Towards Energy Efficient Computing with Linux : Enabling Task Level Power Awareness and Support for Energy Efficient Accelerator

by

Digant Pareshkumar Desai

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved May 2013 by the
Graduate Supervisory Committee:

Sarma Vrudhula, Chair
Chaitali Chakrabarti
Carole-Jean Wu

ARIZONA STATE UNIVERSITY

August 2013

ABSTRACT

With increasing transistor volume and reducing feature size, it has become a major design constraint to reduce power consumption also. This has given rise to aggressive architectural changes for on-chip power management and rapid development to energy efficient hardware accelerators. Accordingly, the objective of this research work is to facilitate software developers to leverage these hardware techniques and improve energy efficiency of the system. To achieve this, I propose two solutions for Linux kernel:

Optimal use of these architectural enhancements to achieve greater energy efficiency requires accurate modeling of processor power consumption. Though there are many models available in literature to model processor power consumption, there is a lack of such models to capture power consumption at the task-level. Task-level energy models are a requirement for an operating system (OS) to perform real-time power management as OS time multiplexes tasks to enable sharing of hardware resources. I propose a detailed design methodology for constructing an architecture agnostic task-level power model and incorporating it into a modern operating system to build an online task-level power profiler. The profiler is implemented inside the latest Linux kernel and validated for Intel Sandy Bridge processor. It has a negligible overhead of less than 1% hardware resource consumption. The profiler power prediction was demonstrated for various application benchmarks from SPEC to PARSEC with less than 4% error. I also demonstrate the importance of the proposed profiler for emerging architectural techniques through use case scenarios, which include heterogeneous computing and fine grained per-core DVFS.

Along with architectural enhancement in general purpose processors to improve energy efficiency, hardware accelerators like Coarse Grain reconfigurable architecture (CGRA) are gaining popularity. Unlike vector processors, which rely on data parallelism, CGRA can provide greater flexibility and compiler level control making it more suitable for present

SoC environment. To provide streamline development environment for CGRA, I propose a flexible framework in Linux to do design space exploration for CGRA. With accurate and flexible hardware models, fine grained integration with accurate architectural simulator, and Linux memory management and DMA support, a user can carry out limitless experiments on CGRA in full system environment.

DEDICATION

To my parents, for their unconditional love and also to Pooja for her support in every step of this journey.

ACKNOWLEDGEMENTS

Firstly I want to thank Dr. Vrudhula for giving me this opportunity and for his steadfast support to make this project successful. I also want to thank Dr. Wu and Dr. Chakarabarti for their encouragement and aid without which this thesis would not have been completed. A special thanks to Vinay Hanumiah, who helped me continuously throughout the masters journey. Vinay, you are truly an outstanding person and an able educator and I thank you from the bottom of my heart. I specially want to thank Mahdi Hamzeh for first hiring me and nurturing me through innumerable advices. I will never forget all the discussions, arguments and all the good times I have had in lab with Mahdi, Benjamin Gaudette, Niranjan Kulkarni, and Nishant Nakula. I also wish to thank all my friends, Rashmin Patel, Hardik Mehta, Urvish Mahida, Himanshu Patel, Sudip Dandnaik, Khushboo Dave and Harsh Vachhani for their continuous encouragement and support.

I want to thank my funding agencies SFAZ and NSF. I would like to give a special acknowledgements to the Center of Embedded System(CES) at Arizona State University and everyone associated with it, especially Dr. Vrudhula and Lisa Christen for providing me this opportunity and the financial support. This work was funded by SFAZ and CES through research grant.

I also want to thank Intel Corporations, Ravi Iyer and his team Li Zhao, Srihari Makineni and Steve King for giving me an opportunity to work with them at Intel Labs, one of the most prestigious place to be an intern. I would also want to thank Chris Lucero and Akella Chakarabarti for providing me guidance and helping me with cutting edge research equipments. I could never have done this without all this support from Intel.

Lastly, I want to thank all my family members for their endless love, support and encouragement through all the difficult times.

