microbenchmark, model accuracy could also suffer.

A model can be no more accurate than the information used to build it. Performance counter implementations display nondeterminism [91] and error [89]. All of these impact model accuracy. Given all these sources of inaccuracy, there seems little need for more complex, sophisticated mathematics when building a model.

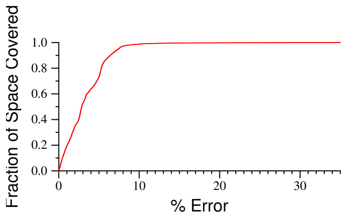
3.4 Management

In the previous section, we discussed our approach to estimating the power consumption of processor resources using power modeling. In this section, we discuss the use of our power model by resource managers that perform task scheduling.

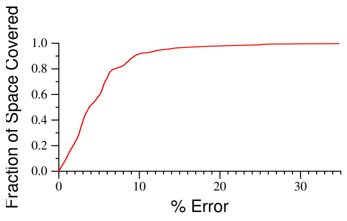
To demonstrate one use of our on-line power model, we experiment with the user-level meta-scheduler from Singh et al. [52, 68]. This live power management application maintains a user-defined system power budget by power-aware scheduling of tasks



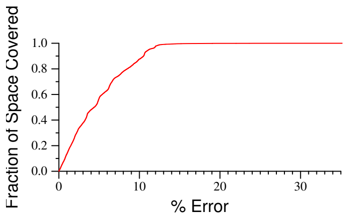
(a) Intel Q6600



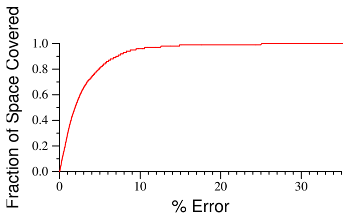
(b) Intel E5430



(c) AMD 9500

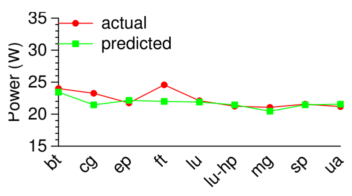


(d) AMD 8212

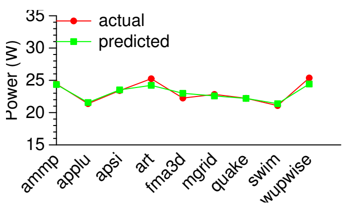


(e) Core^{TM}i7

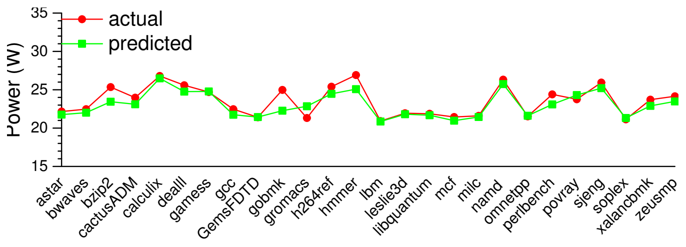
Figure 3.13: *Cumulative Distribution Function (CDF) Plots Showing Fraction of Space Predicted (y axis) under a Given Error (x axis) for Each System*



(a) NAS

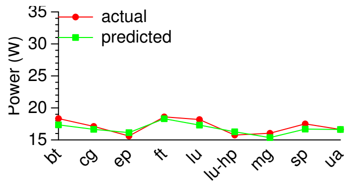


(b) SPEC-OMP

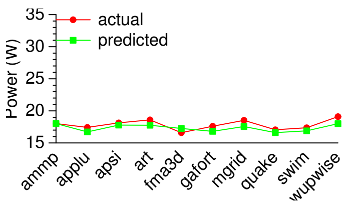


(c) SPEC 2006

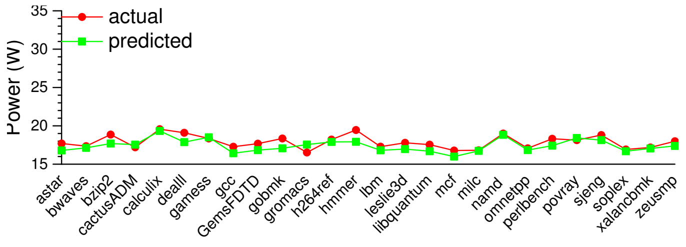
Figure 3.14: *Estimated versus Measured Error for the Intel Q6600 system*



(a) NAS

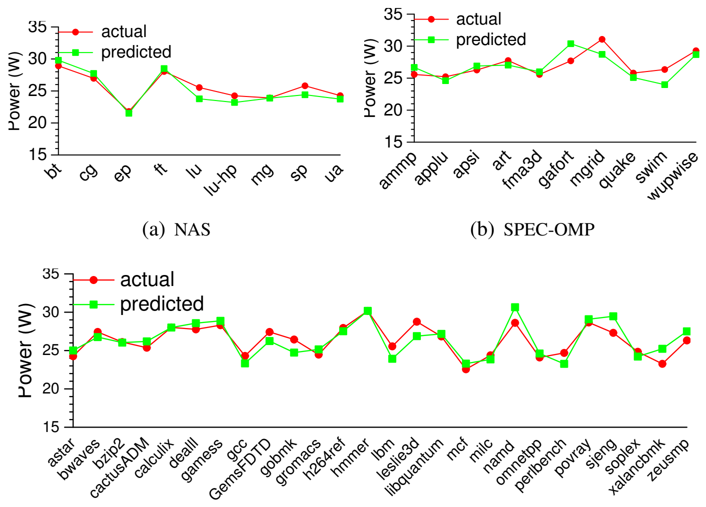


(b) SPEC-OMP



(c) SPEC 2006

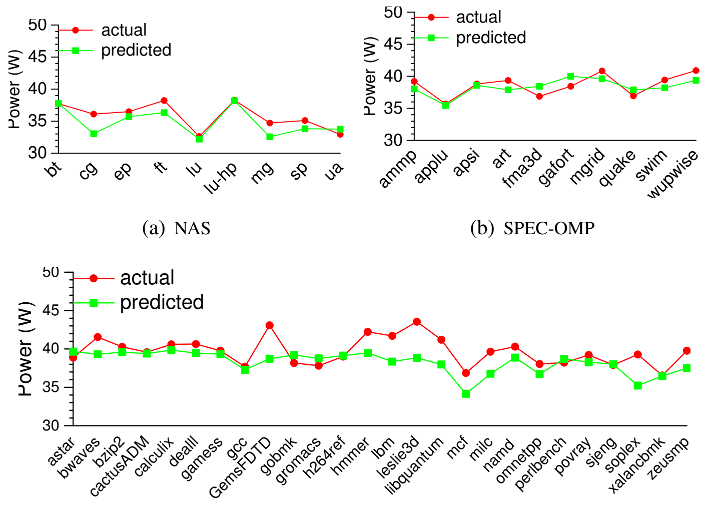
Figure 3.15: *Estimated versus Measured Error for Intel E5430 system*



(a) NAS (b) SPEC-OMP

(c) SPEC 2006

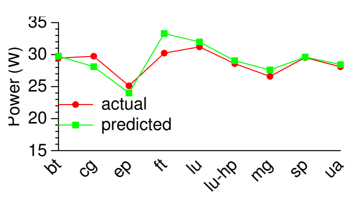
Figure 3.16: *Estimated versus Measured Error for the AMD Phenom^{TM}9500*



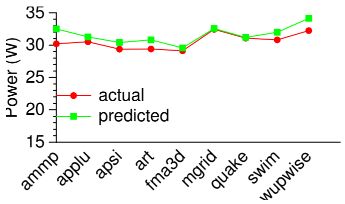
(a) NAS (b) SPEC-OMP

(c) SPEC 2006

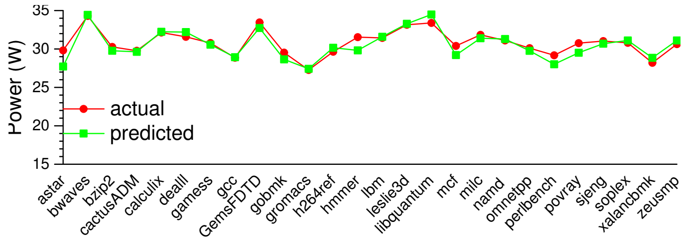
Figure 3.17: *Estimated versus Measured Error for the AMD Opteron^{TM}8212*



(a) NAS



(b) SPEC-OMP



(c) SPEC 2006

Figure 3.18: *Estimated versus Measured Error for the Intel Core^{TM}i7-870*

and/or by using DVFS. We use the power model to compute core power consumption dynamically. The application spawns one process per core. The scheduler reads PMC values via *pfmon* and feeds the sampled PMC values to the power model to estimate core power consumption.

The meta-scheduler binds the affinity of the processes to a particular core to simplify task management and power estimation. It dynamically calculates power values at a set interval (one second, in our case) and compares the system-power envelope with the sum of power for all cores together with the uncore power.

When the scheduler detects a breach in the power envelope, it takes steps to control the power consumption. The scheduler employs two knobs to control system-power consumption: dynamic voltage-frequency scaling as a fine knob, and process suspension as a coarse knob. When the envelope is breached, the scheduler first tries to lower the power consumption by scaling down the frequency. If power consumption remains too high, the scheduler starts suspending processes to meet the envelope's demands. The scheduler maintains the record of the power being consumed by the process at the time of suspension. When the estimated power consumption is less than the target power envelope, the scheduler checks whether any of the suspended processes can be resumed based on the power they were consuming at the time of suspension. If the gap between the current power consumption and the target power budget is not enough to resume a suspended process, and if the processor is operating at a frequency lower than maximum, the scheduler scales up the voltage-frequency. Fig. 3.19 shows the flow diagram of the meta-scheduler.

3.4.1 Sample Policies

When the scheduler suspends a process, it tries to choose the process that will have the least impact on completion time of all the processes. We explore the use of our power model in a scheduler via two sample policies for process suspension.

The **Throughput** policy targets maximum power efficiency (max instructions/watt) under a given power envelope. When the envelope is breached, the scheduler calculates the ratio of instructions/UOPS retired to the power consumed for each core and suspends the process having committed the fewest instructions per watt of power consumed. When resuming a process, it selects the process (if there is more than one suspended process) that had committed the maximum instructions/watt at the time of suspension. This policy favors processes that are less often stalled while waiting for load operations to complete. This policy thus favors CPU bound applications.

The **Fairness** policy divides the available power budget equally among all processes. When applying this policy, the scheduler maintains a running average of the power consumed by each core. When the scheduler must choose a process for suspension, it chooses the process having consumed the most average power. For resumption, the scheduler chooses the process that had consumed the least average power at the time of suspension. This policy strives to regulate core temperature, since it throttles cores consuming the most average power. Since there is high correlation between core power consumption and core temperature, this makes sure that the core with highest temperature receives time to cool down, while cores with lower temperature continue working. Since memory-bound applications are stalled more often, they consume less average power, and so this policy favors memory-bound applications.

3.4.2 Experimental Setup

For the scheduler experiments, we use an Intel Core^{TM}i7-870 system, using DVFS to vary frequencies between 2.926, 2.66, 2.394, 2.128, 1.862, 1.596 and 1.33 GHz. We form separate power models for different frequencies. The power manager can thus better implement policy decisions by estimating power for each frequency. If all core frequencies have been reduced but the power envelope is still breached, we suspend processes to reduce power. We compare against runtimes with no enforced power envelope. We divide our workloads into three sets based on *CPU intensity*. We define CPU intensity as the ratio of instructions retired to last-level cache misses. The three sets

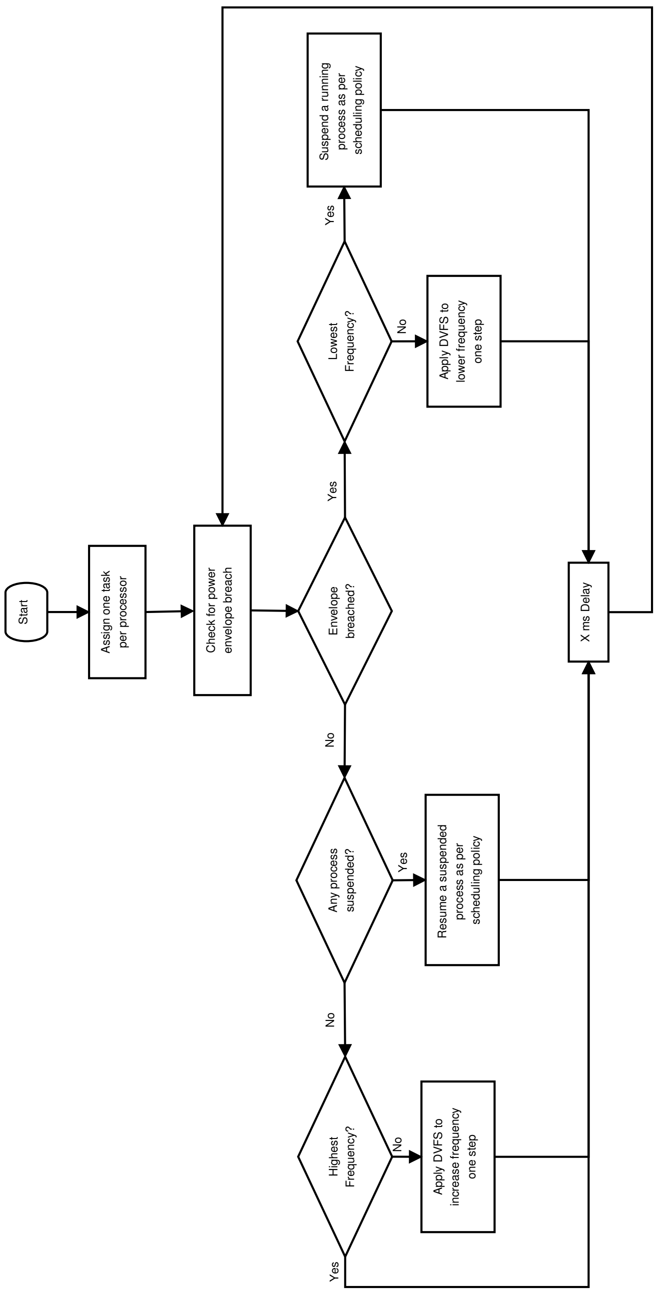
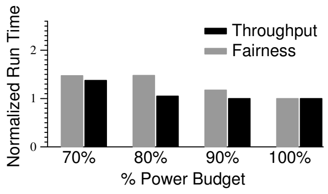

Start
Assign one task per processor
Check for power envelope breach
Envelope breached?
Yes
Lowest Frequency?
Yes
Suspend a running process as per scheduling policy
No
Apply DVFS to lower frequency one step
No
Any process suspended?
Yes
Resume a suspended process as per scheduling policy
No
Highest Frequency?
No
Apply DVFS to increase frequency one step
Yes
X ms Delay

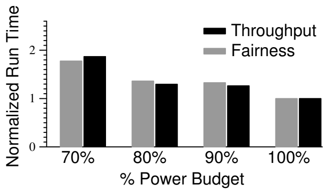
Figure 3.19: *Flow diagram for the Meta-Scheduler*

Benchmark Category	Benchmark Applications	Peak System Power (W)
CPU-Bound	ep, gamess, namd, povray	130
Moderate	art, lu, wupwise, xalancbmk	135
Memory-Bound	astar, mcf, milc, soplex	130

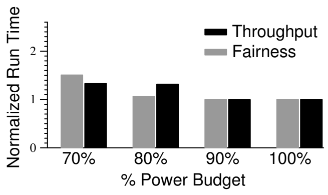
Table 3.7: *Workloads for Scheduler Evaluation*



(a) CPU-bound



(b) Memory-bound



(c) Moderate

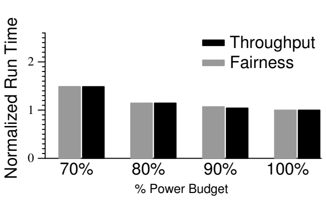
Figure 3.20: *Runtimes for Workloads on the Intel Core^{TM}i7-870 (without DVFS)*

are designated as CPU-Bound, Moderate, and Memory-Bound workloads (in decreasing order of CPU intensity). The workloads categorized in these sets are listed in Table 3.7.

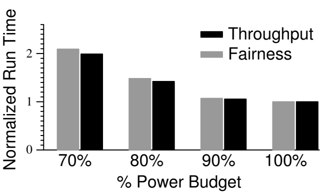
3.4.3 Results

Fig. 3.20 shows the normalized runtimes for all the workloads when only process suspension is used to maintain the power envelope. The Throughput policy should favor CPU-bound workloads, while the Fairness policy should favor memory bound workloads, but this distinction is not always visible. This is because of the differences in the runtimes of the various workloads. This can be seen from the execution times of the CPU-bound benchmarks when using the Throughput policy. The CPU bound applications *ep* and *gamess* have the lowest computational intensities and execution times. As a result, these two applications are suspended most frequently, which does not affect the total execution time, even when the power envelope is set to 80% of peak usage.

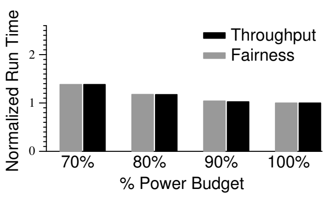
Fig. 3.21 shows the results when the scheduler uses both DVFS and process sus-



(a) CPU-bound



(b) Memory-bound



(c) Moderate

Figure 3.21: *Runtimes for Workloads on the Intel Core^{TM}i7-870 (with DVFS)*

pension to maintain the given power envelope. As noted, the scheduler uses DVFS as a fine knob and process suspension as a coarse knob in maintaining the envelope. The Intel Core^{TM}i7-870 processor that we use for our experiments supports fourteen different voltage-frequency points. These frequency points range from 2.926 GHz to 1.197 GHz. For our experiments, we make models for seven frequency points (2.926, 2.66, 2.394, 2.128, 1.862, 1.596 and 1.33 GHz), and we adjust the processor frequency across these points. Our experimental results show that for CPU-bound and moderate benchmarks, there is little difference in execution time under different suspension policies. This suggests that for these applications, the scheduler hardly needs to suspend the processes and regulating DVFS points proves sufficient to maintain the power envelope. Performance for DVFS degrades compared to cases where no DVFS is used. The explanation for this lies in the difference between runtimes of different applications within the workload sets. When no DVFS is used, all processes run at full speed. And even when one of the processes is suspended, if that process is not critical, it still runs at full speed later in parallel with the critical process. But in the case of DVFS being given higher priority over process suspension, when the envelope is breached, all processes are slowed, and this affects total execution time.

3.5 Related Work

The difficulty of obtaining accurate, real-time power measurements together with the growing emphasis on green computing have sparked much research on power-estimation models. An early effort by Tivari et al. [104] develops a measurement-based approach to model instruction-level power effects. They associate energy cost for each type of instruction, instruction pairs and inter-instruction effects like pipeline stalls and cache misses. They do not distinguish between static and dynamic power components. Their methodology requires forming the models separately at each processor frequency.

Russell and Jacome [105] develop simpler models for two members of the Intel i960® processor family, finding that detailed per-instruction power information is unnecessary for these processors. Their models are based on measured median average power and an estimated power constant.

Like Tivari et al., Cignetti et al. [106] develop models to provide software developers with information on the energy implications of their design decisions. They target PalmOS® devices, choosing device power states as their level of abstraction and assuming that transitions between states result from system calls. They build system models on the measured steady-state power of each power state and the power cost of transitioning between states.

As the availability and accessibility of hardware performance counters on commodity processors increased, more recent power models started utilizing them to build the processor power model.

Joseph and Martonosi [48] model power consumption of individual microarchitectural components by choosing performance counters that intuitively correlate with component utilization and deriving weights for each component's activity factor. They run their experiments at a single frequency (since DVFS was not common in 2001) and make no distinction between static and dynamic power. They cannot estimate power for 24% of the chip, and so they assume peak power for those structures.

Contreras and Martonosi [51] use five PMCs to estimate power for different frequencies on an XScale system (with an in-order uniprocessor). They gather data from multiple benchmark runs because the number of performance events in their model is greater than the number of available counters. They derive power weights for frequency-voltage pairs, and form a parameterized linear model. Their model exhibits low percentage error, like ours, but they test on only seven applications. Unlike our approach, this methodology is not intended for on-line use.

Economou et al. [42] use PMCs to predict power on a blade server. They profile the system using application-independent microbenchmarks. The resulting model estimates power for the CPU, memory, and hard drive with 10% average error. Their model provides power breakdown between system components but not for on-chip components.

Lee and Brooks [54] predict power via statistical inference models. They build correlations based on hardware parameters, using the most significant parameters to train their model. They profile their design space a priori, and then estimate power on random samples. This methodology requires training on the same applications for which they want to estimate power, and so their approach depends on having already sampled all applications of interest.

Merkel and Bellosa [107] use PMCs to estimate power per processor in an eight-way SMP, shuffling processes to reduce overheating of individual processors. They do not state which PMCs they use, nor how they are chosen. Their goal is not to reduce power, but to distribute it. Their estimation method suffers less than 10% error in a linux kernel implementation.

The power model presented in this chapter is based on the approach of Singh et al. [52, 68]; we augment that work by porting and validating the model on many platforms, improving accuracy of the model by augmenting the microbenchmarks and counter selection methodology, analyzing estimation errors, and exploiting multiple DVFS operating points for scheduler validation.

Bertran et al. [50] develop a decomposable model to estimate the power consumed by eight microarchitectural components. Their model relies on 13 performance events. Note that since most processors support counters (two to eight are common), tracking more events than the number of available counters requires time multiplexing among the counters. This reduces observational acuity when sampling, which can contribute to model error. In contrast, both our modeling approaches uses number of performance events that are smaller than or equal to the number of available performance counters.