TABLE OF CONTENTS

PAGE

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Energy-efficiency and low-power operation are no more second class constraints in the design and operation of processors, especially for the battery-limited mobile and the embedded systems. At the system-level, for such devices, high energy-efficiency of an operation is expected at every node of computation. Towards this, I propose two system level enhancements. First, for today's energy-efficient multi core processors I propose a methodology to accurately estimate task-level power consumption. Using such a task profiler, sophisticated energy aware algorithms can be developed to bridge the gap in energy efficiency between the system level software and the architecture. Next, I introduce an extensible, hardware-software simulation platform to perform design space explorations on accelerator design and programming model combinations for accelerator rich, future system-on-chip architecture. This framework provides a coarse grain hardware accelerator support with in contemporary Linux OS.

The reminder of this thesis is divided in two parts:

(i) Task-level Power Profiler and

(ii) Hardware-Software Co-Design Framework for CGRA.

1.1 Need for a Task-level Power Profiler

Enabling low power operation to improve the energy efficiency of processors has become the key challenge of processor design in every market segment – including battery-powered mobile phones and laptops, desktops and high performance servers in data centers. A key and essential enabling technology is the availability of accurate models of power consumption. Over the years, manufacturers have developed very detailed register-transfer level (RTL) processor power models that are used at early design stages. However, the computational complexity of such models is too high for them to be used in real-time dynamic power management. To enable accurate dynamic control of the

power consumption of processors, there is a need for simpler and more abstract power models that are of sufficiently low computational complexity so that they can be incorporated into an operating system's (OS) scheduler.

There are many benefits of task-level power profiling. Task-level power profiling helps in gauging the energy demands of the tasks under execution, which can either be used to decide the appropriate voltage-frequency states to limit the execution power consumption, or to migrate the tasks to appropriate cores for reducing the hot spots and for balancing the execution load in a multicore environment.

A processor's power consumption can be profiled at various levels – for an entire processor (a single power value), or for individual cores, or individual tasks. Task-level power profiles would be the most beneficial but they are also the most difficult to develop both with respect to accuracy and computational complexity. The problem becomes even more difficult because of sharing of processor components such as caches, buses, memory bandwidth etc among many different tasks.

1.2 Task Power Profiler: Key Contributions

In this thesis, I propose an effective methodology of constructing an energy model for a multicore processor based on a single total processor power measurement, and available processor event recordings and temperatures. The events used in building the model include P-state, T-state, instruction-per-cycle (IPC), and cache related events. P (or performance)-state control is a global dynamic voltage and frequency (DVFS) control for all cores, while T (or throttle)-state modulates the duty cycle of clock on a per-core basis. T-state control can be considered as dynamic frequency scaling (DFS) for all practical purposes. Using this energy model, we build a task profiler in the Linux OS, which periodically samples the energy consumption of the processor, and with the knowledge of the current tasks in execution, accurately estimates power consumption of all tasks. Figure 1.1 shows the complete flow of identification and integration of models within

Linux OS.

The model identification process followed in this work is loosely based on factorial design-of-experiments (DoE) method. In this method, based on the knowledge of the processor, I identify several parameters that help in the estimation of the task power. A set of control experiments is conducted on the processor, where each of these parameters is varied in conjunction with other parameters to determine the joint effect of several parameters on the total power consumption. The relation between power consumption and the factors is expressed as multi variate polynomials whose coefficients are estimated using least squares. The overview of the model identification process is illustrated in Fig. 1.1(B)

There are several advantages to the proposed power profiler: **(i)** The methodology requires little to no knowledge of the processor architecture. As such, the model identification techniques are portable across various processors. **(ii)** There is no need for a priori knowledge of the workload under execution. **(iii)** The model identification process does not affect the normal workload execution as it is conducted usually at the boot time or at very rare intervals. **(iv)** The task-level power profiler is extremely light-weight and its overhead is negligible.

The derived models are integrated inside a dynamic task profiler. Being part of the OS scheduler, the proposed task profiler is capable of estimating power of every individual task based on its performance and thermal behaviour. The proposed methodology is validated on a real state-of-the-art machine. With negligible average overhead of less than 1%, the profiler shows 99.95% accuracy in estimating performance and greater than 96% accuracy in estimating power.