Spiliopoulos et al. [36] use instructions executed and instructions retired counter to develop a power model. Their model shows high errors for memory-bound benchmarks since they do not consider memory operations in their model.

The use of power-estimation models for power-aware or energy-aware scheduling has been explored by various research works.

Isci et al. [39] analyze global power management policies to enforce a given power budget and to minimize power consumption for the given performance target. They conduct their experiments on the Turandot [108] simulator. They assume the presence of on-core current sensors to acquire core power information, while they use performance counters to gather core performance information. They have developed a global power manager that periodically monitors the power and performance of each core and sets the operating mode (akin to DVFS performance states) of the core for the next monitoring interval. They assume that the power mode can be set independently for each core. They experiment with three policies to evaluate their global power manager. The *Priority* policy assigns different priorities to different cores in a multi-core processor and tries to speed up the core with highest priority while slowing down the lowest priority core when the the power consumption overshoots the assigned budget. The policy called *pullHipushLo* is similar to our Fairness policy from the above case study; it tries to balance the power consumption of each core by slowing down the core with highest power consumption when the power budget is exceeded and speeds up the core with lowest power consumption when there is a power surplus. *MaxBIPS* tries to maximize the system throughput by choosing the combination of power modes on different cores that is predicted to provide maximum overall BIPS (Billion Instructions Per Second).

Rajamani et al. [34] use their power estimation model to drive a power management policy called *PerformanceMaximizer*. For a given power budget, they exploit the DVFS levels of the processor to try to maximize the processor performance. For each performance state (P-state), they apply their power model to estimate power consumption at the current P-state. They use the calculated power value to estimate the power consumption at other P-states by linearly scaling the current power value with frequency. The scheduler increases the performance state to highest level that it estimates would be safely below the power budget value.

Su et al. [109] use their power model to enforce power capping. They only use DVFS to enforce the power budget while we explore the combination of DVFS and task suspension to achieve the power capping requirements.

Banikazemi et al. [38] use a power-aware meta-scheduler. Their meta-scheduler monitors the performance, power and energy of the system by using performance counters and in-built power monitoring hardware. It uses this information to dynamically remap the software threads on multi-core servers for higher performance and lower energy usage. Their framework is flexible enough to substitute the hardware power monitor with a performance-counter based model.

Meng et al. [40] apply a multi-optimization power saving strategy to meet the constraints of a chip-wide power budget on reconfigurable processors. They run a global power manager that configures the CPU frequency and/or cache size of individual cores. They use risk analysis to evaluate the trade-offs between power saving optimizations and potential performance loss. They select the power-saving strategies at design time to create a static pool of candidate optimizations. They make an analytic power and performance model using performance counters and sensors that allows them to quickly evaluate many power modes and enables their power manager to choose a global power mode at periodic intervals that can obey the processor-wide power budget while maximizing the throughput.

Apart from the work done at the level of individual processor, power-aware strategies have also been explored at depth to schedule jobs across multiple nodes in high-performance clusters. Below we discuss few of the related research work.

Kim et al. [41] propose scheduling algorithms for bag-of-tasks applications with the simultaneous aim of meeting deadline constraint and minimizing energy expenditure. They leverage cluster-wide DVFS to implement power-aware scheduling algorithms on their simulator.

Ge et al. [110] propose a power-aware runtime system called *CPU MISER* that implements a system-wide power management using DVFS. Their system uses performance prediction model to achieve a user-specified performance loss constraint while attempting to reduce energy expenditure.

Marathe et al. [111] develop a power-aware runtime system that aims to maximize performance while distributing available power to nodes and cores. Their runtime system dynamically explores optimal thread concurrency levels and DVFS state to enforce a power budget while simultaneously identifying the critical paths that can be prioritized during power budget allocation.

Patki et al. [112] propose a low-overhead resource manager for power-constrained clusters. They aim to maximize system power utilization by using power-aware scheduling algorithms and reduce application turnaround time.

3.6 Conclusions

We derive statistical, piece-wise multiple linear regression power models mapping PMC values and temperature readings to power, and demonstrate their accuracy for the SPEC 2006, SPEC-OMP, and NAS benchmark suites. We write microbenchmarks to exercise the PMCs in order to characterize the machine and run those microbenchmarks with a power meter plugged in to generate data for building the models. For our regression models, we select the PMCs that correlate most strongly with measured power. Because they are built on microbenchmark data, and not actual workload data, the resulting models are application independent. We apply the models to 45 benchmarks (including multithreaded applications) on five CMPs containing dual- or quad-core chips totaling four or eight cores. In spite of our generality, estimation errors are consistently low across five different systems. We observe overall median errors per machine between 1.2% and 4.4%.

We then show the effectiveness of our power model for live power management on an Intel Core$^{\text{TM}}$system with and without DVFS. We suspend and resume processes based on per-core power usage, ensuring that a given power envelope is not breached. We also scale core frequencies to remain under the power envelope.

As numbers of cores and power demands continue to grow, efficient computing requires better methods for managing individual resources. The per-core power estimation methodology presented here extends previously published models in both breadth and depth, and represents a promising tool for helping meet those challenges, both by providing useful information to resource managers and by highlighting opportunities for improving hardware support for energy-aware resource management. Such support is essential for fine-grained, power-aware resource management.

4 Energy Characterization for Multicore Processors Using Dynamic and Static Power Models

The power modeling methodology presented in Chapter 3 is used to estimate power consumption at the granularity of individual cores. This power modeling technique was shown to be fairly accurate and portable across multiple platforms and also effective for implementing power budgeting. But a major limitation of this power model was that it had to be trained separately for each voltage-frequency operating point and hence, involves considerable initial investment when the number of individual voltage-frequency operating points are in double digits like in modern Intel processors. Moreover, this model does not distinguish between core and uncore components of the processor. The uncore power consumption is equally distributed between the power estimation values of individual cores. For the purpose of energy characterization of the microarchitecture, a power model that can provide estimation values at finer granularity would be more useful.

Mair et al. [113] emphasize the importance of distinguishing between static and dynamic power due to the fundamental differences in their impact and their parameters. Both depend on supply voltage, but static power depends on temperature and dynamic power depends on frequency. These power components scale differently from one voltage-frequency step to another, and accurate information on the breakdown between them is critical for understanding energy expenditure trends when applying DVFS.

Another reason to model static and dynamic power separately is to characterize system energy efficiency. Static power consumption is due to leakage current in the transistors, and hence it does not contribute towards performing useful work. The static energy thus represents "wasted energy" in the system. The ratio of static to total system energy represents energy inefficiency of the system and can be useful in making qualitative architectural comparisons.

We present a systematic methodology for deriving models that calculate static and dynamic power consumption for the uncore and the cores. We show how to isolate and quantify power consumption of different processor components. We validate our power models at four different voltage-frequency steps for single-threaded and parallel benchmarks, and show that our models estimate power with good accuracy (mean absolute error of 3.14%) across all benchmarks.

We use our models to characterize the energy efficiency of scaling clock frequency the Haswell processor. We find that uncore static energy represents a significant portion of total system energy (up to 60%), making it a major source of energy inefficiency. We also show that running an application at lower frequencies does not always expend less energy: the degree to which an application is memory-bound must be considered when choosing energy-efficient DVFS policies.

We use the results from energy characterization for frequency scaling to implement a low overhead online DVFS manager and that can switch the system frequency at runtime to achieve energy-efficiency for the currently running application. We show the efficacy of our DVFS implementation by comparing the energy expenditure and EDP values from our DVFS runs with those from the runs at best static frequency (chosen after exhaustic search).

Finally, we use our power models to characterize the energy efficiency of thread scaling on Haswell processor. We identify four orthogonal parameters that affect the performance and energy efficiency of thread scaling namely, serial fraction, core-to-core communication overhead, bandwidth contention and locking mechanism overhead. We leverage work done previously for characterizing TM systems by Hong et al. [114] to create a set of microbenchmarks to vary these four parameters orthogonally and analyze their effects on the energy expenditure while varying the number of parallel threads. We make different insightful observations and show the level of concurrency at which lowest energy expenditure is achieved varies based upon the value of identified parameters.

4.1 Power Modeling Methodology

We run experiments on an Intel Core™ i7 4770 (Haswell) processor that has four physical cores with a maximum clock frequency of 3.4 GHz. Each core has two eight-way 32 KB private L1 caches (separate I and D), a 256 KB private L2 cache (combined I and D), and an 8 MB shared L3 cache, with 16 GB of physical memory on board.

Since we want to focus on the CPU portion of the chip, our experiments do not use the on-chip GPU and eDRAM (which are power-gated and cannot affect results). Power models for these components would be interesting and useful, but deriving them is beyond the scope of the work presented here. We disable hyper-threading and Turbo Boost for all experiments. In the rest of our discussion, we use chip and CPU interchangeably.

We measure power consumption at the ATX CPU power rails using infrastructure that we built for previous work [115]. This infrastructure uses current transducers to convert measured current to voltage, which is logged using a data acquisition unit.

We collect performance counter values using the `pfmon` tool provided by the `libpfm4` library. We use the Linux® kernel driver `coretemp` to read package temperatures via the on-die Digital Temperature Sensor (DTS) [94].

Our models break total power into four components:

1. uncore dynamic power,
2. core dynamic power,
3. uncore static power, and
4. core static power.

We strive to develop models for each component in isolation in order to prevent model estimation error of any component from affecting the estimation accuracy of other components.

4.1.1 Uncore Dynamic Power Model

We break the uncore dynamic power into uncore idle dynamic power and uncore active dynamic power. Our experiments indicate that the uncore is neither clock-gated nor power-gated when idle. Eq. 4.1 gives the formula for idle uncore dynamic power consumption.

$$P(uncore)_{idle\_dynamic} = \alpha \times C \times V^2 \times F \tag{4.1}$$

where $\alpha =$ idle uncore activity factor

$C =$ uncore capacitance

$V =$ uncore supply voltage

$F =$ uncore frequency

When idle, the uncore effective capacitance (αC in Eq. 4.1) remains virtually constant across frequency changes (we disregard the insignificant uncore activity caused by OS housekeeping threads). Recall that uncore static power depends on uncore voltage and package temperature. We fix uncore voltage from the BIOS and measure idle chip power at the same package temperature at different uncore frequencies. We can thus safely assume that static power remains constant across frequency changes. Any differences in measured power must therefore be due to changes in uncore dynamic power. We measure idle CPU power consumption at frequencies F_1 and F_2 while making sure that the package temperature is same across two readings. Then, the idle CPU power at uncore frequency F_n can be calculated as $P_n = \alpha \times C \times V^2 \times F_n + P_{static}$. We then calculate αC for the idle uncore using Eq. 4.2.

$$\alpha \times C = \frac{P_2 - P_1}{V^2 F_2 - V^2 F_1} \tag{4.2}$$

Eq. 4.3 shows the model we generate for uncore idle dynamic power. We verify the value of αC by repeating our measurements for different values of F_1 and F_2.

$$P(uncore)_{idle\_dynamic} = 1.41 \times 10^{-9} \times V^2 \times F \tag{4.3}$$

where $V =$ uncore supply voltage

$F =$ uncore frequency

We next analyze the effects of L2 cache misses on uncore dynamic power consumption. An L2 miss causes activity on the ring interconnect and in the L3 last-level cache. To measure the effects of L2 misses on CPU power consumption, we create a microbenchmark that interleaves memory accesses with enough integer and floating point operations to hide L3 latency. We measure CPU power consumption as we gradually increase the benchmark's working-set size such that core activity (micro-operations executed per cycle) and L2 activity (L2 requests per cycle) remain constant but the ratio of L2 misses to L2 requests increases. We attribute differences in consumed power to increases in uncore activity from the increased L2 misses. We run regression analysis on the L2 miss rate and increase in power consumption to generate a quadratic model for the L2-miss contribution to uncore dynamic power. Eq. 4.4 shows this model, and Figure 4.1 shows its accuracy with respect to our measured values. The R^2 coefficient measuring the goodness of the fit for the estimation model is 0.999.

$$\begin{aligned} P(uncore)_{L2\_miss\_dynamic} = {} & (3.043 \times 10^{-9} \times x^2 \\ & + 2.881 \times 10^{-9} \times x) \\ & \times V^2 \times F \end{aligned} \tag{4.4}$$

where x = chip-wide L2 misses per uncore cycle
V = uncore supply voltage
F = uncore frequency

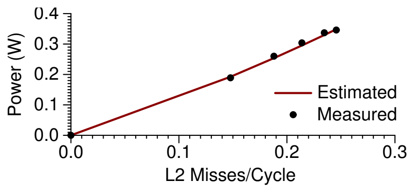


Figure 4.1: *Model Fitness for Power Consumed due to L2 Misses at F=800 MHz and V=0.7V*

We perform a similar experiment to measure effects of L3 misses on uncore power. For realistic L3 miss rates ($\leq$ 50 MPKI), CPU power consumption increases by less than 0.5 W even though DRAM power consumption increases significantly. Our models thus omit L3 misses, and we compute total uncore dynamic power consumption as the sum of uncore idle dynamic power and uncore power due to L2 misses.

4.1.2 Core Dynamic Power Model

To model core dynamic power consumption, we study the effects of microarchitectural events like micro-operations (uops) executed, L1 accesses, and L2 accesses. We start by quantifying the effects of non-memory operations on core dynamic power. We would prefer to analyze integer and floating point operations separately, but the Haswell microarchitecture provides no performance counter for floating point operations. We thus average over both instruction types. We create microbenchmarks that loop over the following instructions, alone and in combination:

- x87 floating point multiplications,
- x87 floating point additions,
- integer multiplications,
- integer additions,
- SIMD floating-point multiplications, and
- SIMD floating-point additions.

For each microbenchmark, we calculate αC using the same technique as for calculating idle uncore power: we fix uncore voltage, uncore frequency, and core voltage from the BIOS. We then initiate microbenchmark execution on all cores, measure power consumption, switch the cores to a higher frequency, and measure again within 10 ms.

Since the uncore voltage, uncore frequency, and package temperature are stable across readings (assuming that temperature change is insignificant within the very small sampling period), the difference in measured power must be due to changes in core dynamic power consumption. αC can now be calculated using Eq. 4.2. We calculate αC for multiple frequency pairs and take an average to reduce estimation error. We follow this approach to calculate αC for all microbenchmarks. We use linear regression to find the best fit for estimating the impact of non-memory instructions on core dynamic power. Eq. 4.5 shows the derived model, and Figure 4.2 shows its fitness with respect to our measured values. The R^2 fitness coefficient is 0.801.

$$\begin{aligned} P(core)_{non-mem\_dynamic} = & (2.448 \times 10^{-10} \times x + \\ & 1.181 \times 10^{-9}) \times V^2 \times F \end{aligned} \tag{4.5}$$

where $x =$ Non-memory instructions per cycle

$V =$ core supply voltage

$F =$ core frequency

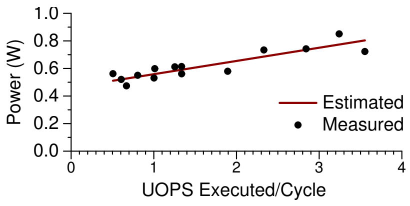


Figure 4.2: *Model Fitness for Power Consumed due to Non-Memory Execution Units at F=800 MHz and V=0.7V*

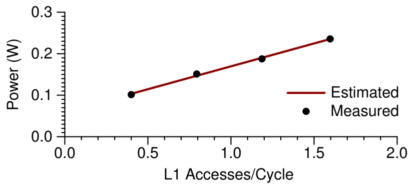


Figure 4.3: *Model Fitness for Power consumed due to L1 Accesses at F=800 MHz and V=0.7V*

We next study effects of memory operations on core dynamic power. We maintain constant activity on the non-memory execution ports (ports 0, 1, 5, and 6) and gradually increase it on the load/store units (ports 2, 3, 4, and 7). We correlate increases in memory operations with growth in core dynamic power to generate the model in Eq. 4.6. Figure 4.3 shows the model fitness. The R^2 coefficient is 0.999.

$$\begin{aligned} P(core)_{mem\_dynamic} = &(2.780 \times 10^{-10} \times x + \\ &1.497 \times 10^{-10}) \times V^2 \times F \end{aligned} \tag{4.6}$$

where $x =$ L1 accesses per cycle
$V =$ core supply voltage
$F =$ core frequency

To study the effects of L2 accesses on core dynamic power consumption, we follow the same approach as for studying the effects of L3 accesses. We again use our microbenchmark to increase the number of L2 accesses while keeping core activity constant. We measure the increase in CPU power consumption as we increase the L2 access rate. We then run linear regression on the L2 access rate and increase in power consumption to generate the model for the L2-access contribution to core dynamic power consumption shown in Eq. 4.7. Figure 4.4 shows the model fitness with respect to our measured values. The R^2 coefficient that measures the goodness of the fit is 0.997.

$$P(core)_{L2\_dynamic} = 2.829 \times 10^{-9} \times x \times V^2 \times F \tag{4.7}$$

$$\begin{aligned} \text{where } \; x &= \text{per-core L2 accesses per uncore cycle} \\ V &= \text{uncore supply voltage} \\ F &= \text{uncore frequency} \end{aligned}$$

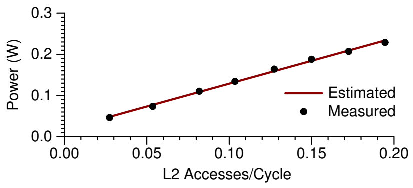


Figure 4.4: *Model Fitness for Power Consumed due to L2 Accesses at F=800 MHz and V=0.7V*

The total core dynamic power consumption can now be expressed by Eq. 4.8.

$$\begin{aligned} P(core)_{dynamic} = P(core)_{mem\_dynamic} + \\ P(core)_{non-mem\_dynamic} + \\ P(core)_{L2\_dynamic} \end{aligned} \tag{4.8}$$

4.1.3 Uncore Static Power Model

The Haswell microarchitecture uses a deep-sleep C-state, C7, when a core is idle. In the C7 state, the cores are power-gated, and so they consume negligible power. In contrast, our experiments show that the Haswell uncore is neither power-gated nor clock-gated when the chip is idle. The idle power consumption of the chip is thus almost entirely due to the uncore. Chip idle power can be expressed by Eq. 4.9.

$$\begin{aligned} P(CPU)_{idle} = & P(uncore)_{idle\_dynamic} + \\ & P(uncore)_{static} \end{aligned} \tag{4.9}$$

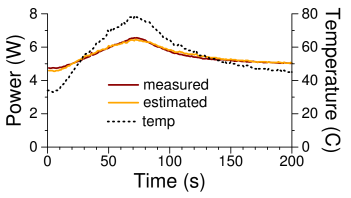
We use the model for $P(uncore)_{idle\_dynamic}$ derived in Section 4.1.1 to isolate $P(uncore)_{static}$. Recall that static power consumption depends on supply voltage and temperature. We use CPU package temperature to approximate average temperature across the uncore. To measure effects of uncore voltage and uncore temperature on uncore static power, we set the uncore voltage to values ranging from 0.7V to 1.0V with increments of 0.05V. At each voltage, we vary CPU temperature using a hot-air gun, and we measure power consumption at the granularity of one sample per second. We subtract uncore idle dynamic power from the measured values and run non-linear regression on this difference, the voltage, and the temperature to create the uncore static power model in Eq. 4.10. This equation is based upon the Poole-Frenkel effect [116] which provides a generic equation for current density through an electrical insulator. We use this equation simply as a base equation for curve-fitting and do not claim to estimate technology constants based upon the values acquired from non-linear regression. Figure 4.5 shows measured versus estimated uncore static values at three voltage points. We ran each experiment until temperature stopped dropping (the figure shows time windows of 200 seconds for ease of comparison — experiments that ran longer continued to show high estimation accuracy during the time not shown). The R^2 coefficient for the estimation model is 0.999 across all sample points.

$$P(uncore)_{static} = A \times e^{-(\frac{-169.083+1202.02\times\sqrt{V}}{273.15+T})} \times V^2 \tag{4.10}$$

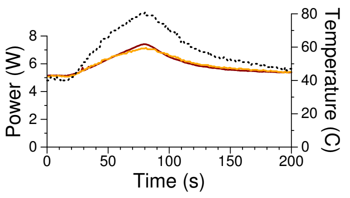
where $V =$ uncore supply voltage
$T =$ package temperature

4.1.4 Core Static Power Model

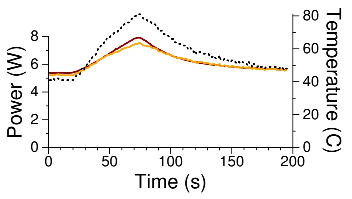
To generate a model for core static power, we force the core to remain in state C0. In this state, the Linux kernel runs a polling idle loop that consumes constant dynamic power. We calculate this power with the same strategy we used to calculate uncore idle dynamic



(a) V=0.70



(b) V=0.85



(c) V=1.0

Figure 4.5: *Uncore Static Model Fitness*

power. We calculate αC for idle cores in state C0 using Eq. 4.2. Eq. 4.11 shows the model for per-core dynamic power in state C0. Eq. 4.12 uses Eq. 4.11 to isolate per-core static power consumption.