Finally, I demonstrate two valuable uses of the task-level profiler: **(1)** mapping tasks to cores on a heterogeneous platform (big vs little cores) that is emulated on the Intel

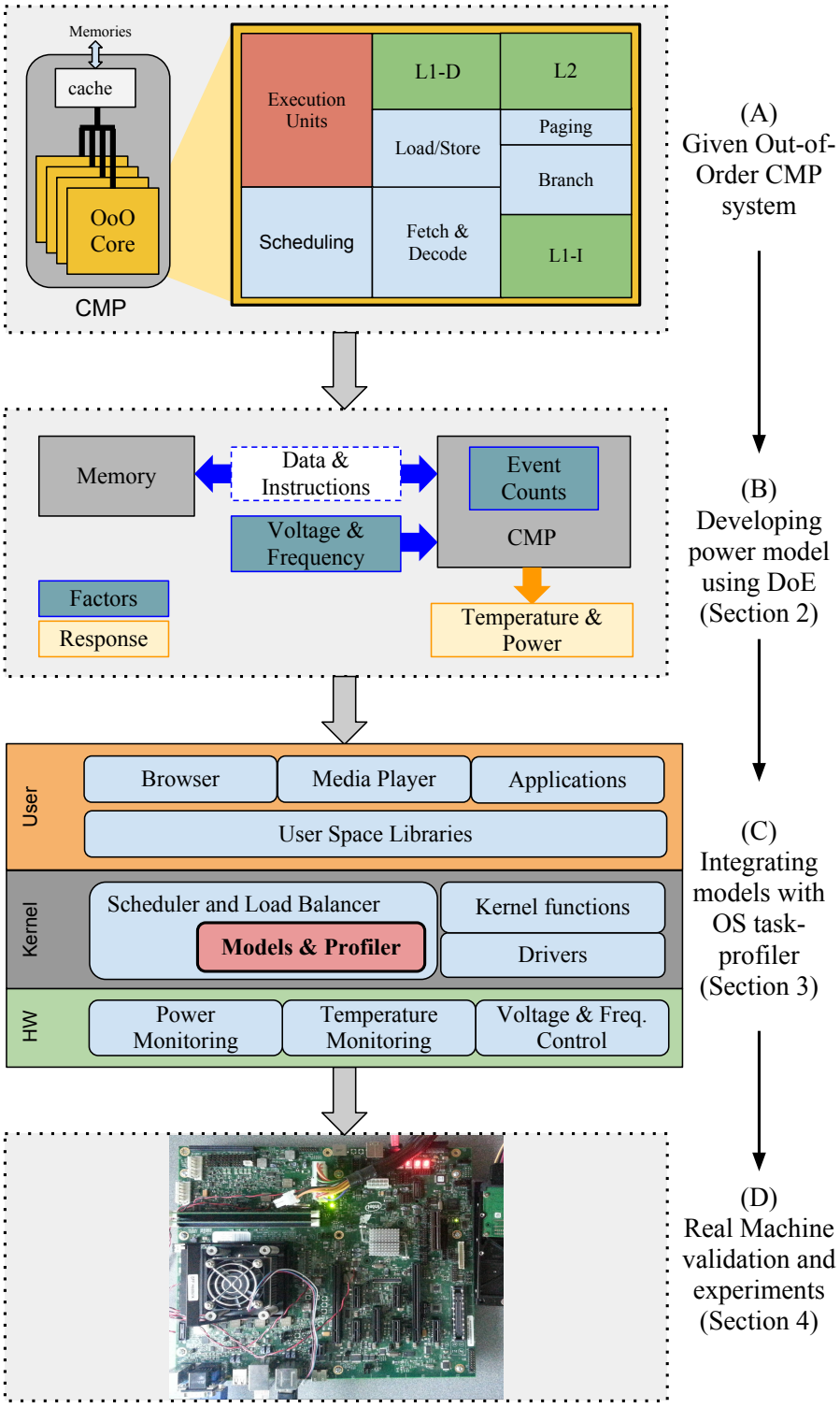

Figure 1.1: Overview of the model identification and integration within OS

Sandy Bridge quad core processor, and **(2)** capability of the task-level voltage and frequency scaling to achieve better *energy efficiency* (performance-per-watt measured by MIPS/Watt) on a heterogeneous platform.

1.3 Need of a Hardware-Software Co-Design Framework for CGRA

Rapid growth of fixed-function hardware accelerators in an SoC environment has led to energy efficient, powerful embedded devices. Even though IP based accelerators can deliver high throughput with a good energy efficiency, their non programmable nature raises doubts about their usefulness in future. Creating special on-chip-networks and a lack of standards in their programming model, restricts the system functionalities. For example, performing a common task like video play back which requires video decryption, decoding and display. Chaining the accelerator and managing memory bandwidth and other resources dynamically to perform video playback creates additional overhead in terms of runtime power. Moreover, increasing chip area and power consumption due to addition of more accelerator IPs on chip is a growing concern among SoC architects. The need for a programmable accelerator is more than ever before which can deliver high throughput at almost the same energy efficiency as regular hardware IP blocks is greater than ever before. A CGRA is a hardware accelerator that can replace more than one fixed-function hardware IP block with nearly the same energy efficiency and performance but with great functional flexibility.

Fig 1.2, shows the relative power consumption and performance envelop for various popular architectural paradigms. Note that the size of the block in the figure represents its programmability. A larger block size indicates greater flexibility to perform different user programmed functions. A general purpose processor and a FPGA, which can execute almost all functions are the largest and ASICs or hardware IPs are the smallest with fixed-function functionality. CGRAs are programmable and are highly energy efficient. Consequently, they hold a greater promise for enabling high performance

energy efficient computing. This makes CGRA a very attractive alternative for the fixed-function accelerators for future SoCs.

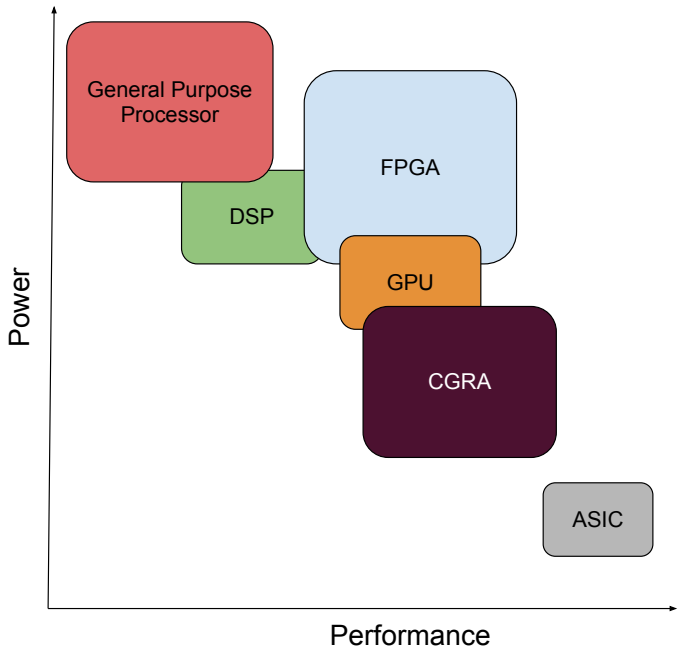

Figure 1.2: Comparison of power and performance envelop for different architectural designs, Size of the block for each design reflects its programming flexibility

The concept of CGRA as programmable accelerator predates GPUs. However the lack of an efficient, optimized compiler prevented its wide spread adoption. The absence of hardware platforms with a CGRA further hindered the development of CGRA. The second part of my thesis is aimed at filling this gap. It includes a hardware-software co-design framework for the CGRA that will allow hardware designers to alter the hardware models and software developers to design a more suitable programming model.

1.4 Hardware-Software Co-Design Framework for CGRA : Key Contribution

The proposed framework provides users a flexible infrastructure to **(i)** perform hardware design space explorations and **(ii)** early software development including developing application libraries, OS support, and programming model changes. Hardware-software co-design can greatly reduce development and design exploration time. The work here

also demonstrates a harmonious coordination of the entire hardware software stack with complete OS and peripheral DMA support. It is also flexible to support power and thermal model and can also be easily extended to support industrial standards like OpenCL [28].

1.5 Thesis Structure

The remainder of this thesis is organized as follows.

(PART 1): Task-level Power Profiler: Chapter 2 contains background and related work in this research area. Chapter 3 discusses the design of experiments (DoE) methodology used to derive the power model. Chapter 4 describes the actual power and thermal model used within the infrastructure. Chapter 5 describes the task profiler and its implementation inside the Linux operating system. Chapter 6 contains results of experiments to validate the models and the task profiler. Chapter 7 demonstrates the effectiveness of such framework through application use-cases.