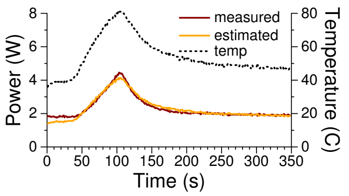
$$P(core)_{C0\_dynamic} = 1.28 \times 10^{-9} \times V^2 \times F \tag{4.11}$$

where V = core supply voltage

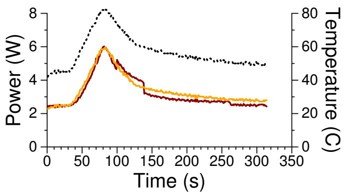
F = core frequency

$$\begin{aligned} P(core)_{static} = \frac{1}{4} \times (P(CPU)_{C0} - P(uncore)_{idle\_dynamic} \\ - P(uncore)_{static} - 4 \times P(core)_{C0\_dynamic}) \end{aligned} \tag{4.12}$$

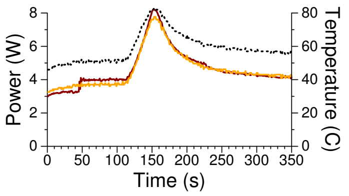
To create the core static power model we first measure the effects of core voltage and temperature. We use package temperature as an approximation for core temperature. We fix cache voltage at 0.7V and frequency at 800 MHz from the BIOS. We again incrementally increase core voltage from 0.7V to 1.05V in steps of 0.05V. At each voltage, we



(a) V=0.70



(b) V=0.85



(c) V=1.0

Figure 4.6: *Core Static Model Fitness*

vary package temperature using a hot-air gun, and we measure changes in power consumption and temperature. We run non-linear regression on collected data to create the core static model in Eq. 4.13. Figure 4.6 shows measured versus estimated core static values at three voltage points. The estimated model fits measured values with an R^2 coefficient of 0.996.

$$P(core)_{static} = 1525.07 \times e^{-(\frac{1884.1+525.556\times\sqrt{V}}{273.15+T})} \times V^2 \tag{4.13}$$

where V = core supply voltage

T = package temperature

4.1.5 Total Chip Power Model

Based on our findings, we must consider two more components in constructing a full-chip power model.

First, our measurements show that when core frequency is increased, chip power consumption increases more than expected at certain frequency steps. These increases appear to depend on the number of active cores but not on core activity. We believe this phenomenon is due to the core PLL switching to higher supply voltages above certain frequencies. An example of this unexpected increase in power is shown in Figure 4.7. We take following steps to conduct this experiment:

1. Fix core and uncore voltage to 1.0V (from BIOS).
2. Fix uncore frequency to 800 MHz (from BIOS).
3. Fix C-state for all cores to C0 (active).
4. Set power measurement equipment to take samples every millisecond.
5. Set core frequency to 800 MHz.
6. Keep cores idle for 50 ms.
7. Switch to next higher frequency for 10 ms.
8. Repeat steps 5 to 7 for all available frequencies.

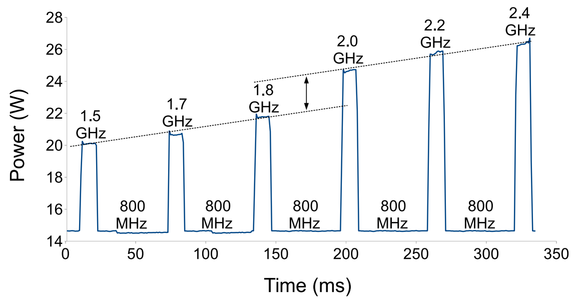


Figure 4.7: *Abrupt Jump in Power Consumption at Higher Frequency*

When the core frequency is switched, the temperature is not expected to change significantly due to small time duration (10ms) during which the higher frequency was enabled. Moreover the uncore voltage and frequency was fixed for the duration of the experiment. Hence, the change in in power consumption observed in the experiments is almost entirely due to change in core dynamic power consumption. Since the core voltage was also fixed from BIOS, we expect to see an almost linear rise in power consumption as frequency is increased. But as seen in Figure 4.7, this is true to till certain frequency point, after which there is an abrupt increase in power value, after which the power consumption again rises linearly as expected.

A similarly unexpected increase in power consumption is observed at higher temperatures — by up to 0.5W per active core — but our experiments and research have been unable to determine the source of this phenomenon. An example of this observation is shown in Figure 4.8. This data was collected by following steps listed below:

1. Fix the uncore voltage at 1.0V and core voltage at 0.85V (from BIOS)
2. Fix the core and uncore frequency to 800 MHz.
3. Fix the C-state for all cores to C0 (active)
4. Set power measurement equipment to take samples every second.
5. Use hot air gun to increase the chip temperature up to 80°C and then let it cool down.

As seen in the figure, the power consumption of the chip increases gradually in correlation with temperature increase till around the temperature of 74°C, at which point the power consumption abruptly increases by 2W. When the temperature is decreasing, the jump occurs at around 68°C. These temperature values have been observed to change as the voltage and frequency are varied but the change in power consumption of 0.5W per active core has been observed to be consistent.

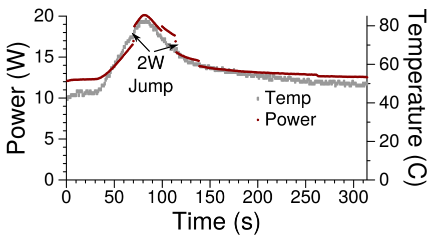


Figure 4.8: *Abrupt Jump in Power Consumption at Higher Temperature*

These two power-consumption components are difficult to model without more information about their causes. We therefore construct an offline table of empirically determined increases in power consumption due to these factors, acknowledging that deeper understanding is required to accurately predict their behavior. We represent these two components collectively with $P(misc)$. Our experiments indicate that $P(misc)$ can amount to as much as 10% of total chip power consumption, depending on the level of

Suite	Benchmarks
NAS	SP, EP, BT, MG, DC, UA, CG
SPEC OMP2001	quake, swim, wupwise, ammp, apsi, applu, mgrid
SPEC CPU2006	cactusADM, calculix, dealII, gamess, GemsFDTD, gromacs, lbm, leslie3d, milc, namd, povray, soplex, zeusmp, astar, bzip2, gcc, gobmk, h264ref, hmmer, libquantum, mcf, omnetpp, perlbench, sjeng, xalancbmk

Table 4.1: *Benchmarks Used for Validation*

core and uncore activity. Total chip power consumption is then given as:

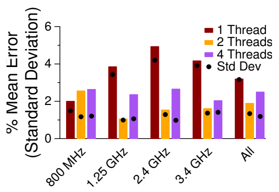
$$\begin{aligned} P(CPU) &= P(uncore)_{dynamic} + P(uncore)_{static} \\ &\quad + P(core)_{dynamic} + P(core)_{static} + P(misc) \end{aligned} \tag{4.14}$$

4.2 Validation

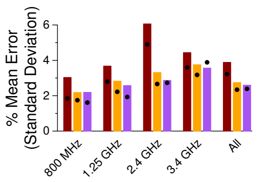
We validate our power models on the single-threaded SPEC CPU2006 [96] benchmarks, NAS Parallel Benchmarks [99], and multithreaded SPEC OMP2001 [97] applications in Table 4.1. We omit *bwaves* from SPEC CPU2006 and *art* and *fma3d* from SPEC OMP2001 because they do not run on our system. We use the NAS class B inputs, and we omit *IS* because it does not run for long enough time periods to gather reliable measurements. We run the parallel applications with one, two, and four threads. Similarly, we run one, two, and four concurrent instances of the sequential applications. Once per second we read package temperature and core voltage, and collect performance counter as described in Section 4.1. We measure processor power every 1 ms and average the values over one second to sync the power values with other parameters. To verify that our model estimates power accurately across different voltage-frequency steps, we validate it at four DVFS points: 800 MHz at 0.7V, 1500 MHz at 0.78V, 2400 MHz at 0.88V, and 3400 MHz at 1.01V.

Figure 4.9 shows estimation error for our benchmark suites. Mean absolute error (MAE) for all NAS benchmarks across the four DVFS points is 3.19% for a single thread, 1.89% for two threads, and 2.50% for four threads. MAE for SPEC CPU2006 is 3.88% for single-instance runs, 2.74% for double-instance runs, and 2.60% for quad-instance runs. MAE for SPEC OMP2001 is 3.79% for a single thread, 2.87% for two threads, and 3.42% for four threads. Mean absolute error across all sample points for all benchmarks and voltage-frequency states is 3.14%, and the standard deviation is 2.87%.

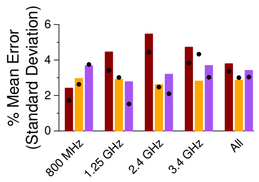
Our model predicts power estimation with good accuracy for most benchmarks. For



(a) NAS Parallel Benchmarks



(b) SPEC CPU2006



(c) SPEC OMP2001

Figure 4.9: *Validation of Total Chip Power*

the benchmarks with rapid phase changes (*DC* in NAS and *ammp* in SPEC OMP2001), we had to re-run the validation experiments at the higher measurement granularity of 100 ms to accurately estimate their rapidly changing power consumption values.

We know that model error arises from at least two sources. First, the Haswell microarchitecture provides no per-core event to track executed floating point operations, which prevents us from creating separate models for floating point and integer instruction executions. This creates slight model inaccuracies for compute-intensive phases. Second, values in the offline table we create to account for $P(misc)$ are not always accurate, especially at higher frequencies during periods of rapid phase change. Despite these known sources of error, our model accurately predicts workload energy trends when scaling both frequency and numbers of threads.

4.3 Energy Characterization of Frequency Scaling

Dynamic voltage-frequency scaling permits hardware or software to adjust clock speed (and voltage) in an attempt to trade-off performance for energy. However, finding the frequencies at which an application meets performance goals while maintaining a given energy budget is not necessarily straightforward. We use our power models to characterize energy efficiency by showing how static and dynamic energy from the core and uncore components scale in conjunction with DVFS.

We perform sensitivity analyses of energy consumption at different voltage-frequency points. Figure 4.10 depicts the mechanism we use for this sensitivity analyses. We *simulate* a hypothetical workload that is parameterized such that workload characteristics like memory-bound fraction can be varied individually without affecting the total work done. We define memory-bound fraction as the fraction of application whose execution time does not scale with CPU frequency. The workload characteristics are fed to the performance model along with operating frequency and number of parallel threads.

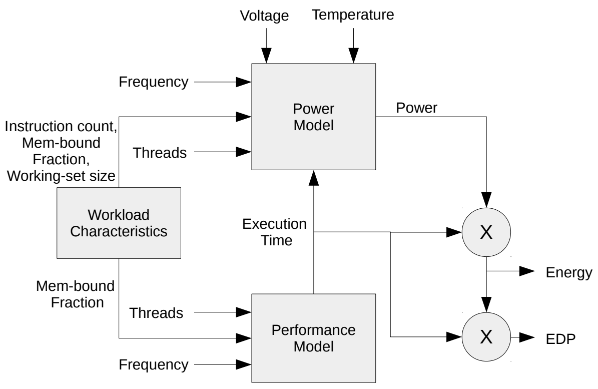


Figure 4.10: *Energy and EDP generation*

The performance model used for frequency scaling experiments calculates execution time based on Equation 4.15.

$$T_{F2} = T_{F1} \times (M + (1 - M) * \frac{F1}{F2}) \tag{4.15}$$

where T_{Fx} = Execution time at Frequency x

M = Memory bound fraction calculated at $F1$

The power model block in Figure 4.10 uses workload instruction count (both memory and non-memory), working-set size, frequency, number of parallel threads and execution time to generate event rates (events/cycle) required to calculate core and uncore dynamic power. For calculating static power, we use average temperature empirically observed across all benchmarks at each combination of DVFS operaring point and number of active cores. The power value generated by the power model is multiplied with the execution time generated by the performance model to calculate energy, which in turn is multiplied by the execution time to generate energy-delay product (EDP).

4.3.1 Energy Effects of DVFS

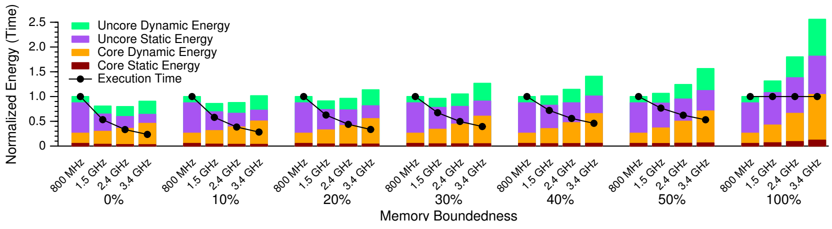
We first examine how voltage-frequency scaling affects energy consumption. Conventional wisdom says that running at higher frequencies boosts performance at the expense of expending more energy [109] because dynamic power scales quadratically with voltage (Eq. 4.1). But when we take into account the effects of energy consumed by the uncore, we find that running at a lower voltage-frequency step can sometimes expend *m*ore energy. Figure 4.11 shows how energy scales for our synthetic workload as we vary its memory-bound fraction and the frequency at which we execute. Energy numbers are normalized to the energy expenditure at the lowest frequency (800 MHz). We make following observations from these results.

Observation 1: The frequency at which lowest energy is expended depends on the memory-bound fraction of the application.

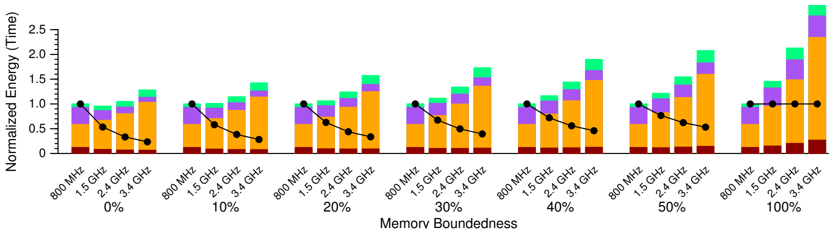
Explanation: At the lowest frequency for single-threaded run, the uncore energy accounts for 74% of the total. Specifically, uncore static energy constitutes 61% of the total energy expenditure. Increasing the CPU frequency reduces execution time and this results in reduction of uncore static energy. But increasing frequency increases core and uncore dynamic energy because of the quadratic relationship between dynamic power and voltage. The extent to which the performance scales with frequency depends on the memory-bound fraction of the application as per Equation 4.15. If the performance gain achieved by increasing frequency is large enough, the reduction in uncore static energy offsets the gain in dynamic energy resulting in overall reduction in total energy. This is what we see in Figure 4.11(a) when the frequency is increased from 800 MHz to 1500 MHz when memory-bound fraction is less than 40%. After a certain frequency, increases in (core and uncore) dynamic energy dwarf reduction in uncore static energy, causing total energy expenditure to rise again. The more memory-bound an application, the less performance it gains from running at higher frequencies, and so reductions in uncore static energy become less significant with increasing frequency. When the memory-bound portion of a single-threaded application exceeds 40%, we see the expected trend: increasing the frequency increases energy expenditure.

Observation 2: There is a significant difference between the energy trends for single-threaded run and four-threaded runs. For example, at 0% memory-boundedness, the least energy is expended at 2.4 GHz for single-threaded runs and 1.5 GHz for four-threaded runs.

Explanation: When more cores are active on the processor instead of one, at any given instant, the ratio of core power to total chip power is higher than the case when only one core is active. As a result, the uncore static energy is less significant for multi-threaded, CPU-bound applications (we measure 49% at two threads and 35% at four threads for 0% memory-bound). Consequently, when the frequency is increased for multi-threaded runs, the increase in core dynamic energy offsets the decrease in uncore static energy at lower memory-bound fractions compared to single-threaded runs. The energy scaling correlates with frequency scaling when a two-thread application is 30% memory-bound and a four-thread application is 20% memory-bound (compared to 40% for a single-threaded application).



(a) 1 Thread



(b) 4 Threads

Figure 4.11: *Effects of DVFS on Total Energy and Performance*

This analysis shows that the extent to which an application is memory bound must be taken into account when choosing a frequency at which to run the application such that it expends the least energy. We repeated our frequency scaling experiments while varying the workload characteristics like instruction mix and working-set size and came to conclusion that relationship between frequency and energy efficiency for any workload is only affected by the memory bound fraction of the workload and the number of parallel threads.

4.3.2 DVFS Prediction

The energy characterization of frequency scaling discussed in Section 4.3.1 underscores the importance of running either entire applications or their individual phases at appropriate frequencies to reduce system's energy expenditure. In this section, we leverage the the energy characterization done in that section to propose an online DVFS manager for energy-efficient frequency scaling. Prior works that leverages power modeling for implementing online DVFS manager [36, 109] propose the following approach:

1. Define a fixed scheduling quantum (for example, 10 ms).
2. Fix the system frequency for the current quantum.
3. Track performance events required to calculate power consumption at the currently chosen frequency.
4. At the end of the scheduling quantum, feed the collected statistics to the chosen energy model to predict energy at all the DVFS operating points for the system assuming that application would maintain similar behavior for the next quantum.
5. Set the system frequency to value that is predicted to expend least energy/EDP in the above step.
6. Repeat steps 3 to 5 at the end of every scheduling quantum.

This approach is shown to work well but requires power model calculations for each available system frequency at every scheduling quantum incurring energy overheads of its own. For example, on our Haswell processor there are sixteen different available frequency points available. This means that making frequency prediction for DVFS management would require sixteen energy calculations at every scheduling interval. This puts a constraint on how small the scheduling interval can be without incurring significant power and/or performance overhead of doing power calculations. We show in this section a much simpler approach to selecting the energy-efficient frequency, one that does not require such extensive calculations and is equally effective.

Figure 4.11 shows that the DVFS operating point at which a particular application will expend lowest energy is a function of an application's memory boundedness and the number of active cores. The memory boundedness of an application is a function of number of LLC misses, the LLC miss latency in terms of core cycles, and its effect on the application's execution time due to stalls in processor pipeline caused by data dependencies on LLC misses. The memory-bound fraction of the application can be formulated as the ratio of cycles during which execution was stalled due to LLC misses to total execution time in cycles. In an out-of-order processor, it is difficult to isolate execution stalls caused specifically by LLC misses from other overlapping stalls from events like branch mispredictions, the load/store queue getting full, or miss penalties from data accesses to lower-level caches. Due to this limitation, we must approximate the number of pipeline stalls caused by LLC misses. The Haswell microarchitecture supports a performance event named *CYCLE_ACTIVITY:STALLS_L2_PENDING*, which counts the number of core cycles during which the execution is stalled *and* an L2 miss is waiting to be served. It should be noted that this counter may not necessarily provide an exact count of execution stall cycles *due* to L2 misses, since it is possible that some of the stall cycles during a pending L2 miss were due to branch mispredictions. But given the high accuracy of modern branch predictors, we consider this event to be a good approximation of L2 miss stall cycles. From this event, we can estimate the L3 miss stall cycles using Equation 4.16. The L3 miss ratio in this equation is the ratio of memory loads that miss in L3 to total memory load operations retired. We consider only loads in this equation because store misses are not in the critical path of execution in modern out-of-order processors.

$$\begin{aligned}&L3\ Miss\ Stall\ Cycles = \\ &CYCLE\_ACTIVITY : STALLS\_L2\_PENDING \\ &\times L3\ Miss\ Ratio\end{aligned} \tag{4.16}$$

The memory-bound fraction can then be calculated using Equation 4.17.

$$Mem\ Bound\ Fraction = \frac{L3\ Miss\ Stall\ Cycles}{Total\ Cycles} \tag{4.17}$$

$$Cycles_{F2} = Cycles_{F1} \times (1 - M + M * \frac{F2}{F1}) \tag{4.18}$$

$$\begin{aligned}\text{where}\quad Cycles_{Fx} &= \text{Execution time in cycles at Frequency } x \\ M &= \text{Memory bound fraction calculated at } F1\end{aligned}$$

Overall, we must track the four performance events listed in Table 4.2 for calculating the memory-bound fraction of an application. To validate the accuracy of Equation 4.17, we run the benchmarks in Table 4.1 (at one, two, and four threads) at one frequency

Performance Event	Description
UNHALTED_CORE_CYCLES	Core cycles
MEM_LOAD_UOPS_RETIRED:L3_HIT	Memory loads that hit L3
MEM_LOAD_UOPS_RETIRED:L3_MISS	Memory loads that miss L3
CYCLE_ACTIVITY:STALLS_L2_PENDING	Pipeline stalls during L2 miss pending

Table 4.2: *Performance events required to calculated memory bound*

Parameter	Values
Frequency (GHz)	0.8, 1.0, 1.1, 1.3, 1.5, 1.7, 1.8, 2.0, 2.2, 2.4, 2.5, 2.7, 2.9, 3.1, 3.2, 3.4
Threads	1, 2, 4
Mem-bound %	0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100

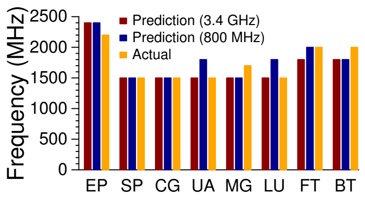
Table 4.3: *Parameter Values Used for Generating Energy Numbers*

and use the calculated memory bound to predict execution time at different frequencies using Equation 4.18. When we predict performance from 3.4 GHz to 800 MHz, the mean absolute error across benchmarks is 3.35%, with a standard deviation of 3.81%. When the performance is predicted from 800 MHz to 3.4 GHz, the mean absolute error is 6.58%, with a standard deviation of 6.37%. The prediction error increases for 800 MHz because some LLC-induced pipeline stalls that occur at higher frequencies may disappear at lower frequencies.

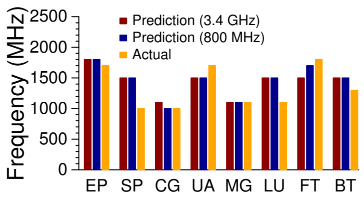
Our validation for performance prediction shows that we can predict the memory-bound fraction at any frequency with reasonable accuracy. This, in turn, means that we can predict the frequency at which the current application phase will be most energy-efficient by just tracking the events required to calculate the memory-bound fraction, instead of tracking all events required to calculate power consumption.

We repeat the methodology from Section 4.3.1, where we use the power models formulated for our system to calculate energy for a hypothetical workload while varying the core frequency, number of parallel threads, and the workload's memory-bound fraction. Table 4.3 lists the values of these three parameters that we use in our experiments. After predicting energy and EDP values at each combination of frequency, threads, and memory-bound fraction, we are able to determine the frequency at which a workload (or an individual phase) will expend the least energy/EDP. We populate a table with this information. An online DVFS manager can then simply use this table to predict the most energy-efficient voltage-frequency point for the current phase based on the calculated memory bound percentage, without the need to predict energy at all available system frequencies. This reduces the overhead associated with DVFS management while still retaining accurate energy-efficient frequency predictions.

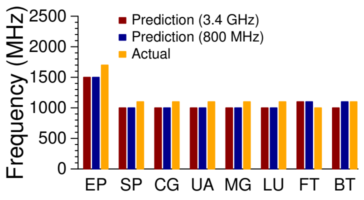
To validate this DVFS prediction methodology, we run benchmarks from Table 4.1 at all system frequencies to find the frequency at which the benchmark expends least energy (hereafter termed F_E). We then calculate the average memory-bound fraction of the application from the performance events collected at 3.4 GHz. Based on the calculated memory-bound fraction, we predict the frequency at which the benchmark would expend least energy (hereafter $P(3.4)_E$) and compare the prediction with the empirically determined best frequency (F_E). We repeat the same experiment by calculating the memory-bound fraction at 800 MHz and predicting the lowest energy frequency ($P(0.8)_E$). Figure 4.12 compares the predicted frequency versus actual frequency for the NAS benchmarks at which least energy is expended. Since the memory-bound fraction changes when the frequency changes, and because of the error in calculating memory boundedness, the predicted frequencies from 800 MHz and 3.4 GHz are not always the same. However, these two predictions are very close to the actual lowest energy frequency for almost all benchmarks. Figure 4.13 shows the energy expenditure at predicted frequencies normalized to energy expenditure at F_E. The error in frequency prediction results in an average of around 5% increase in energy expenditure across the benchmarks.



(a) 1 Thread

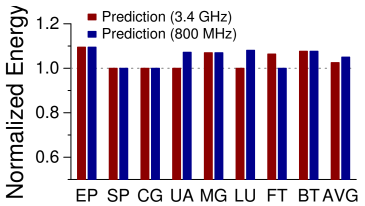


(b) 2 Threads

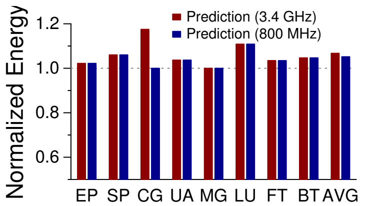


(c) 4 Threads

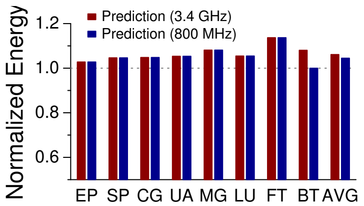
Figure 4.12: *Actual versus predicted frequency at which least energy is expended for NAS benchmarks*



(a) 1 Thread



(b) 2 Threads



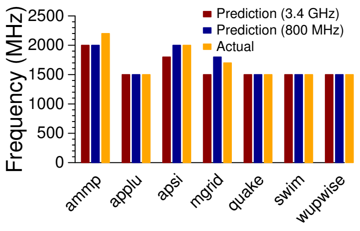
(c) 4 Threads

Figure 4.13: *Energy expenditure at $P(3.4)_E$ and $P(0.8)_E$ normalized to F_E for NAS benchmarks*

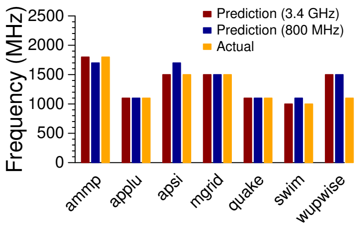
Figures 4.14 and 4.15 show the prediction error for SPEC OMP2001 benchmarks. Figures 4.16 and 4.17 show the prediction error for single-threaded SPEC CPU2006 benchmarks. For both SPEC OMP2001 and SPEC CPU2006, the frequency prediction for the lowest energy $(P(3.4)_E and P(0.8)_E)$ is very close to the actual best frequency (F_E), while the average energy costs of mispredictions are less than 5% for both benchmark suites.

The experimental results presented above validate our methodology for predicting energy-efficient frequencies without the need to predict energy at those frequencies.

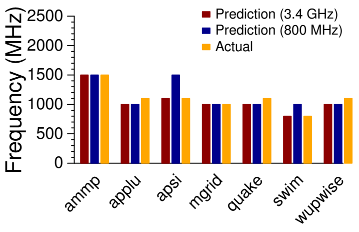
For online DVFS management, we must predict the energy-efficient frequency without prior knowledge of application's average memory boundedness. We developed a proof-of-concept userspace DVFS scheduler that tracks the performance events listed in Table 4.2 and uses Equation 4.17 to calculate memory boundedness every 10ms. We then use our look-up table, which gives the frequency for the lowest energy prediction for the calculated memory boundedness, and we switch to that frequency. We conduct the same experiments for achieving lowest EDP (this requires a separate look-up table for the lowest EDP prediction at the parameter values in Table 4.3). We first measure the power overhead of our DVFS manager (by running it when the system is idle) to confirm that invoking the DVFS scheduler every 10ms incurs less than 0.05W of power overhead. We then run our DVFS manager with an entirely CPU-bound application and



(a) 1 Thread

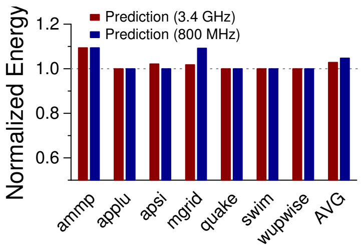


(b) 2 Threads

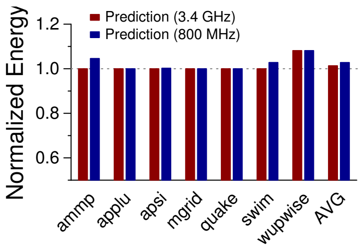


(c) 4 Threads

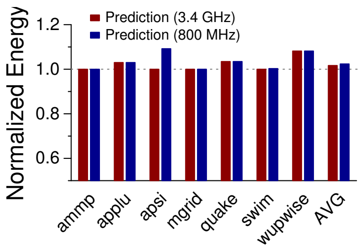
Figure 4.14: *Actual versus Predicted Frequency at which the Least Energy is Expended for SPEC OMP2001 Benchmarks*



(a) 1 Thread



(b) 2 Threads



(c) 4 Threads

Figure 4.15: *Energy Expenditure at $P(3.4)_E$ and $P(0.8)_E$ Normalized to F_E for SPEC OMP2001 Benchmarks*

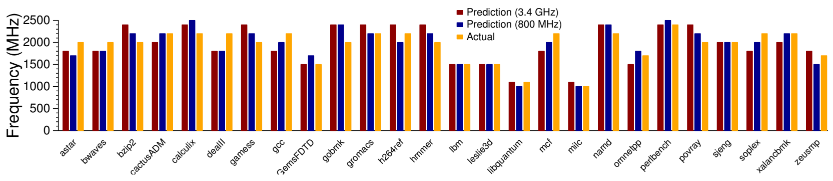


Figure 4.16: *Actual versus Predicted Frequency at which Least Energy is Expended for SPEC 2006 Benchmarks*

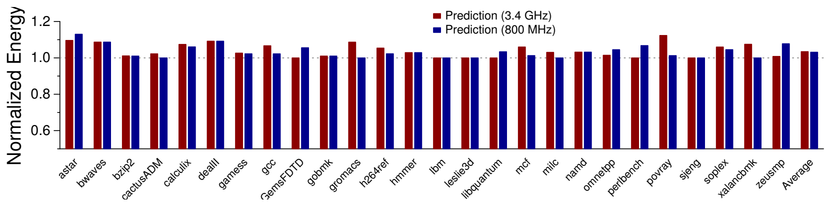


Figure 4.17: *Energy Expenditure at $P(3.4)_E$ and $P(0.8)_E$ Normalized to F_E for SPEC 2006 Benchmarks*

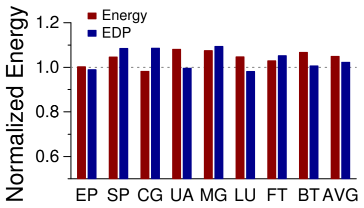
compare the performance overhead with the instance where the application is run at a statically determined lowest energy frequency, finding no discernible performance overhead. We then validate the DVFS manager for actual benchmarks. Figures 4.18, 4.19, and 4.20 show the energy expenditure/EDP while running benchmarks from different suites concurrently with the DVFS manager, with results normalized to the energy expenditure/EDP when the same benchmarks are run at the statically determined lowest energy/EDP frequency. For most of the benchmarks, energy expenditure under dynamic frequency prediction is within 10% of energy value obtained by running at the lowest energy/EDP frequency. In some instances, DVFS manager is able to reduce energy/EDP even more than the statically determined frequency because of the opportunities provided by phase changes within the workload. It should be noted here that since our Haswell processor only provides chip-wide DVFS, we are restricted to changing frequencies for all cores simultaneously. But our methodology can be easily used for performing per-core DVFS management.

4.4 Energy Characterization of Thread Scaling

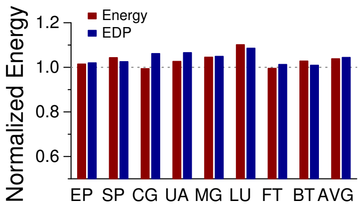
Chip multiprocessors can improve application performance by exploiting its inherent parallelism. A plethora of research has been done to maximize performance gains by parallelizing applications. But apart from performance gains, it is becoming increasingly important to consider the impact of parallelism on energy expenditure.

On a processor with ideal energy efficiency, performing a given amount of work should expend roughly the same amount of energy, regardless of the number of cores employed to do that work. But the sources of energy inefficiency in modern processors — and certain limiting characteristics of parallel applications — result in energy trends that differ from this "ideal" scenario.

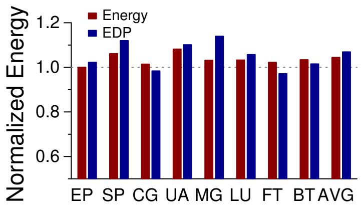
The following characteristics of parallel applications contribute to energy overhead when the number of parallel threads is increased:



(a) 1 Thread

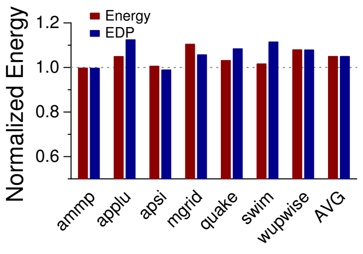


(b) 2 Threads

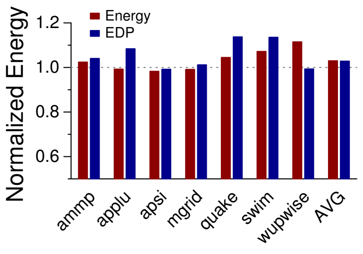


(c) 4 Threads

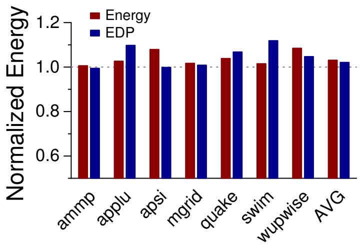
Figure 4.18: *Energy Consumed During DVFS versus Best Static Frequency for NAS Benchmarks*



(a) 1 Thread



(b) 2 Threads



(c) 4 Threads

Figure 4.19: *Energy Consumed During DVFS versus Best Static Frequency for SPEC OMP2001 Benchmarks*

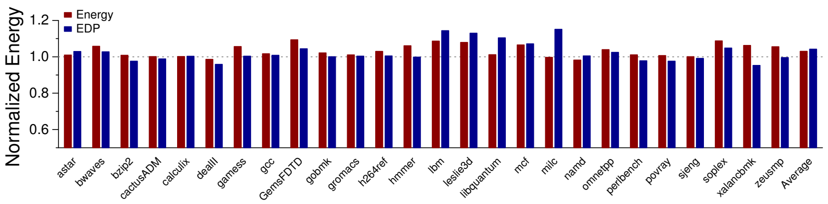


Figure 4.20: *Energy Consumed during DVFS versus Best Static Frequency for SPEC 2006 benchmarks*

- **Serial fraction**. Ideally, while executing the serial portion[1] of the application (when only one core is doing useful work), the other cores should consume no power. This can be achieved by power gating idle cores. But since a time penalty is associated with bringing the power gated core back to an active state, the processor only activates that state if it is sure that the time penalty is worth it. Depending on the length of serial portions, the processor may instead either let idle cores remain in active states or let them enter intermediate sleep states in which the core (or parts of it) may be clock gated but not power gated. Hence, the idle cores may consume some power, which directly contributes to the energy overhead of running applications in parallel.
- **Core-to-core communication overhead**. Most of the parallel threads share data to some extent. When one core requests data from another core, the execution pipeline of the first core may be stalled if its execution window does not have enough independent instructions to hide the remote cache access latency. The power consumed in the core during these execution stalls adds to the energy overhead of parallelism.
- **Bandwidth contention**. Depending on the application, the working set may grow when the number of parallel threads is increased. This could lead to increased contention for off-core and off-chip resources, resulting in potential increase in execution stalls, which in turn adds to the energy overhead.
- **Locking mechanism overhead**. In lock-based applications, as the number of parallel threads grows, the contention among threads trying to acquire the same lock also increases. Depending on the lock mechanism employed, this could result in increased execution cycles and/or total instructions executed, with both scenarios adding to the energy overhead. Another synchronization mechanism that can potentially increase total instructions executed is hardware/software transactional memory, which will be covered in Chapter 5.

We use our power models to analyze how thread scaling under the influence of variations in the listed parameters affects total energy energy expenditure and that of the core and uncore components.

To accurately study the affect of these parameters on the energy efficiency of thread scaling, we need must vary them in isolation. It is possible to do so using the same technique we use for DVFS scaling: simulate a hypothetical microbenchmark by feeding appropriate values to the power model. However, this would require creating a thread-scaling performance model that takes into account variables like remote core access

[1]we user the term "serial portion" to refer to individual serial sections of the application and "serial fraction" to refer to the sum of all serial portions as fraction of total application

latency, locking mechanism overhead, and effects of bandwidth contention on performance. In addition to the inputs required for performance modeling, other parameters like temperature variations, sleep state latencies, and instruction count overheads from synchronization primitives, make it more difficult to calculate a simulated application's performance and energy accurately. To avoid these limitations, we instead use a (real) microbenchmark for thread-scaling analysis and measure performance from runs on hardware instead of using performance model. We leverage the code from Eigenbench [114] developed by Hong et al. to create microbenchmarks for characterizing the energy when scaling numbers of threads. Eigenbench is a parameterized microbenchmark created to characterize the design space of transactional memory (TM) systems by defining TM-related characteristics that can be explored orthogonally. The same approach can be used to characterize the energy efficiency of thread scaling by defining orthogonal communication parameters and studying the effects of their variation on energy expenditure during thread scaling.

In the following sections, we describe the energy characterization analysis of thread scaling when varying individual application characteristics. The core and uncore frequency is fixed at 3.4 GHz for all analyses in this section.

4.4.1 Energy Effects of Serial Fraction

To analyze how the serial fraction of a parallel application affects its energy expenditure, we need a microbenchmark for which the serial fraction can be easily parameterized. In the original Eigenbench code, all work is done from the function `eigenbench_core`, which is called from within the parallel threads. To accurately control and vary the serial fraction of the application — without requiring to create a critical section — we call `eigenbench_core` from both inside and outside the parallel section. We then estimate the serial fraction of the application as the ratio of the total read and write operations done from the serial fraction to the sum of the reads and writes from both serial and parallel sections. We verify the serial fraction thus calculated by comparing the execution time predicted Equation: 4.19 (based on Amdahl's Law) to actual execution time. As per our experimental results, this gives us a reasonably accurate estimate of the application's serial fraction. Figure 4.21 shows the pseudo-code for the microbenchmark we use in this analysis.

$$T_{N2} = T_{N1} * (S + (1 - S) * \frac{N1}{N2}) \tag{4.19}$$

Test	oloop	iloop	array1	array2	R1	W1	R2	W2	Frequency	Range
Serial-Large	100	10000	32KB	32KB	0	Var	0	Var	3.4 GHz	W2 + W1 = 100, W2 = 0, 20, 40,...,100
Serial-Small	100	300	32KB	32KB	0	Var	0	Var	3.4 GHz	W2 + W1 = 100, W2 = 0, 20, 40,...,100

Table 4.4: *Microbenchmark Parameters for Serial Fraction Analysis*

where T_{Nx} = Execution time when number of parallel threads = x

S = Serial fraction of the application

During execution of the serial portion of the application, the idle cores (cores not running that serial portion) go into a sleep state (C-state on Intel machines). The bigger the serial portion, the deeper the sleep state that the idle cores can enter, and the larger the energy savings. The size of the serial portion depends on the number of operations in the `eigenbench_core` function, which, in turn, depends on the value of the variable `iloop`. For our analysis, we test with two sizes of `iloop`: 10000, which make sure that the idle cores during the serial portion enter the deepest sleep state (C7), and 300, which does not give enough time for idle cores to enter a sleep state deeper than C3. Table 4.4 lists the parameters we use for testing the sensitivity of energy expenditure on an application's serial fraction.

Figure 4.22 shows the energy expenditure and execution times from our analysis. Both the energy consumption and execution time values are normalized to those obtained for single-thread runs. We make the following observations about our results:

Observation 1: When the number of threads grows, the fraction of uncore energy to total chip energy goes down.

Explanation: This test does not share any data among parallel threads and the working-set fits in L2 cache. Hence the number of uncore accesses are negligible. This means that the uncore energy (both static and dynamic) is almost entirely idle energy. When only one core is active, this uncore idle energy constitutes 58% of total chip energy at a 0% serial fraction. When parallelism is introduced and more cores are active, the ratio of uncore idle power to total chip power goes down, and as a result, we see a reduction in uncore static and dynamic energy with increasing numbers of threads. At a 0% serial fraction in Figure 4.22(a), the uncore idle energy fraction goes down to 42% for two threads and 26% for four threads. When the serial fraction increases, the reduction in uncore idle energy becomes less pronounced. For example at 20% serial fraction in Figure 4.22(a), the serial fraction goes down from 58% at one thread to 46% at two threads and 35% at four threads. This is because when the application's serial fraction increases, the performance gain from increasing the number of threads diminishes. When the application is completely serial, there is no performance gain when the

```
void eigenbench_core(loop,r,w,array) {
  long val = 0;
  for (i=0; i<iloop; i++) {
    for (j=0; j<(w+r) ; j++) {
      (action) = rand_action(w, r);
      index = rand_index(tid, array);
      if (action == READ)
        val += array[index];
      else
        array[index] = val;
    }
  }
}

void main() {
  int R1,W1,R2,W2; // R1 and W1: reads and writes in serial
                       // R2 and W2: reads and writes in parallel
  long *array2[NUM_THREADS], *array1; //array1 is accessed
      serially
                                    //array2 is accessed by
                                        parallel threads;
                                    // but each thread writes
                                        in own copy of array2
  int iloop,oloop,array2_size,array1_size;

  set_params(R1,W1,R2,W2,iloop,oloop,array1_size,array2_size);

  Initialize_array(array1,array1_size);
  for(j=0;j<NUM_THREADS;j++) {
    Initialize_array(array2[j],array2_size);
  }

  for(i=0;i<oloop;i++) {
    for(j=0;j<NUM_THREADS;j++) {
      // ARGS -> iloop/NUM_THREADS, R2, W2, array2[j]
      thread_start(eigenbench_core(),ARGS);
    }
    barrier();
    eigenbench_core(iloop,R1,W1,array1);
  }
}
```

Figure 4.21: *Pseudo-Code for Serial Fraction Analysis*

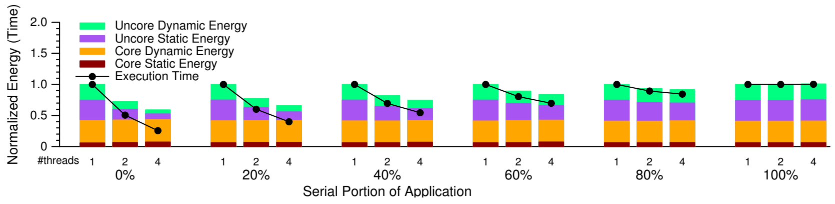
degree of parallelism increases. This is why there is no reduction in uncore idle energy from one to four threads. This analysis highlights the role of energy expenditure in the idle uncore.

Observation 2: The degree of parallelism at which the lowest energy is expended depends on the serial fraction and the size of the serial portion of the application.