(PART 2): Hardware-Software Co-Design Framework for CGRA: Similar to part 1, part 2 begins with the related research work in Chapter 8. The next chapter explains the design and the components of the proposed framework in detail. Once the framework is explained in detail Chapter 10 demonstrates how to use the framework with a sample application and results. Lastly, Chapter 11 concludes the thesis.

Chapter 2

BACKGROUND AND RELATED WORK

In this chapter, we examine the related research work on task level power profiling. Since the implementation of the task level power profiler is done within the Linux OS, I will first describe the major components of the Linux OS. We will focus on Linux internals that deal with the ”Completely Fair Scheduler(CFS)” and the ”performance counter subsystem” for chip multiprocessors.

2.1 Linux Scheduling

Fairness

The main goal of the ”Completely Fair Scheduler (CFS)” in Linux is to realise ”ideal” CPU sharing by allocating the execution time on the CPU in proportion to the task priorities. Let delta_exec denote the CPU time allocated to a task where there is no need to share the CPU i.e. when there are no other tasks. Let *nice* denote the task priority assigned by the user. For the historical reasons, the value of *'nice'* lies within [-20,19]. More negative *'nice'* value indicates a preference for higher priority. The weight of task i is given by,

$$w_i = \left\{ -20 \leq nice \leq 19 : NICE\_0\_LOAD \cdot \left(1.25^{-(nice)}\right) \right\} \tag{2.1}$$

The default value of *'nice'* is '0' in which case $w_i = NICE\_0\_LOAD$, and which defines the meaning of $NICE\_0\_LOAD$. Its value can by some factor of 2 and assumed to be 1024.

CFS computes a *virtual runtime*, based on the task weight. The virtual run-time indicates how much CPU resources a given task has consumed. If a task's virtual runtime is low, the CFS will try to increase its share of the CPU usage. The default value, when *'nice'*=0 and therefore $w_i = NICE\_0\_LOAD$, will be delta_exec.

In general the virtual runtime is given by

$$t_{virtual\_runtime} += \frac{delta\_exec}{w_i}(NICE\_0\_LOAD). \tag{2.2}$$

CFS attempts to achieve "fairness" by ensuring that all tasks have approximately the same virtual runtime over the long run and tries to achieve Generalised Processor Sharing(GPS) [20]. Thus, given a choice of tasks to run in the next epoch, CFS selects that task with the lowest virtual runtime.

The share of a time interval $[t_1, t_2]$ of a runnable task i is computed as

$$S_i(t_1, t_2) = \frac{w_i}{\sum_{\forall j \in R} w_j}(t_2 - t_1). \tag{2.3}$$

where S_i is in seconds, R is a the set of all runnable tasks that are currently in run queue of a core. Two additional parameters are also defined for scheduling purposes. One is *sched_latency_ns* which represents the scheduling epoch, and the other is *sched_min_granularity_ns*, which denotes the minimum pre-emption time.

Interactivity

Quick response time is expected for interactive task and it can be achieved through low schedule latency. The OS assumes that the interactive tasks sleep more frequently. Heuristics based on this assumption gives rewards to such tasks to achieve good response time. However, maintaining fairness with interactive tasks is challenging. An interactive task should not take advantage of such rewards and should not starve other non interactive tasks or vice versa.

CFS gives rewards to newly activated tasks by adjusting their virtual runtime to achieve better interactive experience. It ensures small latencies for interactive tasks. The virtual runtime is adjusted by

$$t_{virtual\_runtime} -= \frac{sysctl\_sched\_latency}{cfs\_rq \rightarrow load}(NICE\_0\_LOAD). \tag{2.4}$$

where *sysctl_sched_latency* is scheduling epoch and $cfs\_rq \rightarrow load$ is scheduler run queue load. CFS ensures fairness by adjusting the virtual runtime.

From equation (2.4), it can be seen that the virtual runtime of a task is reduced once it wakes up. This adjustment in virtual runtime is reflected as increased priority while allocating CPU since the sleeper task's new virtual runtime will be lower then other tasks in the run-queue which are running without sleeping frequently.