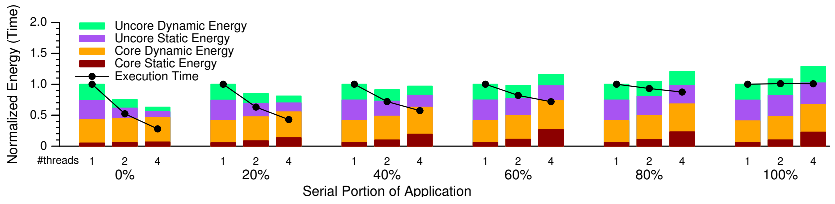
Explanation: When the serial fraction is small, increasing the number of parallel threads always results in reduced energy expenditures because of the reduction in uncore idle energy discussed above. When the serial fraction increases, which results in lower speedups, we still see reductions in energy for applications with large serial portions. This is because when the serial portion is large, the inactive cores enter the C7 sleep state in which the core is power gated, and hence they do not consume either static or dynamic power. As a result, core dynamic and static energy remain the same across varying serial fraction and degrees of parallelism, as Figure 4.22(a) shows. Combined with reductions in uncore energy, this results in decreases in total chip energy (until the serial fraction hits 100%). But when the serial portion is small, the inactive cores only enter the shallower sleep states (C1/C1E/C3) in which the cores are not power gated and only partially clock gated. The cores still consume static and dynamic power, resulting in increases in core static and dynamic energy as numbers of threads are increased. This offsets the reduction in uncore energy to some extent. Figure 4.22(b) shows that at lower serial fraction, the reduction in uncore energy offsets the increase in core energy at all degrees of parallelism, and hence running the application at four threads is most energy efficient. At around a 40% serial fraction, when the number of threads increases from two to four, the reduction in uncore energy is not enough to offset the increase in core energy, and hence energy expenditure at two threads is less than at four threads, even though the execution time at four threads is smaller. As the serial fraction is increased, this effect becomes more pronounced. When the serial fraction is greater 80% or more, it becomes more energy efficient to simply run the application serially.

4.4.2 Energy Effects of Core-to-core Communication Overhead

To analyze the effects of core-to-core communication on the energy efficiency of thread scaling, we need our microbenchmark to accurately control the fraction of shared memory writes to total memory writes. We tweak the microbenchmark for serial fraction analysis to use two buffers from the `eigenbench_core` function: one (`array2`) is private to the thread and hence does not generate coherence operations, and the other (`array1`) is shared among the threads and hence generates coherence operations. Figure 4.23 shows the pseudocode for this analysis. In this version of the pseudocode, we



(a) Large serial portion: `iloop`=10000



(b) Small serial portion: `iloop`=300

Figure 4.22: *Effects of Serial Fraction on the Energy Efficiency of Thread Scaling*

Test	iloop	array1	array2	R1	W1	R2	W2	Frequency	Range
Core-to-core	50000	32KB	32KB	0	Var	0	Var	3.4 GHz	W2 + W1 = 100, W2 = 0, 20, 40,...,100

Table 4.5: *Microbenchmark Parameters for Core-to-core Communication Overhead Analysis*

do not call the work function from outside the parallel section, and hence the serial fraction is always zero. We do not use synchronization primitives to protect shared writes because we only want to isolate the effects of core-to-core communication overheads and correctness is not of essence here.

Table 4.5 lists the parameters used for testing sensitivity of energy expenditure on core-to-core communication overhead. Figure 4.24 shows the execution time and energy expenditure trends for varying fractions of shared writes to total writes by parallel threads. Both energy and execution times are normalized to those for single-thread runs. The absolute energy expenditure at all single-thread runs is the same, since for single-thread runs all cache access hit in the private L1/L2 caches.

We make the following observations from these results.

Observation 1: The execution time and energy expenditure are almost the same for 0% and 20% shared writes, even though there is a significant increase in L2 misses for the latter.

Explanation: When only 20% of the memory accesses are shared, the latency of getting data from a remote cache is hidden by the instruction-level parallelism in the

```
void eigenbench_core(loop,r1,w1,array1,r2,w2,array2) {
  long val = 0;
  for (i=0; i<iloop; i++) {
    for (j=0; j<(w1+r1+w2+r2) ; j++) {
      (action) = rand_action(r1, w1, r2, w2);
      index = calc_index(tid, array); //Randomization not used to
                                      //avoid non-buffer memory
                                          ops

      if (action == READ1)
        val += array1[index];
      else if (action == WRITE1)
        array1[index] = val;
      else if (action == READ2)
        val += array2[index];
      else
        array2[index] = val;
    }
  }
}

void main() {
  int R1,W1,R2,W2; // R1 and W1: reads and writes in array1
                       // R2 and W2: reads and writes in array2

  long *array2[NUM_THREADS], *array1; // Each thread writes in
      own copy of array2
                                      // array1 is shared among
                                          parallel threads

  int iloop,oloop,array2_size,array1_size;

  set_params(R1,W1,R2,W2,iloop,oloop,array1_size,array2_size);

  Initialize_array(array1,array1_size);
  for(j=0;j<NUM_THREADS;j++) {
    Initialize_array(array2[j],array2_size);
  }

  for(j=0;j<NUM_THREADS;j++) {
    // ARGS -> iloop/NUM_THREADS, R1, W1, array1, R2, W2, array2[
        j]
    thread_start(eigenbench_core(),ARGS);
  }
}
```

Figure 4.23: *Pseudo-Code for Core-to-core Communication Overhead Analysis*

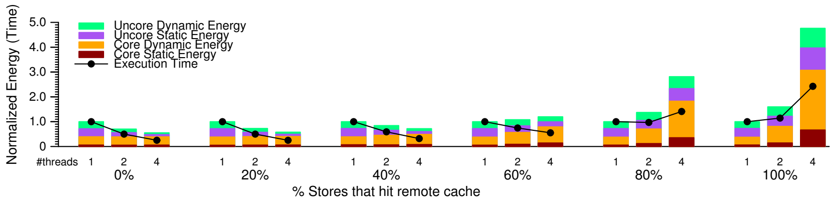


Figure 4.24: *Effects of Core-to-core Communication on Energy Efficiency of Thread Scaling*

Test	oloop	iloop	array1	array2	R1	W1	R2	W2	Frequency	Range
BW Contention	1	100000	0	Var	0	0	10	90	3.4 GHz	array2 = 512KB,1MB,...,16MB

Table 4.6: *Microbenchmark parameters for Bandwidth Contention Analysis*

core pipeline. The execution time does not increase for small fractions of shared memory accesses. The energy expenditure increases only slightly (2%) because the contribution of uncore accesses to uncore dynamic energy is small compared to uncore idle energy. This, again, underlines the importance of low-power design for the uncore, which could potentially lead to significant energy savings.

Observation 2: There are significant increases in core energy at higher fractions of shared memory accesses, even though the work done remains the same.

Explanation: When the fraction of shared writes is 60%, execution time from one to four threads decreases by 45%, but energy expenditure increases by 20%. When the fraction of shared writes is 80%, the execution time from one to two threads decreases by 3% and energy increases by 37%. In the first case, the uncore energy goes down by 17%, and in the second case, it increases by 7%. The main contribution to the increase in energy expenditure comes from the increase in core energy, both static and dynamic, even though the number of instructions executed remains the same across the number of threads. This is because when the pipeline is stalled due to L2 misses, the core is in the active C-state (C0), and hence it is consuming both static and dynamic energy. This results in increases in core energy expenditure for the same amount of work done.

4.4.3 Energy Effects of Bandwidth Contention

This analysis is designed to study cases where increasing parallelism results in increased bus and memory contention due to increases in working-set size, even though the total work done remains the same. For this analysis, we use the microbenchmark code shown in Figure 4.21, but with the serial fraction set to zero and with varying sizes of `array2`. Table 4.6 lists the parameter values we use.

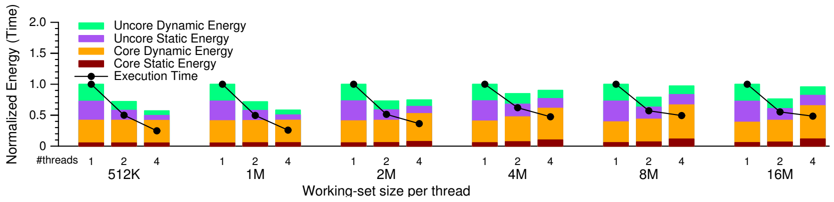


Figure 4.25: *Effects of Bandwidth Contention on Energy Efficiency of Thread Scaling*

The execution time and energy expenditure results from this analysis are shown in Figure 4.25. The working-set size on the x-axis is per thread. We make the following observations from these results.

Observation 1: There is almost no time or energy penalty for increases in contention when the total working-set size fits within L3 cache.

Explanation: When the working-set size per thread is less than 2 MB, the working set fits in the 8 MB L3 cache at all levels of parallelism. When the working set per thread is increased from 512KB to 1MB, we see an increase of around 17% in total L2 misses, but no increase in execution time or energy. Even when we decrease the working-set size per thread to 128 KB (not shown in the figure), the execution time and energy do not change significantly, even though the number of L2 misses does decrease as expected. This is because the latency of fetching data from the L3 is short enough that instruction level parallelism in pipeline can hide it. The energy also remains the same because, again, the energy expenditure due to L2 misses is a small fraction of uncore idle energy.

Observation 2: When the total working-set size does not fit in the L3 cache, running at two threads expends the least energy, even though four threads gives the lowest execution time.

Explanation: As we observed in the core-to-core communication analysis, when core execution is stalled due to data dependencies on L3 misses, the core waits for data in an active state, and hence it consumes static and dynamic energy. Also, as we observed before, any speedup due to parallelism almost always leads to reduction in uncore energy. When the parallelism leads to increased memory contention but still delivers some speedup, uncore energy is reduced, but core energy is increased due to stall cycles. Going from one to two threads, the reduction in uncore energy is greater than the increase in core energy. However, going from two to four threads, the increase in core energy offsets the reduction in uncore energy, resulting in the energy trends we observe in our results.

Test	iloop	array	R1	W1	R2	W2	Frequency	Range
Lock Overhead	100000	32KB	0	Var	0	Var	3.4 GHz	W1=100,50,40,30,20,10 W2=W1*9

Table 4.7: *Microbenchmark Parameters for Lock-Overhead Analysis*

4.4.4 Energy Effects of Locking-Mechanism Overhead

In this analysis, we characterize the effects of overheads related to locking mechanisms on the energy efficiency of thread scaling. Figure 4.26 shows the pseudo-code of the microbenchmark we use for this test. We use a pthreads mutex as the locking mechanism. Table 4.7 shows the parameter values we use. We vary the size of critical section by controlling the number of stores inside it. The number of stores outside the critical section is always equal to nine times the number from inside ($W2 = W1 \times 9$), which effectively sets the serial fraction of the application to 10%. We observe the effects of locking-mechanism overhead on the energy efficiency of thread scaling as we decrease the size of the critical section. Figure 4.27 shows our results.

We make following observations from these results.

Observation 1: The core energy increases significantly with increases in the number of parallel threads. When the critical section is large (100 stores), the core energy increases by 16% from one to two threads and by 40% from one to four threads. When the critical section is small (10 stores), the core energy increases by 93% from one to two threads and 133% from one to four threads.

Explanation: As we saw in our serial fraction analysis for the case where the serial portion of the application is small (Figure 4.22(b)), core energy is expected to increase when parallelism is introduced due to inactive cores not being able to enter deeper sleep states. In this case, the critical section is small enough that the inactive cores spend their idle cycles in either the active state or the shallowest sleep state (C1/C1E), resulting in increase sin energy consumption. Moreover, as the critical section gets smaller, the overhead of acquiring and releasing locks gets higher. When the critical section has 100 stores, the total instruction execution count increases by 3% from one to four threads. When the critical section has only 10 stores, the instruction count increases by 15% from one to four threads, contributing to the overall increase in core energy expenditure.

Observation 2: For small critical sections, the energy expenditure at two threads is more than the energy expenditure at one and four threads. Also, running at one thread is most energy efficient, even when running at four threads gives best performance.

Explanation: When the size of the critical section is reduced, the overhead related to acquiring the lock starts affecting both speedup and the energy efficiency of thread scaling. When the application is scaled from one to two threads, the energy reduction in the uncore (due to speedup) is not enough to offset the energy increase. When the

```
void eigenbench_core(loop,r1,w1,r2,w2,array) {
  long val = 0;
  for (i=0; i<iloop; i++) {

    acquire_lock();
    for (j=0; j<(w1+r1) ; j++) {
      (action) = rand_action(w1, r1);
      index = rand_index(tid, array);
      if (action == READ)
        val += array[index];
      else
        array[index] = val;
    }
    release_lock();
    barrier();

    for (j=0; j<(w2+r2) ; j++) {
      (action) = rand_action(w2, r2);
      index = rand_index(tid, array);
      if (action == READ)
        val += array[index];
      else
        array[index] = val;
    }
  }
}

void main() {
  int R1,W1,R2,W2; // R1 and W1: reads and writes inside critical
       section
                       // R2 and W2: reads and writes outside
                          critical section
  long *array[NUM_THREADS]; //array is accessed by parallel
      threads;
                            // but each thread writes in own copy
                                 of array
  int iloop,array_size;

  set_params(R1,W1,R2,W2,iloop,array_size);

  for(j=0;j<NUM_THREADS;j++) {
    Initialize_array(array[j],array_size);
  }

  for(j=0;j<NUM_THREADS;j++) {
    // ARGS -> iloop/NUM_THREADS, R1, W1, R2, W2, array[j]
    thread_start(eigenbench_core(),ARGS);
  }
}
```

Figure 4.26: *Pseudo-Code for Locking Mechanism Overhead Analysis*

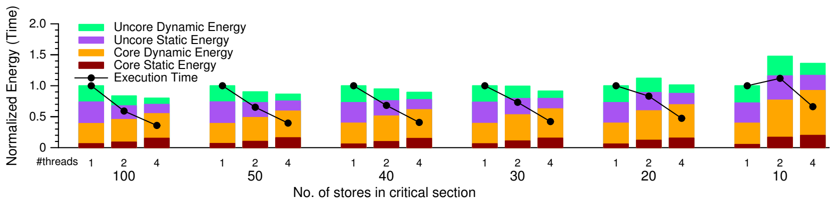


Figure 4.27: *Effects of Locking-Mechanism Overhead on the Energy Efficiency of Thread Scaling*

application is scaled from two to four threads, the speedup obtained further reduces uncore energy expenditure, which this time offsets the increase in core energy but not enough to reduce the energy expenditure compared to the run at a single thread. We see this trend for all sizes of critical sections smaller than or equal to 20 stores. But this holds true only when the serial fraction is small enough that increasing the number of threads provides significant speedups. If the serial fraction is increased from 10% to greater than 60%, the speedup obtained going from two to four threads is not enough to offset the increase in core energy expenditure, and, as a result, total energy at four threads is higher than the energy value at two threads.

4.5 Related Work

Various publications in the past have explored the usage of performance counters for power estimation. Some of the related work has already been discussed in the previous chapter. Here, we discuss previous research on power estimation modeling that explicitly attempts to differentiate between static and dynamic power consumption.

The power model proposed by Bertran et al. [50] makes a distinction between dynamic and static power. However, they define static power as idle power and use regression on the values of their chosen events to derive both static (idle) and dynamic power. Unlike us, their power models disregard both voltage and temperature and hence, cannot be used for characterizing the true static and dynamic power consumption of the chip.

Spiliopoulos et al. [36] also address static and dynamic power separately in their approach to power estimation, but they also define static power as idle power. They create a table (offline) of idle power values at all frequency steps and “typical core temperature ranges”. We instead differentiate between static power (based on voltage and temperature) and idle dynamic power.

Huang et al. [117] present the methdology used to develop a power proxy for estimating per-core power consumption for IBM POWER7+ server chip. They break down the power consumption values in active power, clock grid power and leakage power. They leverage the in-house information available about the technology constants to build their power model while our methodology requires no such prior knowledge. Their power estimation logic is implemented partly in hardware and partly in firmware and hence, requires on-chip hardware support; while our model is implemented fully in software.

Su et al. [109] use 12 performance events to estimate power consumption for their AMD system and hence, require time-multiplexing to gather data for all events. They address idle and dynamic power separately. Like our second approach, they use voltage and temperature to develop a static power model, but they group idle dynamic power into their static power model, which does not reflect the true static power consumption of the core and the northbridge (equivalent to the uncore in our models).

The use of power models to drive DVFS policies has also been explored by some of the previous research papers.

Su et al. [109] and Spiliopoulos et al. [36] use their performance and energy estimation models to predict the voltage-frequency operating point at which the current application phase will expend lowest energy/EDP. In contrast to our work, their DVFS scheduler needs to perform energy calculations for each available DVFS operating point at runtime, while we use a look-up table to directly predict the low energy/EDP operating point without the need to perform any calculations and hence, induces lower overhead. This makes our methodology more suitable for future systems where scheduling decisions need to be taken more frequently and possibly, across more number of voltage-frequency pairs.

Su et al. [109] also their models to characterize energy expenditure on their AMD FX-8320 system at various frequencies. Their system always expends least energy at the lowest voltage-frequency state in contrast to our observations on Haswell machine. The frequency scaling energy trends would differ among microarchitectures and implementations depending on the energy expenditure fraction of idle uncore (or Northbridge) which is affected by the level of clock and power gating.

Patki et al. [112] develop a power-aware job scheduler for power constrained cluster. Like our DVFS scheduler, their scheduling policies also rely upon a look-up table populated with details like execution time and power for certain application configurations in advance to achieve the scheduling complexity of $O(1)$.

Dzhagaryan et al. [118] study the impact of thread and frequency scaling on performance and energy expenditure of Intel Xeon processor. Unlike us they do not analyze the underlying reasons for the observed energy trends, namely the static and dynamic power consumption of core and uncore. They use PARSEC benchmarks for their study while we use microbenchmarks to isolate the effect of individual communication parameters.

4.6 Conclusions

Accurately estimating the power consumption of processor components is important for supporting better power-aware resource management. We present a methodology to estimate static and dynamic power consumption of the core and the uncore processor components. The methodology uses core and uncore voltage, package temperature, and performance counters to create models that can estimate power consumption for sequential and parallel applications across all system frequencies. We validate our models on benchmarks from NAS, SPEC CPU2006, and SPEC OMP2001, and we show that our models can estimate power with high accuracy (3.14% mean absolute error) across all voltage-frequency pairs and different concurrency levels.

We use our power models to study the impact of DVFS on energy consumption, showing that — contrary to conventional wisdom — it is not always most energy efficient to run applications at the lowest frequency. Uncore static energy effects must be taken into consideration. The frequency at which an application expends the lowest energy depends on how memory bound it is and how many concurrent threads it uses. We study the impact of thread scaling on energy expenditure, showing that the relative serial portion of a program influences the level of concurrency at which the program will expend the least energy.

We leverage the energy characterization for frequency scaling to implement an online DVFS manager that can predict energy-efficient frequency for the current application phase without performing actual energy predictions at runtime by using a look-up table, thus incurring low overhead. We show that such a simple DVFS manager is able to achieve high levels of energy-efficiency.

We characterize the energy expenditure at different levels of parallelism and demonstrate how the static and dynamic power consumption from core and uncore components affect the energy trends. We define four orthogonal communication parameters and isolate their affect on the energy efficiency of thread scaling. We show that the number of threads at which lowest energy expenditure is achieved for a parallel program is not the same at which lowest execution time is achieved. The energy consumed by the idle uncore and the stalled core significantly impact the energy efficiency of thread scaling.

5 Characterization of Intel's Restricted Transactional Memory

Transactional memory (TM) [119] simplifies some of the challenges of shared-memory programming. The responsibility for maintaining mutual exclusion over arbitrary sets of shared-memory locations is devolved to the TM system, which may be implemented in software (STM) or hardware (HTM). TM presents the programmer with fairly easy-to-use programming constructs that define a *transaction* — a piece of code whose execution is guaranteed to appear as if it occurred atomically and in isolation.

The research community has explored this design space in depth, and a variety of proposed systems take advantage of transaction characteristics to simplify implementation and improve performance [120–123]. Hardware support for transactional memory has been implemented in Rock [124] from Sun Microsystems, Vega from Azul Systems [125], and Blue Gene/Q [126] and System z [127] from IBM. Haswell is the first Intel product to provide such hardware support. Intel's Transactional Synchronization Extensions (TSX) allow programmers to run transactions on a best-effort HTM implementation, i.e., the platform provides no guarantees that hardware transactions will commit successfully, and thus the programmer must provide a non-transactional path as a fallback mechanism. Intel TSX supports two software interfaces to execute atomic blocks: Hardware Lock Elision (HLE) is an instruction set extension to run atomic blocks on legacy hardware, and Restricted Transactional Memory (RTM) is a new instruction set interface to execute transactions on the underlying TSX hardware.

The previous chapters have highlighted the need for better hardware introspection into power consumption. As better hardware support for such introspection becomes available, it is important to evaluate accuracy and acuity so that users can choose among various combinations of measurement and modeling techniques. Since Intel's Core i7 4770 microarchitecture is among the first to support both power modeling and support for transactional memory, it makes an interesting platform for analyzing power/performance trade-offs.

As an initial study, we compare the Haswell RTM performance and energy of the Haswell implementation of RTM to those of other approaches for controlling concurrency. We use a variety of workloads to test the susceptibility of RTM's best-effort nature to performance degradation and increased energy consumption. We compare RTM performance to TinySTM, a software transactional memory implementation that uses time to reason about the consistency of transactional data and about the order of transaction commits.[1] We highlight these crossover points and analyze the impact of thread scaling on energy expenditure.

We find that RTM performs well with small to medium working sets when the amount of data (particularly that being written) accessed in transactions is small. When data contention among concurrent transactions is low, TinySTM performs better than RTM, but as contention increases, RTM consistently wins. RTM generally suffers less overhead than TinySTM for single-threaded runs, and it is more energy-efficient when working sets fit in cache.

5.1 Experimental Setup

The Intel 4th Generation Core$^{\text{TM}}$ i7 4770 processor comprises four physical cores that can run up to eight simultaneous threads when hyper-threading is enabled. Each core has two eight-way 32 KB private L1 caches (separate for I and D), a 256 KB private L2 cache (for combined I and D), and an 8 MB shared L3 cache, with 16 GB of physical memory on board. We compile all microbenchmarks, benchmarks, and synchronization libraries using gcc v4.8.1 with *-O3* optimization flag. We use the *-mrtm* flag to access the Intel TSX intrinsics. We schedule threads on separate physical cores (unless running more than four threads) and fix the CPU affinity to prevent migration.

[1]We choose TinySTM because during our experiments we find that it consistently outperforms other STM alternatives like TL2 (to which RTM was compared in another recent study [128]).

We modify the *task* example from *libpfm4.4* to read both the performance counters and the processor package energy via the Running Average Power Limit (RAPL) [84] interface. As we concluded in Chapter 2, RAPL is fairly accurate when the time between successive RAPL counter reads is more than tens of milliseconds. Since all our benchmarks run for at least few seconds, we decided to use RAPL for our energy measurements. We implement Intel TSX synchronization as a separate library and add RTM definitions to the STAMP *tm.h* file. When transactions fail more than eight times, we invoke reader/writer lock-based fallback code to ensure forward progress. If the return status bits indicate that an abort was due to another thread's having acquired the lock (in the fallback code), we wait for the lock to be free before retrying the transaction. The following shows pseudocode for a sample transaction.

Algorithm 1 Implementation of `BeginTransaction`

```
while true do
    nretries ← nretries + 1
    status ← _xbegin()
    if status = _XBEGIN_STARTED then
        if arch_read_can_lock(serialLock) then
            return
        else
            _xabort(0)
        end if
    end if
    {*** fall-back path ***}
    while not arch_read_can_lock(serialLock) do
        _mmpause()
    end while
    if nretries ≥ MAX_RETRIES then
        break
    end if
end while
arch_write_lock(serialLock);
return
```

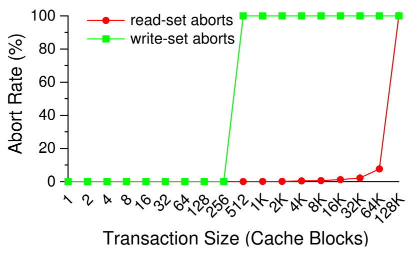


Figure 5.1: *RTM Read-Set and Write-Set Capacity Test*

5.2 Microbenchmark analysis

5.2.1 Basic RTM Evaluation

We first quantify RTM's hardware limitations that affect its performance using microbenchmark studies. We detail the results of these experiments below.

RTM Capacity Test. To test the limitations of read-set and write-set capacity for RTM, we create a custom microbenchmark, results for which are shown in Fig. 5.1. The abort rate of write-only transactions tops out at 512 cache blocks (the size of L1 data cache). We suspect this is because write-sets are tracked only in L1, and so evicting any transactionally written cache line from L1 results in a transaction abort. For read-sets, the abort rate saturates at 128K cache blocks (the size of L3 cache). This suggests that evicting transactionally read cache lines from L3 (but not L1) triggers transaction aborts, and thus RTM maintains performance for much larger read-sets than write-sets.

RTM Duration Test. Since RTM aborts can be caused by system events like interrupts and context switches, we study the effects of transaction duration (measured in CPU cycles) on success rate. For this analysis, we use a single thread, set the working-set size to 64 bytes, and set the number of writes inside the transaction to 0. This tries to ensure that the number of aborts due to memory events and conflicts remains insignificant. We gradually increase the duration by increasing the number of reads within the transaction. Fig. 5.2 shows that transaction duration begins to affect the abort rate at about 30K cycles and that durations of more than 10M cause all transactions to abort (note that these results are likely machine dependent).

RTM Overhead Test. Next we quantify performance overheads for RTM compared to spin locks and the atomic compare-and-swap (CAS) instruction. For this test, we create a microbenchmark that removes elements from a queue (defined in the STAMP [129] library). We initialize the queue to 1M elements, and threads extract elements until the queue is empty. Work is not statically divided among threads. We first compare RTM against the spinlock implementation in the Linux kernel (`arch/x86/include/asm/`

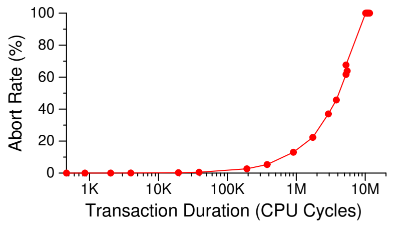


Figure 5.2: *RTM Abort Rate versus Transaction Duration*

spinlock.h). We then compare against a version of the queue implementation modified to use CAS in queue_pop(). For RTM, we simply retry the transaction on aborts.

We perform three sets of experiments. To observe the cost of starting an RTM transaction in the absence of contention, we first run single-threaded experiments. We repeat the experiment with four threads to generate a high-contention workload. Finally, we lower contention by making threads work on local data for a fixed number of operations after each critical section. Table 5.1 summarizes execution times normalized to those of the lock-based version.

	Type of synchronization			
Contention	None	Lock	CAS	RTM
None	0.64	1	1.05	1.45
Low	N/A	1	0.64	0.69
High	N/A	1	0.64	0.47

Table 5.1: *Relative Overhead of RTM versus Locks and CAS*

Table 5.1 shows that the cost of starting a transaction makes RTM perform worse than the other alternatives when executing non-contended critical sections with few instructions. RTM suffers about a 45% slowdown compared to using locks and CAS, and it takes over twice the time of an unsynchronized version. In contrast, our multi-threaded experiments reveal that RTM exhibits roughly 30% and 50% lower overhead than locks in low and high contention, respectively, while CAS is in both cases around 35% better than locks. Note that transactions avoid hold-and-wait behavior, which seems to give RTM an advantage in our study. When comparing locks and CAS, the higher lock overhead is likely due in part to the ping-pong coherence behavior of the cache line containing the lock and to cache-to-cache transfers of the line holding the queue head.

Characteristic	Definition
Concurrency	Number of concurrently running threads
Working-set size[a]	Size of frequently used memory
Transaction length	Number of memory accesses per transaction
Pollution	Fraction of writes to total memory accesses inside transaction
Temporal locality	Probability of repeated address inside transaction
Contention	Probability of transaction conflict
Predominance	Fraction of transactional cycles to total application cycles

[a]Working-set size for Eigenbench is defined per-thread.

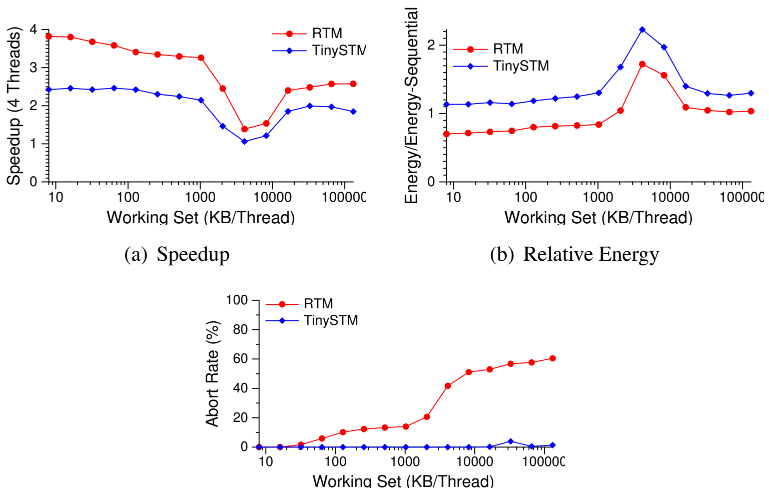
Table 5.2: *Eigenbench TM Characteristics*

5.2.2 Eigenbench Characterization

To compare RTM and STM in detail, we next study the behaviors of Hong et al.'s Eigenbench [114]. This parameterizable microbenchmark attempts to characterize the design space of TM systems by orthogonally exploring different transactional application behaviors. Table 5.2 defines the seven characteristics we use to compare performance and energy expenditure of the Haswell RTM implementation and the TinySTM [130] software transactional memory system. Hong et al. [114] provide a detailed explanation of these characteristics and the equations used to quantify them.

Unless otherwise specified, we use the following parameters in our experiments, results for which we average over 10 runs. Transactions are 100 memory references (90 reads and 10 writes) in length. We use one small (16KB) and one medium (256KB) working set size to demonstrate the differences in RTM performance. Since L1 size has no influence on TinySTM's abort rates, we only show TinySTM results for the smaller working set size. To prevent L1 cache interference, we run four threads with hyper-threading disabled as our default, and we fix the CPU affinity to prevent thread migration. For each characteristic, we compare RTM and TinySTM performance and energy (versus sequential runs of the same code) and transaction-abort rates. For the graphs in which we plot two working-set sizes for RTM, the speedups and energy efficiency given are relative to the sequential run of the same size working set.

Working-Set Size. Fig. 5.3 shows Eigenbench results over a logarithmic scale as we increase each thread's working set from 8KB to 128MB. RTM performs best with the smallest working set, and its performance gradually degrades as working-set size increases. The performance of both RTM and TinySTM drops once the combined working sets of all threads exceed the 8MB L3 cache. RTM performance suffers more because events like L3 evictions, page faults, and interrupts trigger a transaction abort, which



(a) Speedup

(b) Relative Energy

(c) Abort Rate

Figure 5.3: *Eigenbench Working-Set Size*

is not the case for TinySTM. The speedups of both RTM and TinySTM are lowest at working sets of 4MB: at this point, the parallelized code's working sets (16MB in total) exceed L3, but the working set of the sequential version (4MB) still fits. For working sets above 4MB, the sequential version starts encountering L3 misses, and thus the relative performances of both transactional memory implementations begins to improve. Parallelizing the transactional code using RTM is energy-efficient compared to sequential version when the combined working sets of all threads fits inside the cache.

Transaction Length. Fig. 5.4 shows Eigenbench results as we increase the transaction length from 10 to 520 memory operations. When the working set (16KB) fits within L1, RTM outperforms TinySTM for all transaction lengths. For 256KB working sets, RTM performance drops sharply when the transaction length exceeds 100 accesses. Recall that evicting write-set data from the L1 triggers transaction aborts, but when the working set fits within L1, such evictions are few. As the working set grows, the randomly chosen addresses accessed inside the transactions have a higher probability of occupying more L1 cache blocks, and hence they are more likely to be evicted. In contrast, TinySTM shows no performance dependence on working-set size. The overhead of starting the hardware transaction affects RTM performance for very small transactions. As observed in the working-set analysis above, RTM is more energy efficient than both the sequential run and TinySTM for all transaction lengths when using the smaller work-

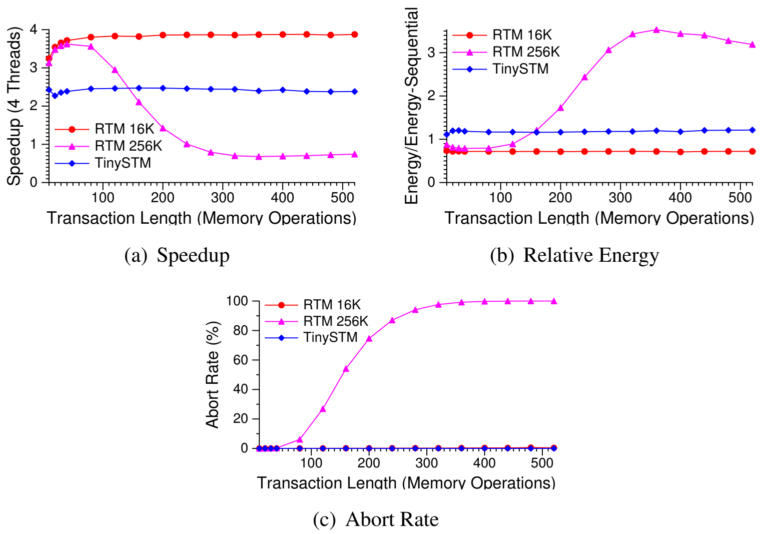


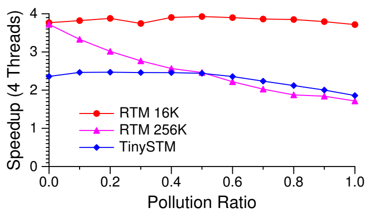
Figure 5.4: *Eigenbench Transaction Length*

ing set. When using the larger working set, RTM expends more energy for transactions exceeding 120 accesses.

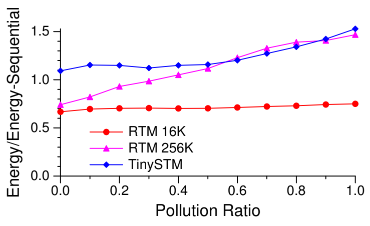
Pollution. Fig. 5.5 shows results when we test symmetry (with respect to handling read-sets and write-sets) by gradually increasing the fraction of writes. The pollution level is zero when all memory operations in the transaction are reads and one when all are writes. When the working set fits within L1, RTM shows almost no asymmetry. But for the larger working-set size, RTM speedup suffers as the level of pollution increases. TinySTM outperforms RTM when the pollution level increases beyond 0.4.

Temporal Locality. We next study the effects of temporal locality on TM performance (where temporal locality is defined as the probability of repeatedly accessing the same memory address within a transaction). The results in Fig. 5.6 reveal that RTM shows no dependence on temporal locality for the 16KB working set, but performance degrades for the 256KB working set (where low temporal locality increases the number of aborts due to L1 write-set evictions). In contrast, TinySTM performance degrades as temporal locality increases, indicating that it favors unique addresses unless only one address is being accessed inside the transaction (locality = 1.0).

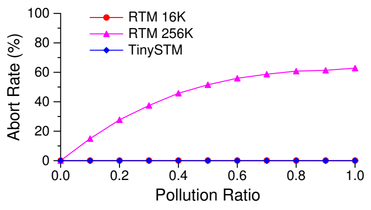
Contention. This analysis studies the behavior of TM systems when the level of contention is varied from low to high. We set the working-set size to 2MB for both RTM and TinySTM. The level of contention is calculated as an approximate value rep-



(a) Speedup

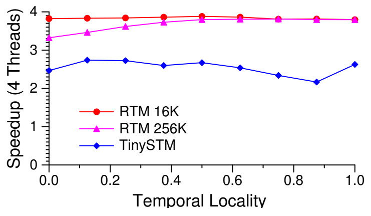


(b) Relative Energy

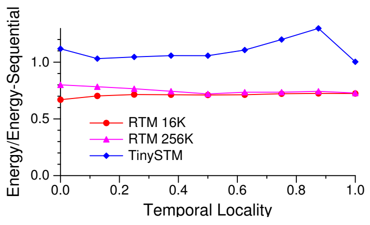


(c) Abort Rate

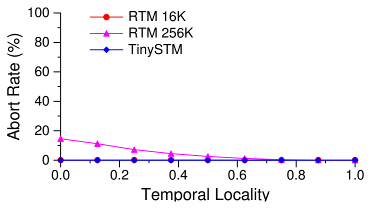
Figure 5.5: *Eigenbench Pollution*



(a) Speedup

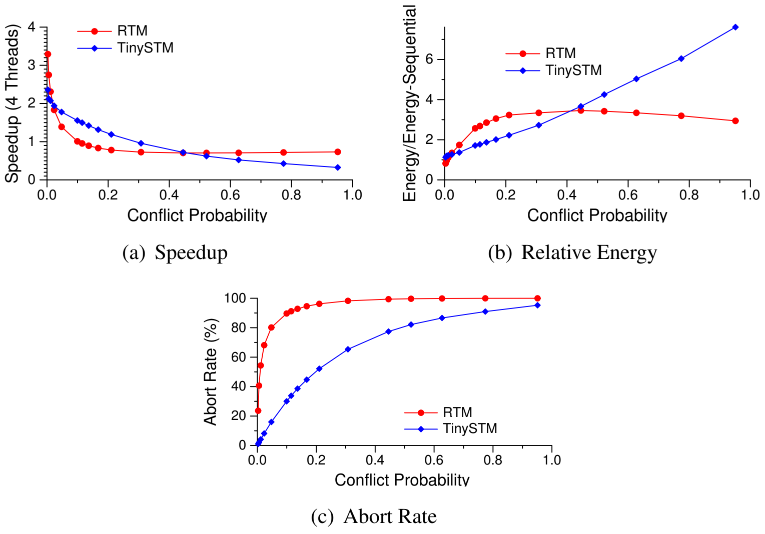


(b) Relative Energy



(c) Abort Rate

Figure 5.6: *Eigenbench Temporal Locality*



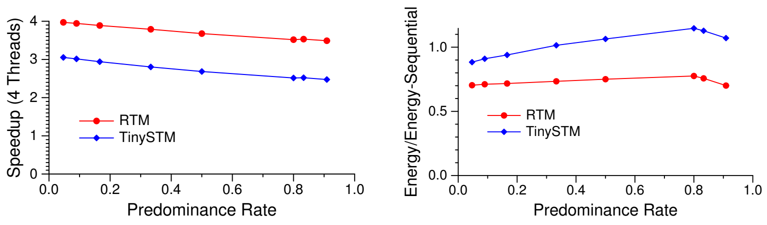
(a) Speedup
(b) Relative Energy
(c) Abort Rate

Figure 5.7: *Eigenbench Contention*

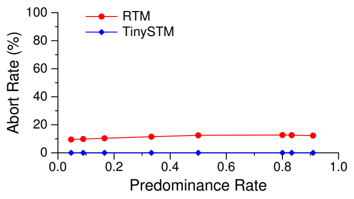
resenting the probability of a transaction causing a conflict (as per the probability formula given by Hong et al. [114]). The conflict probability figures shown in Fig. 5.7 are calculated at word granularity and hence are specific to TinySTM. Since RTM detects conflicts at the granularity of cache line (64 bytes), the contention level is actually higher for RTM with the same workload configuration. When the degree of contention among competing threads is very low, RTM performs better than TinySTM. For low to medium contention, TinySTM considerably outperforms RTM. However, for high contention workloads, TinySTM performance degrades while RTM performance remains almost the same.

Predominance. We study the behavior of the TM systems when varying the fraction of application cycles executed within transactions to the total number of application cycles. For this analysis, we set working-set size to 256KB for both TM systems, we set contention to zero, and we vary the predominance ratio from 0.125 to 0.875. Fig. 5.8 shows that performance for both RTM and TinySTM suffers as the ratio of transactional cycles to non-transactional cycles grows. This can be attributed to the overheads associated with the TM systems: for the same level of predominance, TinySTM introduces more overhead because it must instrument the program memory accesses.

Concurrency. Next we study how the performance and energy of RTM and TinySTM scale when concurrency is increased from one thread to eight. Fig. 5.9 shows that RTM



(a) Speedup (b) Relative Energy



(c) Abort Rate

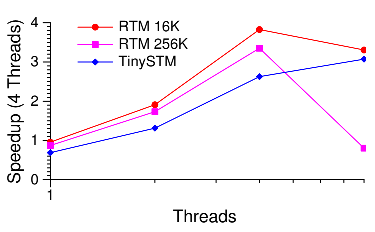
Figure 5.8: *Eigenbench Predominance*

scales well up to four threads. At eight threads, the L1 cache is shared between two threads running on the same core. This cache sharing degrades performance for the larger working set more than for the smaller working set because hyper-threading effectively halves the write-set capacity of RTM. In contrast, TinySTM scales well up to eight threads. For the small working set, RTM proves to be more energy-efficient than either TinySTM or the sequential runs.

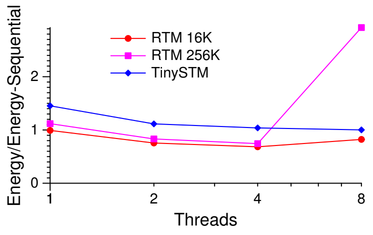
The results from the Eigenbench analysis help us in identifying a range of workload characteristics for which either RTM or TinySTM is better performing or more energy efficient. We next apply the insights gained from our microbenchmark studies to analyze the performance and energy numbers we see for the STAMP benchmark suite.

5.3 HTM versus STM using STAMP

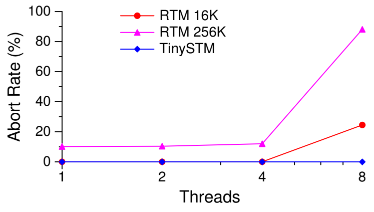
Next we use the STAMP transactional memory benchmark suite [129] to compare the performance and energy efficiency of RTM and TinySTM. We use the lock-based fallback mechanism explained in Section 5.1 and run the applications with input sizes that create large working sets and high contention. We average all results over 10 runs. Fig. 5.10 shows STAMP execution times for RTM and TinySTM normalized to the average execution time of sequential (non-TM) runs. Fig. 5.11 shows the corresponding en-



(a) Speedup



(b) Relative Energy



(c) Abort Rate

Figure 5.9: *Eigenbench Concurrency*

ergy expenditures, again normalized to the average energy of the sequential runs. Results for single-threaded TM versions of the benchmarks illustrate the TM system overheads.

`bayes` has a large working set and long transactions, and thus RTM performs worse than TinySTM. This corresponds to our findings in the Eigenbench transaction-length analysis in Fig. 5.4. As expected, RTM does not improve the performance of `bayes` as the number of threads scales, and TinySTM performs better overall. Since the time the `bayes`'s algorithm takes to learn the network dependencies depends on the computation order, we see significant deviations in learning times for multi-threaded runs.