Load balance

In a CMP, system load balancing is an important objective to maintain a high level of system performance. Load balancing in CMPs maintains fairness among cores at a coarser level then a task level scheduler. It is handled by the CFS subsystem in two ways: (i) passive balancing and (ii) active balancing. Passive balancing attempts to move tasks if there is an imbalance among cores in terms of run queue weight. However, since passive load balancing is respects task priorities ,it can fail to strike a balance. Unlike passive load balancing, active load balancing moves one task from a busy CPU to an idle CPU without comparing priorities.

2.2 Linux perf_events subsystem

This section discusses the implementation of the *perf_event* subsystem in the latest Linux OS. The *perf_event* subsystem is a software module within Linux that is responsible for initiating and managing hardware performance subsystem at low level. In newer chips, hardware designers have added much more performance measurement and evaluation support. Software developers can utilize the new features for both static as well as run time optimization. Linux developers have developed this subsystem to permit applications profiling support in the user space. It allows users to attach hardware events to the task

and record the event. It allows mapping of events to counters, and sharing of counters among different events. By sampling and remapping counters at every context switch makes every hardware event practically private to each task. This is also known as a *vitalization* of hardware performance counters. A user can use this subsystem through a single system call '*perf_event_open()*'. The main task of this subsystem is to handle time multiplexing of events on performance counters. Time multiplexing is required when the number of events to be monitored exceeds the number of available counters. This subsystem also handles counter overflow and other interrupts. Providing an architecture independent software layer for hardware performance counters and maintaining software level events are few of the many features of the *perf_events* subsystem.

The *perf_event* subsystem is not suitable for use inside the kernel. If used inside the kernel for any event mapped to any counter, it blocks that event and counter for user space. Inspite of lack of documentation, the *perf_event* can be used to handle performance counters within the kernel fairly easily. The *perf_event* maintains different performance monitoring units (PMUs) including the kernel's general purpose PMU and other hardware PMUs. The event attribute data structure is used to allocate and configure an event, which can be software or hardware event. Once this event is configured with the *perf_event*, it can be attached to either task or a core. If configured in a task mode, the *perf_event* handles the event remapping when a task migrates through CPUs. Here for our experiments we use the perf_events extensively for task profiling inside the kernel with minimal overhead.

2.3 Current State-of-the-Art

Compiler based task profiling is a widely established method used to understand the behavior of a program. However it is not suited for capturing the dynamic behavior of the program in terms of performance and power consumption. To enable developers to more accurately characterize the code behavior, vendors have started to provide more hardware support to enable dynamic profiling of programs. This has resulted in the development of

many tools for dynamic performance profiling of tasks. Similarly, power and thermal profiling of a task is also gaining popularity for low power, battery operated devices as well as for the power-hungry servers. The increasing demand for power management and the increasing hardware support has resulted in a large body of work on power profiling at the system and task levels. The existing approaches can be classified into two broad categories: (1) those based on hardware performance counters and (2) those that employ system calls.

Performance counter based approaches

Isci et al. [12] proposed a methodology to estimate chip power using models based on the utilization of functional units, which are monitored by counters. However, their method is not applicable for task-level power profiling. Moreover, no accounting for thermal variations is present. This is extremely important as temperature directly and significantly affects the leakage power, which in turn raises the temperature. One of the earliest attempts at task-level power estimation was *PowerScope* [5]. This was an external application running on a separate unit that measured the power consumption of tasks. Although this method is not practical for present day processors, it does demonstrate the benefits of maintaining power profiles of tasks even with limited information about task behavior. The approach described in [11] uses performance counters to classify a task into six categories, depending on the ratio of memory access to total number of instructions. The result is a very coarse level assignment of power values to tasks, which can result in rapid switching between DVFS states, and sub-optimal control. It can also interfere with the hardware allocation scheme on a per-task basis performed by the OS during task scheduling. Merkel et al. in [16], added the capability to the OS scheduler to monitor performance counters, and proposed exponential smoothing over performance counter values to avoid glitches due to very short scheduling periods. However, their work lacks detailed power models that would be required to account for the effect of temperature on

leakage.

System call based alternatives

An alternative approach to assigning energy values to a task is to trace the system calls it generates, and use the energy usage per system call to estimate the energy usage of the task [23, 24]. However, obtaining complete knowledge of power consumption of every system call is not usually possible without manufacturer's support. Moreover, this approach is not sufficiently accurate for present day multicore processors, which share hardware resources among tasks and cores.