`genome` has medium transactions, a medium working-set size, and low contention. Most transactions have fewer than 100 accesses. Recall that in the working-set analysis shown in Fig. 5.3(a) (for transaction length 100), RTM slightly outperforms TinySTM for working-set sizes up to 4MB. On the other hand, TinySTM outperforms RTM when contention is low (Fig. 5.7(a)). The confluence of these two factors within `genome` yields similar performances for RTM and TinySTM up to four threads. For eight threads, as expected, TinySTM's performance continues to improve, whereas RTM's suffers from increased resource sharing among hyper-threads.

`intruder` is also a high-contention benchmark. As with `genome`, RTM performance scales well from one to four threads. Since `intruder` executes very short transactions, scaling to eight threads does not cause as much resource contention as for genome, and thus RTM and TinySTM perform similarly. Even though this application has a small to medium working set — which might otherwise give RTM an advantage — its performance is dominated by very short transaction lengths.

kmeans is a clustering algorithm that groups data items in *N*-dimensional space into *K* clusters. As with bayes, our 10 runtimes deviate significantly for the multi-threaded versions. On average, RTM performs better than TinySTM. The short transactions experience low contention, and the small working set has high locality, all of which give RTM a performance advantage over TinySTM. Even though both TM systems show speedups over the sequential runs, synchronizing the kmeans algorithm in TinySTM expends more energy at all thread counts.

labyrinth routes a path in a three-dimensional maze, where each thread grabs a start and an end point and connects them through adjacent grid points. Fig. 5.10 shows that labyrinth does not scale in RTM. This is because each thread makes a copy of the global grid inside the transaction, triggering capacity aborts that eventually cause the fallback to using a lock. Energy expenditure increases for the RTM multi-threaded runs because the threads try to execute the transaction in parallel but eventually fail, wasting many instructions while increasing cache and bus activity.

ssca2 has short transactions, a small read-write set, and low contention, and thus even though it has a large working set, it scales well to higher thread counts. Performance for eight threads is good for both RTM and TinySTM. In general, RTM performs better (with respect to both execution time and energy expenditure) but not by much, as is to be expected for very short transactions.

vacation has low to medium contention among threads and a medium working set size. The transactions are of medium length, locality is medium, and contention is low. Like genome, vacation scales well up to four threads, but performance degrades for eight threads because its read-write set size is large enough that cache sharing causes resource limitation issues.

yada has big working set, medium transaction length, large read-write set, and medium contention. All these conditions give TinySTM a consistent performance advantage over RTM at all thread counts.

Our results in Fig. 5.11 indicate that the energy trends of applications do not always follow their performance trends. Applications like bayes, labyrinth, and yada expend more energy as they are scaled up, even when performance changes little (or even improves, in the case of yada). Only intruder, kmeans, and ssca2 benefit from hyper-threading under RTM. In contrast, most STAMP applications benefit from hyper-threading under TinySTM, and those that do not suffer only small degradations.

Fig. 5.12 shows the overall abort rates for all benchmarks, including the contributions of different abort types. Based on our observations of hardware counter values,

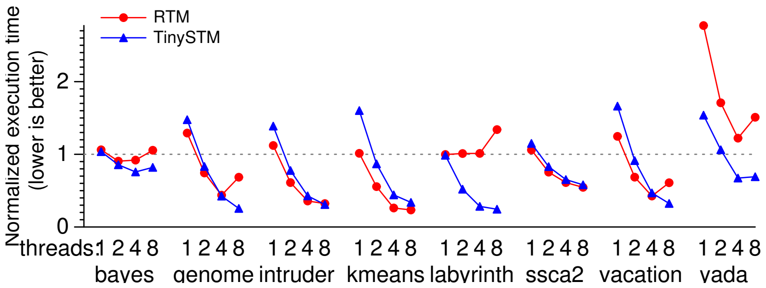


Figure 5.10: *RTM versus TinySTM Performance for STAMP Benchmarks*

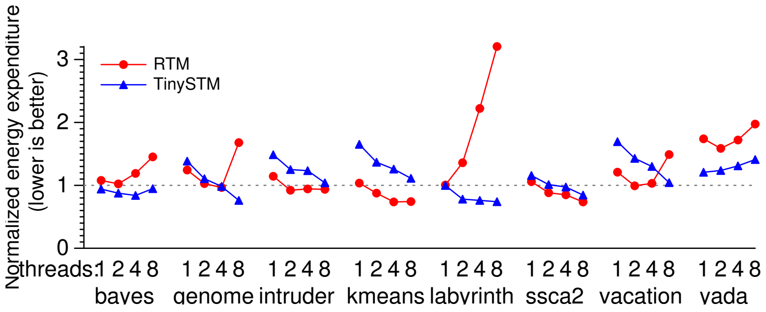


Figure 5.11: *RTM versus TinySTM Energy Expenditure for STAMP Benchmarks*

Abort Type	Description
Data-conflict/ Read-capacity	Conflict aborts and read-set capacity aborts
Write-capacity	Write-set capacity aborts
Lock	Conflict and explicit aborts caused by serialization locks
Misc3	Unsupported instruction aborts[a]
Misc5	Aborts due to none of the previous categories[b]

[a] includes explicit aborts and aborts due to page fault/page table modification
[b] interrupts, etc.

Table 5.3: *Intel RTM Abort Types*

the current RTM implementation does not seem to distinguish between data-conflict aborts and aborts caused by read-set evictions from L3 cache, and thus both phenomena are reported as conflict aborts. When a thread incurs the maximum number of failed transactions and acquires the lock in the fallback path, it forces all currently running transactions to abort. We term this a *lock* abort. These aborts are reported either as conflict aborts, *explicit* aborts (i.e., deliberately triggered by the application code), or both (i.e., the machine increments multiple counters). Lock aborts are specific to the fallback mechanism we use in our experiments. Other fallback mechanisms that do not use serialization locks within transactions (they can be employed in non-transactional code) do not incur such aborts. Note that avoiding lock aborts does not necessarily result in better performance since the lock aborts mask other type of aborts (i.e., that would have occurred subsequently). This can be seen in abort contributions shown in the figure. As applications are scaled, the fraction of aborts caused by locks increases because every acquisition potentially triggers N-1 lock aborts (where N is the number of threads).

The `RTM_RETIRED:ABORTED_MISC3` performance counter reports aborts due to events like issuing unsupported instructions, page faults, and page table modifications. The `RTM_RETIRED:ABORTED_MISC5` counter includes miscellaneous aborts not categorized elsewhere, such as aborts caused by interrupts. Table 5.3 gives an overview of these abort types. In addition to these counters, three more performance counters represent categorized abort numbers: `RTM_RETIRED:ABORTED_MISC1` counts aborts due to memory events like data conflicts and capacity overflows; `RTM_RETIRED:ABORTED_MISC2` counts aborts due to uncommon conditions; and `RTM_RETIRED:ABORTED_MISC4` counts aborts due to incompatible memory types (e.g., due to cache bypassing or I/O accesses). In our experiments, `RTM_RETIRED:ABORTED_MISC4` counts are always less than 20, which we attribute to hardware error (as per the Intel specification update [131]). In all our experiments, `RTM_RETIRED:ABORTED_MISC2` is zero.

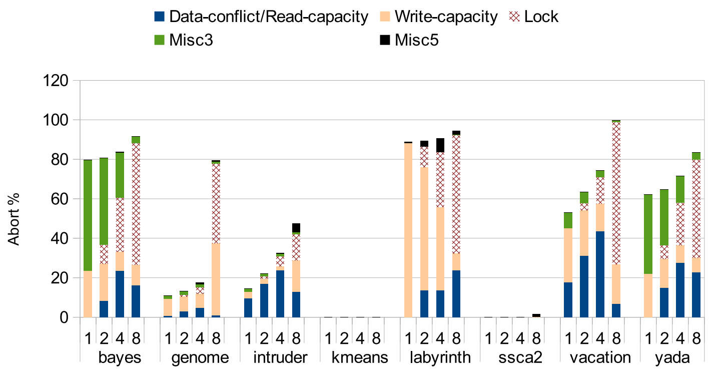


Figure 5.12: *RTM Abort Distributions for STAMP Benchmarks*

5.4 Related Work

Hardware transactional memory systems must track memory updates within transactions and detect conflicts (read-write, write-read, or write-write conflicts across concurrent transactions or non-transactional writes to active locations within transactions) at the time of access. The choice of where to buffer speculative memory modifications has microarchitectural ramifications, and commercial implementations naturally strive to minimize modifications to the the cores and on-chip memory hierarchies on which they are based. For instance, Blue Gene/Q [126] tracks updates in the 32MB L2 cache, and the IBM System z [127] series and the canceled Sun Rock [124] track updates in their store queues. Like the Haswell RTM implementation that we study here, the Vega Azul Java compute appliance [125] uses the L1 cache to record speculative writes. The size of transactions that can benefit from such hardware TM support depends on the capacity of the chosen buffering scheme. Like us, others have found that rewriting software to be more transaction-friendly improves hardware TM effectiveness [125].

Previous studies investigate the characteristics of hardware transactional memory systems. Wang et al. [126] use the STAMP benchmarks to evaluate hardware transactional memory support on Blue Gene/Q, finding that the largest source of TM overhead is loss of cache locality from bypassing or flushing the L1 cache. Schindewolf et al. [132] analyze the performance of TM subsystem in Blue Gene/Q using CLOMP-TM benchmark.

Yoo et al. [128] use the STAMP benchmarks to compare Haswell RTM with the TL2 software transactional memory system [133], finding significant performance differences between TL2 and RTM. We perform a similar study and find that TinySTM consistently outperforms TL2, and thus we choose the former as our STM point of comparison. Our RTM scaling results for STAMP benchmark concur with their results.

Wang et al. [134] evaluate RTM performance for concurrent skip list scalability, comparing against competing synchronization mechanisms like fine-grained locking and lock-free linked-lists. They use the Intel RTM emulator to model up to 40 cores, corroborating results for one to eight cores with Haswell hardware experiments. Like us, they highlight RTM performance limitations due to capacity and conflict misses and propose programmer actions that can improve RTM performance.

Others have also studied power/performance trade-offs for TM systems. For instance, Gaona et al. [135] perform a simulation-based energy characterization study of two HTM systems: the LogTM-SE *Eager-Eager* system [120] and the Scalable TCC *Lazy-Lazy* system [136]. Ferri et al. [137] estimate the performance and energy implications of using TM in an embedded multiprocessor system-on-chips (MPSoCs), providing detailed energy distribution figures from their energy models.

In contrast to the work presented here, none of these studies analyzes energy expenditure for a commercial hardware implementation.

5.5 Conclusions

The Restricted Transactional Memory support available in the Intel Haswell microarchitecture makes programming with transactions more accessible to parallel computing researchers and practitioners. In this study, we compare RTM and TinySTM, a software transactional memory implementation, in terms of performance and energy. We highlight RTM's hardware limitations and quantify their effects on application behavior, finding that performance degrades for workloads with large working sets and long transactions. Enabling hyper-threading worsens RTM performance due to resource sharing at the L1 level. We give details about the sources of aborts in a TM application and a way to quantiy these aborts. Using the knowledge presented in this thesis, parallel programmers can optimize TM applications to better utilize the Haswell support for RTM.

6
Conclusion

The significance of energy-efficient computing is growing because of the concerns regarding ecological footprint of the IT sector and the economical effect of operating computers with high power consumption. The feedback from the system about its power and/or energy usage is required to facilitate energy-efficient computing. This thesis proposes methodologies for power measurement and power modeling that can be used by the system and software to enable power-aware decision making. Alternatively, the information about power consumption can be collected, analyzed and correlated with other performance events to characterize various aspects of the system. This characterization data can be used to determine the energy efficiency of the system, make quantitative comparisons between competing hardware or software designs, or provide valuable information to formulate offline/online system policies.

This thesis first analyzes different techniques to measure power consumption and provides a quantitative comparison between these techniques. We sample the power consumption at three different points in the system: wall outlet, ATX power rails and directly on motherboard. Each successive technique is more intrusive. We compare the measurements at these points for accuracy, sensitivity and temporal granularity. The power measurement at wall outlet is most accessible and requires cheapest hardware but suffers from lack of granularity. We find that measuring power consumption at ATX power rails provides the best balance between accuracy, accessibility and granularity. We test the accuracy of Intel's RAPL energy counter against power measurements at ATX power rails and find that, although the RAPL values are fairly accurate, they exhibit irregularities in temporal granularity and comparatively lower sensitivity to temperature. Based upon this comparison, we conclude that RAPL energy counter is a good choice for measuring energy of an application runs for few tens of milliseconds but for instantaneous power trends, actual power measurements are superior to values reported by RAPL.

This thesis proposes two methodologies for estimating power consumption of the chip. The first is a statistical model that uses correlation to select performance events and piece-wise multiple linear regression analysis to assign weights to the selected performance events and temperature values to estimate the power consumption. This modeling methodology is well suited for portability across multiple platforms which we prove by validating the model across six different platforms. We demonstrate the high accuracy of the model across all six platforms (overall median errors per machine between 1.2% and 4.4%). The linear regression equation used for estimating the power in this methodology is suitable for estimating power consumption at runtime. We show its effectiveness for power budgeting by incorporating the model in a user-level meta scheduler. However, this model has few limitations: First, it has limited decomposability. Although it provides power consumption values at the granularity of each individual core, it does not estimate separate power values for the processor uncore. Second, this model does not distinguish between static and dynamic power. As a result, the model has to be retrained when the voltage-frequency operating point on the core is changed. To address these limitations, we propose a second power modeling methodology that uses a mix of analytical and statistical approach to model static and dynamic power consumption for processor uncore and individual cores. We validate this model for multi-threaded and single-threaded benchmark suites across different voltage-frequency operating points on Haswell microarchitecture and show that the model exhibits good accuracy at all levels of parallelism and frequencies (mean error of 3.14% across all experiments).

We use the power model developed using the second approach to characterize the energy efficiency of frequency scaling on Haswell. Our characterization study shows that uncore idle energy can be a significant source of energy inefficiency on Haswell, with uncore static energy contributing up to 61% of total energy expenditure in some cases. As a result of this energy inefficiency, lowering system frequency does not always lower total energy expenditure and the frequency at which lowest energy is expended depends on the memory bound fraction of application. We use this information to develop a low overhead DVFS scheduler that uses memory bound fraction calculated at runtime to a simple look-up in a table created offline to choose the most energy efficient frequency for the current phase of the application. Based upon our experimental results, we conclude that this simplistic DVFS scheduler is almost as effective as some of the more complex schedulers suggested by prior works in choosing energy-efficient frequencies for different phases in an application.

We next use the power model to characterize the energy efficiency of thread scaling. We identify four aspects of communication — serial fraction, core-to-core communication overhead, bandwidth contention and locking mechanism overhead — that can be varied orthogonally to study their effects on the efficiency of thread scaling. We demonstrate how these parameters affect the core/uncore static and dynamic energy trends as the application is scaled from one to more threads.

We next characterize Intel's restricted transactional memory implementation called RTM and compared it with TinySTM, a software transactional memory implementation, in terms of performance and energy. We quantify RTM's limitations and identify the hardware implementation details that result in these limitations. We conclude that RTM can provide better performance gains than STM implementations if the parallel programmer is aware of these hardware limitations. For example, since RTM write-set size is limited by the size of L1 data cache, enabling hyper-threading almost always results in performance degradation due to halving of the write-set size. Similarly, since RTM detects contention at block granularity while STM detects contention at word granularity, for the same level of contention, RTM may experience more aborts. But when the contention is low, RTM outperforms and is more energy efficient than STM implementations. The parallel programmers can use the performance and energy characterization data presented in this thesis to make efficient utilization of RTM.

The work done as part of this thesis can be leveraged to enable future research in multiple directions. The energy characterization on Haswell shows that the power consumed by the idle uncore plays an important role in influencing the energy trends observed during frequency and thread scaling. On a processor that implements more or less aggressive power saving techniques, we expect to see different energy trends. Our methodology for modeling static and dynamic power consumption can be used to quantitatively compare the energy trends across different microarchitectures.

Most of the current research on thread scaling is performance centric and energy savings are generally a side-effect of reduction in execution time. But our analysis shows that energy trends for thread scaling do not necessarily match performance trends. The runtime systems or task schedulers that prioritize reducing energy expenditure over increasing performance can use the insights gained from thread scaling energy characterization done as part of this thesis to formulate energy-aware scaling policies.

The transactional memory characterization shows that cross-over points exist where it becomes more energy-efficient to use HTM over STM or vice-versa. The data gathered from TM characterization can be employed to formulate energy-efficient hybrid TM systems. Such a TM system can employ STM in the fall-back path of RTM and force RTM to abort before the retry threshold is reached if the collected metrics indicate that STM will be more energy-efficient for the current atomic block.

Bibliography

[1] N. Oreskes, "The scientific consensus on climate change," *Science*, vol. 306, no. 5702, pp. 1686–1686, 2004.

[2] Ryan Heath and Linda Cain, "Digital Agenda: global tech sector measures its carbon footprint," `http://europa.eu/rapid/press-release_IP-13-231_en.htm`, 2013.

[3] S. Murugesan, "Harnessing green it: Principles and practices," *IEEE IT Professional*, vol. 10, no. 1, pp. 24–33, 2008.

[4] Gary Cook and Jodie Van Horn, "How Dirty is your Data," *A Look at the Energy Choices That Power Cloud Computing*, 2011.

[5] J. Whitney and P. Delforge, "Data center efficiency assessment," *Natural Resources Defense Council, New York City, New York*, 2014.

[6] V. Khandelwal and A. Srivastava, "Leakage control through fine-grained placement and sizing of sleep transistors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 7, pp. 1246–1255, 2007.

[7] J. T. Kao and A. P. Chandrakasan, "Dual-threshold voltage techniques for low-power digital circuits," *Solid-State Circuits, IEEE Journal of*, vol. 35, no. 7, pp. 1009–1018, 2000.

[8] J. W. Tschanz, S. G. Narendra, Y. Ye, B. A. Bloechel, S. Borkar, and V. De, "Dynamic sleep transistor and body bias for active leakage power control of microprocessors," *Solid-State Circuits, IEEE Journal of*, vol. 38, no. 11, pp. 1838–1845, 2003.

[9] A. Abdollahi, F. Fallah, and M. Pedram, "Leakage current reduction in cmos vlsi circuits by input vector control," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 2, pp. 140–154, 2004.

[10] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1st edition, 1990.

[11] Peter Greenhalgh, “Big. little processing with arm cortex-a15 & cortex-a7,” *ARM White paper*, pp. 1–8, 2011.

[12] N. Magen, A. Kolodny, U. Weiser, and N. Shamir, “Interconnect-power dissipation in a microprocessor,” in *Proceedings of the 2004 international workshop on System level interconnect prediction*. ACM, 2004, pp. 7–13.

[13] Parthasarathy Ranganathan, Sarita Adve, and Norman Jouppi, “Reconfigurable caches and their application to media processing,” in *Proc. 27th IEEE/ACM International Symposium on Computer Architecture*, 2000, pp. 214–224.

[14] A. Settle, D.A. Connors, E. Gibert, and A. Gonzalez, “A dynamically reconfigurable cache for multithreaded processors,” *Journal of Embedded Computing: Special Issue on Single-Chip Multi-core Architectures*, vol. 1, no. 1, pp. 221–233, Dec. 2005.

[15] Adria Armejach, Azam Seydi, Rubén Titos-Gil, Ibrahim Hur, Adrián Cristal, Osman Unsal, and Mateo Valero, “Using a reconfigurable l1 data cache for efficient version management in hardware transactional memory,” in *Proc. 20th International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[16] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, “Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures,” in *Proc. IEEE/ACM 33nd International Symposium on Microarchitecture*, Dec. 2000, pp. 245–257.

[17] G. Keramidas, C. Datsios, and S. Kaxiras, “A framework for efficient cache resizing,” in *Embedded Computer Systems (SAMOS), 2012 International Conference on*. IEEE, 2012, pp. 76–85.

[18] S. Kaxiras, Z. Hu, and M. Martonosi, “Cache decay: Exploiting generational behavior to reduce cache leakage power,” in *Proc. 28th IEEE/ACM International Symposium on Computer Architecture*, June 2001, pp. 240–251.

[19] N.S. Kim, K. Flautner, D. Blaauw, and T. Mudge, “Drowsy instruction caches: Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction,” in *Proc. IEEE/ACM 36th International Symposium on Microarchitecture*, Nov. 2002, pp. 219–230.

[20] Hewlett Packard, Intel, Microsoft, Phoenix, and Toshiba, *Advanced Configuration and Power Interface Specification*, 4.0a edition, apr 2010.

[21] L. A. Barroso and U. Hölzle, “The case for energy-proportional computing,” *Computer*, , no. 12, pp. 33–37, 2007.

[22] Daniel Wong and Murali Annavaram, “Knightshift: Scaling the energy proportionality wall through server-level heterogeneity,” in *Microarchitecture (MICRO),*

2012 45th Annual IEEE/ACM International Symposium on. IEEE, 2012, pp. 119–130.

[23] M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, “Compiler optimizations for low power systems,” in *Power aware computing*, pp. 191–210. Springer, 2002.

[24] B. Rountree, D. H. Ahn, S. De R. Bronis, D. K. Lowenthal, and M. Schulz, “Beyond dvfs: A first look at performance under a hardware-enforced power bound,” in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 947–953.

[25] C. Lefurgy, X. Wang, and M. Ware, “Power capping: a prelude to power shifting,” *Cluster Computing*, vol. 11, no. 2, pp. 183–195, 2008.

[26] Jason Cong and Bo Yuan, “Energy-efficient scheduling on heterogeneous multi-core architectures,” in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*. ACM, 2012, pp. 345–350.

[27] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, “Hierarchical power management for asymmetric multi-core in dark silicon era,” in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 174.

[28] Q. Tang, S.K.S. Gupta, and G. Varsamopoulos, “Energy-efficient thermal-aware task scheduling for homogeneous high-performance computing data centers: A cyber-physical approach,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 11, pp. 1458–1472, 2008.

[29] T. Mukherjee, A. Banerjee, G. Varsamopoulos, S.K.S. Gupta, and S. Rungta, “Spatio-temporal thermal-aware job scheduling to minimize energy consumption in virtualized heterogeneous data centers,” *Elsevier Computer Networks*, vol. 53, no. 17, pp. 2888–2904, 2009.

[30] G. Von Laszewski, L. Wang, A.J. Younge, and X. He, “Power-aware scheduling of virtual machines in DVFS-enabled clusters,” in *IEEE International Conference on Cluster Computing*, 2009, pp. 1–10.

[31] Josep Ll Berral, Íñigo Goiri, Ramón Nou, Ferran Julià, Jordi Guitart, Ricard Gavaldà, and Jordi Torres, “Towards energy-aware scheduling in data centers using machine learning,” in *Proc. the 1st International Conference on energy-Efficient Computing and Networking*. ACM, 2010, pp. 215–224.

[32] D. Jenkins, “Node Manager — a dynamic approach to managing power in the data center,” White Paper, Intel Corporation. `http://communities.intel.com/docs/DOC-4766`, Jan. 2010.

[33] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proc. IEEE/ACM 36th Annual International Symposium on Microarchitecture*, 2003, pp. 93–104.

[34] K. Rajamani, H. Hanson, J. Rubio, S. Ghiasi, and F. Rawson, "Application-aware power management," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, Oct. 2006, pp. 39–48.

[35] Z. Cui, Y. Zhu, Y. Bao, and M. Chen, "A fine-grained component-level power measurement method," in *Proc. 2nd International Green Computing Conference*, July 2011, pp. 1–6.

[36] V. Spiliopoulos, S. Kaxiras, and G. Keramidas, "Green governors: A framework for continuously adaptive DVFS," in *Proc. 2nd International Green Computing Conference*, July 2011, pp. 1–8.

[37] Krishna Rangan, Gu-Yeon Wei, and David Brooks, "Thread motion: Fine-grained power management for multi-core systems," in *Proc. 36th IEEE/ACM International Symposium on Computer Architecture*, June 2009.

[38] M. Banikazemi, D. Poff, and B. Abali, "PAM: A novel performance/power aware meta-scheduler for multi-core systems," in *Proc. IEEE/ACM Supercomputing International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2008, number 39.

[39] C. Isci, A. Buyuktosunoglu, C.Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proc. IEEE/ACM 39th Annual International Symposium on Microarchitecture*, Dec. 2006, pp. 347–358.

[40] Ke Meng, Russ Joseph, Robert P. Dick, and Li Shang, "Multi-optimization power management for chip multiprocessors," in *Proc. the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 177–186.

[41] Kyong Hoon Kim, Rajkumar Buyya, and Jong Kim, "Power aware scheduling of bag-of-tasks applications with deadline constraints on DVS-enabled Clusters," in *CCGRID*, 2007, vol. 7, pp. 541–548.

[42] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan, "Full-system power analysis and modeling for server environments," in *Proc. Workshop on Modeling, Benchmarking, and Simulation*, June 2006.

[43] Xin Zhan and Sherief Reda, "Techniques for energy-efficient power budgeting in data centers," in *Proc. the 50th Annual Design Automation Conference*, 2013, pp. 176:1–176:7.

[44] E. Rotem, A. Naveh, D. Rajwan, A. Ananthadrishnan, and E Weissmann, "Power management architecture of the 2nd generation Intel® Core™ microarchitecture, formerly codenamed Sandy Bridge," in *Proc. 23rd HotChips Symposium on High Performance Chips*, Aug. 2011.

[45] Advanced Micro Devices, "AMD Opteron™ 6200 series processors Linux tuning guide," 2012, `http://developer.amd.com/wordpress/media/2012/10/51803A_OpteronLinuxTuningGuide_SCREEN.pdf`.

[46] NVIDIA, *Nvidia NVML API Reference Manual*, 2012.

[47] Abhinav Pathak, Y Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Proc. ACM SIGOPS/EuroSys European Conference on Computer Systems*, Apr. 2011, pp. 153–168.

[48] R. Joseph and M. Martonosi, "Run-time power estimation in high-performance microprocessors," in *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, Aug. 2001, pp. 135–140.

[49] F. Bellosa, S. Kellner, M. Waitz, and A. Weissel, "Event-driven energy accounting for dynamic thermal management," in *Proc. Workshop on Compilers and Operating Systems for Low Power*, Sept. 2003.

[50] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade, "Decomposable and responsive power models for multicore processors using performance counters," in *Proc. 24th ACM International Conference on Supercomputing*, June 2010, pp. 147–158.

[51] G. Contreras and M. Martonosi, "Power prediction for Intel XScale processors using performance monitoring unit events," in *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, Aug. 2005, pp. 221–226.

[52] K. Singh, M. Bhadauria, and S.A. McKee, "Real time power estimation and thread scheduling via performance counters," in *Proc. Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, Nov. 2008.

[53] R. Ren, J.E. Sanz, and M. Raulet, "System-level PMC-driven energy estimation models in RVC-CAL video codec specifications," in *Proc. IEEE Conference on Design and Architectures for Signal and Image Processing*, Oct. 2013, pp. 8–10.

[54] B.C. Lee and D.M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proc. 12th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006, pp. 185–194.

[55] V. Jimenez, CF. azorla, R. Gioiosa, E. Kursun, C. Isci, A. Buyuktosunoglu, P. Bose, and M. Valero, "Energy-aware accounting and billing in large-scale computing facilities," *IEEE Micro*, vol. 31, no. 3, pp. 60–71, 2011.

[56] Jeonghwan Choi, Chen-Yong Cher, Hubertus Franke, Henrdrik Hamann, Alan Weger, and Pradip Bose, "Thermal-aware task scheduling at the system software level," in *Proc. the 2007 international symposium on Low power electronics and design*. ACM, 2007, pp. 213–218.

[57] Ayse Kivilcim Coskun, Tajana Simunic Rosing, and Keith Whisnant, "Temperature aware task scheduling in mpsocs," in *Proc. the conference on Design, automation and test in Europe*. EDA Consortium, 2007, pp. 1659–1664.

[58] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang, "Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof," in *Proc. the 7th ACM European Conference on Computer Systems*, 2012, EuroSys '12, pp. 29–42.

[59] K.S. Banerjee and E. Agu, "PowerSpy: fine-grained software energy profiling for mobile devices," in *Proc. International Conference on Wireless Networks, Communications and Mobile Computing*, 2005, vol. 2, pp. 1136–1141 vol.2.

[60] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," in *Proc. 2nd IEEE Workshop on Mobile Computing Systems and Applications*, Feb. 1999, pp. 2–10.

[61] Y.S. Shao and D. Brooks, "Energy characterization and instruction-level energy model of intel's xeon phi processor," in *Proc. 2013 IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2013, pp. 389–394.

[62] Ramon Bertran, Alper Buyuktosunoglu, Meeta S. Gupta, Marc Gonzalez, and Pradip Bose, "Systematic energy characterization of CMP/SMT processor systems via automated micro-benchmarks," in *Proc. the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2012, MICRO-45, pp. 199–211, IEEE Computer Society.

[63] T. Mukherjee, G. Varsamopoulos, S. K S Gupta, and S. Rungta, "Measurement-based power profiling of data center equipment," in *2007 IEEE International Conference on Cluster Computing*, 2007, pp. 476–477.

[64] Epifanio Gaona, Rubén Titos-Gil, Juan Fernandez, and Manuel E. Acacio, "Characterizing energy consumption in hardware transactional memory systems," in *Proc. 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2010, pp. 9–16.

[65] Y. Guo, S. Chheda, I. Koren, C. M. Krishna, and C. A. Moritz, "Energy Characterization of Hardware-Based Data Prefetching," in *Proc. IEEE International Conference on Computer Design*, 2004, pp. 518–523.

[66] Marc Gamell, Ivan Rodero, Manish Parashar, Janine C. Bennett, Hemanth Kolla, Jacqueline Chen, Peer-Timo Bremer, Aaditya G. Landge, Attila Gyulassy, Patrick McCormick, Scott Pakin, Valerio Pascucci, and Scott Klasky, "Exploring power behaviors and trade-offs of in-situ data analytics," in *Proc. SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013, SC '13, pp. 77:1–77:12.

[67] John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta, "Evaluating the effectiveness of model-based power characterization," in *Proc. the 2011 USENIX Conference on USENIX Annual Technical Conference*, 2011, pp. 12–12.

[68] K. Singh, *Prediction Strategies for Power-Aware Computing on Multicore Processors*, Ph.D. thesis, Cornell University, June 2009.

[69] B. Goel, S.A. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati, "Portable, scalable, per-core power estimation for intelligent resource management," in *Proc. 1st International Green Computing Conference*, Aug. 2010, pp. 135–146.

[70] C. Sun, L. Shang, and R.P. Dick, "Three-dimensional multiprocessor system-on-chip thermal optimization," in *Proc. 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, Sept. 2007, pp. 117–122.

[71] E. Stahl, "Power Benchmarking: A new methodology for analyzing performance by applying energy efficiency metrics," White Paper, IBM, 2006.

[72] Standard Performance Evaluation Corporation, "SPECpower_ssj2008 benchmark suite," `http://www.spec.org/power_ssj2008/`, 2008.

[73] B. Goel, R. Titos-Gil, A. Negi, S.A. McKee, and P. Stenstrom, "Performance and energy analysis of the restricted transactional memory implementation on haswell," in *Proc. Proc. International Parallel and Distributed Processing Symposium*, 2014, To appear.

[74] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig, "Measuring energy consumption for short code paths using RAPL," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 3, pp. 13–17, Jan. 2012.

[75] Electronic Educational Devices, "Watts Up PRO," `http://www.wattsupmeters.com/`, May 2009.

[76] X. Wang and M. Chen, “Cluster-level feedback power control for performance optimization,” in *Proc. 14th IEEE International Symposium on High Performance Computer Architecture*, Feb. 2008, pp. 101–110.

[77] D. Bedard, M. Y. Lim, R. Fowler, and A. Porterfield, “Powermon: Fine-grained and integrated power monitoring for commodity computer systems,” in *Proc. IEEE SoutheastCon 2010 (SoutheastCon)*, Mar. 2010, pp. 479–484.

[78] Server System Infrastructure Forum, *EPS12V Power Supply Design Guide*, Dell, HP, SGI, and IBM, 2.92 edition, 2006.

[79] LEM Corporation, “Intel current transducer LTS 25-NP,” Datasheet, LEM, Nov. 2009.

[80] National Instruments Corporation, “NI bus-powered m series multifunction daq for usb,” `http://sine.ni.com/ds/app/doc/p/id/ds-9/lang/en`, Apr. 2009.

[81] Intel Corporation, “Voltage regulator-down (VRD) 11.1,” Design Guidelines, Intel Corporation, Sept. 2009.

[82] International Rectifier, “Digital multi-phase buck controller chl8316, v1.02,” `http://www.irf.com/product-info/datasheets/data/pb-chl8316.pdf`, Oct. 2012, [Online; accessed 10-May-2016].

[83] D. Hackenberg, R. Schone, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, “An energy efficiency feature survey of the intel haswell processor,” in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 896–904.

[84] Intel, *Intel Architecture Software Developer’s Manual: System Programming Guide*, June 2013.

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

[86] R. Ge, X. Feng, S. Song, H. Chang, D. Li, and K. W. Cameron, “Powerpack: Energy profiling and analysis of high-performance systems and applications,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 5, pp. 658–671, 2010.

[87] M.S. Floyd, S. Ghiasi, T.W. Keller, K. Rajamani, F.L. Rawson, J.C. Rubio, and M.S. Ware, “System power management support in the IBM POWER6 microprocessor,” *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 733–746, 2007.

[88] H.-Y. McCreary, M.A. Broyles, M.S. Floyd, A.J. Geissler, S.P. Hartman, F.L. Rawson, T.J. Rosedahl, J.C. Rubio, and M.S. Ware, "Energyscale for IBM POWER6 microprocessor-based systems," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 775–786, 2007.

[89] V.M. Weaver and S.A. McKee, "Can hardware performance counters be trusted?," in *Proc. IEEE International Symposium on Workload Characterization*, Sept. 2008, pp. 141–150.

[90] D. Zaparanuks, M. Jovic, and M. Hauswirth, "Accuracy of performance counter measurements," Tech. Rep. USI-TR-2008-05, Università della Svizzera italiana, Sept. 2008.

[91] V.M. Weaver and J. Dongarra, "Can hardware performance counters produce expected, deterministic results," in *In Proc. 3rd Workshop on Functionality of Hardware Performance Monitoring*, Dec. 2010.

[92] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2013.

[93] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *IEEE Computer*, vol. 36, no. 12, pp. 68–75, Dec. 2003.

[94] M. Berktold and T. Tian, "CPU monitoring with DTS/PECI," White Paper 322683, Intel Corporation. `http://download.intel.com/design/intarch/papers/322683.pdf`, Sept. 2010.

[95] C. Spearman, "The proof and measurement of association between two things," *The American Journal of Psychology*, vol. 15, no. 1, pp. 72–101, Jan. 1904.

[96] Standard Performance Evaluation Corporation, "SPEC CPU benchmark suite," `http://www.specbench.org/osg/cpu2006/`, 2006.

[97] Standard Performance Evaluation Corporation, "SPEC OMP benchmark suite," `http://www.specbench.org/hpg/omp2001/`, 2001.

[98] V. Aslot and R. Eigenmann, "Performance characteristics of the SPEC OMP2001 benchmarks," in *Proc. European Workshop on OpenMP*, Sept. 2001.

[99] D.H. Bailey, T. Harris, W.C. Saphir, R.F. Van der Wijngaart, A.C. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[100] S. Eranian, "Perfmon2: A flexible performance monitoring interface for Linux," in *Proc. 2006 Ottawa Linux Symposium*, July 2006, pp. 269–288.

[101] C. Boneti, R. Gioiosa, F.J. Cazorla, and M. Valero, "A dynamic scheduler for balancing HPC applications," in *Proc. IEEE/ACM Supercomputing International*

Conference on High Performance Computing, Networking, Storage and Analysis, Nov. 2008, number 41.

[102] E. Betti, M. Cesati, R. Gioiosa, and F. Piermaria, "A global operating system for HPC clusters," in *Proc. IEEE International Conference on Cluster Computing*, Aug. 2009, p. 6.

[103] Advanced Micro Devices, *AMD Athlon Processor Model 6 Revision Guide*, 2003.

[104] V. Tivari, S. Malik, A. Wolfe, and M.T.-C. Lee, "Instruction level power analysis and optimization of software," *Kluwer Journal of VLSI Design*, vol. 13, no. 3, pp. 223–238, 1996.

[105] J.T. Russell and M.F. Jacome, "Software power estimation and optimization for high-performance 32-bit embedded processors," in *Proc. IEEE International Conference on Computer Design*, Oct. 1998, pp. 328–333.

[106] T. Cignetti, K. Komarov, and C.S. Ellis, "Energy estimation tools for the Palm," in *Proc. ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, Aug. 2000, pp. 96–103.

[107] A. Merkel and F. Bellosa, "Balancing power consumption in multicore processors," in *Proc. ACM SIGOPS/EuroSys European Conference on Computer Systems*, Apr. 2006, pp. 403–414.

[108] M. Moudgill, P. Bose, and J. Moreno, "Validation of Turandot, a fast processor model for microarchitecture exploration," in *Proc. International Performance, Computing, and Communications Conference*, Feb. 1999, pp. 452–457.

[109] B. Su, J. Gu, L. Shen, W. Huang, J.L. Greathouse, and Z. Wang, "PPEP: Online performance, power, and energy prediction framework and DVFS space exploration," in *Proc. IEEE/ACM 47th Annual International Symposium on Microarchitecture*, Dec. 2014, pp. 445–457.

[110] R. Ge, X. Feng, W. Feng, and K.W. Cameron, "CPU MISER: A performance-directed, run-time system for power-aware clusters," in *Proc. International Conference on Parallel Processing*, Sept. 2007, pp. 18–26.

[111] A. Marathe, P. E. Bailey, D. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "A run-time system for power-constrained hpc applications," in *High Performance Computing*. Springer, 2015, pp. 394–408.

[112] T. Patki, D. K. Lowenthal, A. Sasidharan, M. Maiterth, B. Rountree, M. Schulz, and B. R. de Supinski, "Practical resource management in power-constrained, high performance computing," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 121–132.

[113] J. Mair, D. Eyers, Z. Huang, and H. Zhang, “Myths in power estimation with Performance Monitoring Counters,” *Elsevier Sustainable Computing: Informatics and Systems*, vol. 4, no. 2, pp. 83–93, June 2014.

[114] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun, “Eigenbench: A simple exploration tool for orthogonal TM characteristics,” in *Proc. the IEEE International Symposium on Workload Characterization*, 2010, pp. 1–11.

[115] B. Goel, S.A. McKee, and M. Själander, *Techniques to Measure, Model, and Manage Power*, vol. 87 of *Advances in Computers*, chapter 2, pp. 7–54, Elsevier, Nov. 2012.

[116] R. B. Hall, *Thin Solid Films*, vol. 8, chapter 2, pp. 263–271, Elsevier, Oct. 1971.

[117] 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 *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 224–234.

[118] A. Dzhagaryan and A. Milenković, “Impact of thread and frequency scaling on performance and energy in modern multicores: a measurement-based study,” in *Proceedings of the 2014 ACM Southeast Regional Conference*. ACM, 2014, p. 14.

[119] Maurice Herlihy and J. Eliot B. Moss, “Transactional memory: Architectural support for lock-free data structures,” in *Proc. 20th International Symposium on Computer Architecture*, 1993, pp. 289–300.

[120] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood, “LogTM-SE: Decoupling hardware transactional memory from caches,” in *Proc. 13th Symposium on High-Performance Computer Architecture*, 2007, pp. 261–272.

[121] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun, “Transactional memory coherence and consistency,” in *Proc. 31st International Symposium on Computer Architecture*, 2004, pp. 102–113.

[122] Marc Lupon, Grigorios Magklis, and Antonio González, “A dynamically adaptable hardware transactional memory,” in *Proc. 43rd International Symposium on Microarchitecture*, 2010, pp. 27–38.

[123] Rubén Titos-Gil, Anurag Negi, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom, “Zebra : A data-centric, hybrid-policy hardware transactional memory design,” in *Proc. 25th International Conference of Supercomputing*, 2011, pp. 53–62.

[124] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum, “Early experience with a commercial hardware transactional memory implementation,” *ACM SIGPLAN Notices*, vol. 44, no. 3, pp. 157–168, Mar. 2009.

[125] Cliff Click, “Azul’s experiences with hardware transactional memory,” 2009, http://sss.cs.purdue.edu/projects/tm/tmw2010/talks/Click-2010_TMW.pdf.

[126] Amy Wang, Matthew Gaudet, Peng Wu, Jose Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael, “Evaluation of Blue Gene/Q hardware support for transactional memories,” in *Proc. the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 127–136.

[127] Christian Jacobi, Timothy Slegel, and Dan Greiner, “Transactional memory architecture and implementation for IBM System z,” in *Proc. the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 25–36.

[128] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar, “Performance evaluation of Intel transactional synchronization extensions for high-performance computing,” in *Proc. SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 19:1–19:11.

[129] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun, “STAMP: Stanford transactional applications for multi-processing,” in *Proc. IEEE International Symposium on Workload Characterization*, 2008, pp. 35–46.

[130] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel, “Time-based software transactional memory,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.

[131] Intel, *Desktop 4th Generation Intel Core^{TM}Processor Family Specification Update*, Aug. 2013.

[132] M Schindewolf, “Performance analysis of and tool support for transactional memory on bg/q,” Tech. Rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2011.

[133] David Dice, Ori Shalev, and Nir Shavit, “Transactional Locking II,” in *Proc. the 19th International Symposium on Distributed Computing*, 2006.

[134] Zhaoguo Wang, Hao Qian, Haibo Chen, and Jinyang Li, “Opportunities and pitfalls of multi-core scaling using hardware transaction memory,” in *Proc. the 4th Asia-Pacific Workshop on Systems*, 2013, pp. 3:1–3:7.

[135] E. Gaona-Ramirez, R. Titos-Gil, J. Fernandez, and M.E. Acacio, “Characterizing energy consumption in hardware transactional memory systems,” in *Proc.*

22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2010, pp. 9–16.

[136] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun, "A scalable, non-blocking approach to transactional memory," in *Proc. 13th Symposium on High-Performance Computer Architecture*, 2007, pp. 97–108.

[137] C. Ferri, R.I. Bahar, A. Marongiu, L. Benini, M. Herlihy, B. Lipton, and T. Moreshet, "SoC-TM: Integrated HW/SW support for transactional memory programming on embedded MPSoCs," in *Proc. the 9th International Conference on Hardware/Software Codesign and System Synthesis*, 2011, pp. 39–48.