An energy profiler for smart-phones was proposed in [33]. The power model was based on the voltage output of a smart-phone battery and was able to characterize the entire system based on the utilization of on-chip peripherals like GPS, WIFI, LCD, etc. Kansal et al. [13] propose a technique for fine-grained, task-level energy profiling for power-aware application design. Use of this technique is limited only to application development and not suitable for runtime, dynamic characterization of tasks. Profiling in this work is done by converting the application resource usage information to energy data using the power specifications of the resources used. The primary focus in references [13, 23, 24, 33] was mainly on overall system power usage and did not consider any detailed models for the CPU.

Uniqueness of the proposed work

None of the above mentioned works describe the effect of tasks sharing hardware resources on power consumption. Sharing of hardware resources is unavoidable due to OS schedulers, which use time multiplexing for tasks. Furthermore, the above works do not account for the interdependency between temperature and leakage power at the task-level. This relationship is of increasing importance as feature sizes shrink. Ignoring the dependency leads to suboptimal DVFS, resulting in significant performance loss or

violation of thermal constraints. These limitations are very critical for on-line power or thermal management using model based estimation. The method presented in this paper is aimed at correcting this deficiency in the existing approaches.

Chapter 3

EXPERIMENTAL DESIGN METHODOLOGY

A task-level *model* of power consumption expresses its power as some computable function of various dependent factors. I use the method of *design of experiments* (DoE) to quantify the effect of each factor on the power consumption. In this section, I give a brief introduction to the DoE concepts that are used in this work in deriving power models for complex out-of-order processors.

3.1 Factorial Design Experiments

Several methods exist for DoE in the literature that aid in modelling the behaviour of systems. One of the simplest and intuitive ways is the *one-factor-at-a-time* experiment [19]. In this experiment, only one control input is varied while maintaining all other control inputs at some constant value. The effect of the controlled input on the output is then observed to derive the model. Although this simplistic approach requires fewer experiments, it clearly misses the effect of correlations among the factors on the output. This approach would not be accurate for processors because their power consumption is the result of complex interactions of various parameters and the changing phases of the workload.

Another commonly used alternative method is the *factorial design* [19], where various factors of a system are varied together directly or indirectly through control inputs. With such an approach, one can devise a series of experiments to enumerate all possible combinations of discrete levels of all factors. An experiment can be defined as a test or a series of tests in which purposeful changes are made to the input variable of the process or system so that we may observe and identify the reasons for changes that may be observed in the output response [19]. Today, many experimental design methodologies exist, requiring no knowledge of a system to a good understanding of a system behaviour. In this experiment, which requires control of various components of a chip multiprocessor

system, a fair amount of knowledge of system architecture and factors affecting its power consumption and heat generation is assumed. With this, we need to narrow down on a strategy of experiments involving analysis of the key factors affecting system's power and temperature. A factor is an independent input variable which can be controlled to observe changes in the output. One popular approach is to carry out the experiment by controlling one factor at a time. The major limitation of this approach is the failure to observe interaction between the factors. It is evident that the one factor at a time strategy is not of particular interest to us especially when the factors of system have strong interactions. For example, there is a clear relation between the frequency of the core and its instruction execution rate. Another popular approach and suitable for our system is factorial design, where factors are varied together instead of one at a time. This enables us to observe not only effectiveness of individual factors but also the interaction between them. This method can be used for a system with any number of factors. However, the total number of experiments grow exponentially with increasing number factors. $levels^{factors}$ indicates the full factorial design can be infeasible where there are more levels and factos. In such cases, a fractional factorial design will be a better choice, as it omits certain discrete levels of factors, resulting in a significantly reduced time for experimentation.

For better understanding of the factorial design, we will consider an example system shown in Figure 3.1. As illustrated in Figure 3.1(a), the system has three independent control variables or factors. Each of the factors, A, B, and C can take two levels. The objective here is to design a method to carry out a series of experiments in order to evaluate the effect of each factor on the output with known input. If we follow factorial design, we will have to perform a series of experiments as shown in Figure 3.1(b). Each node of the cube represents an experiment configuration with the level selected for each factor. This particular type of factorial experiment is called "2^3 *factorial design*" leading to 8 different experiments on the system to evaluate the effect of all 3

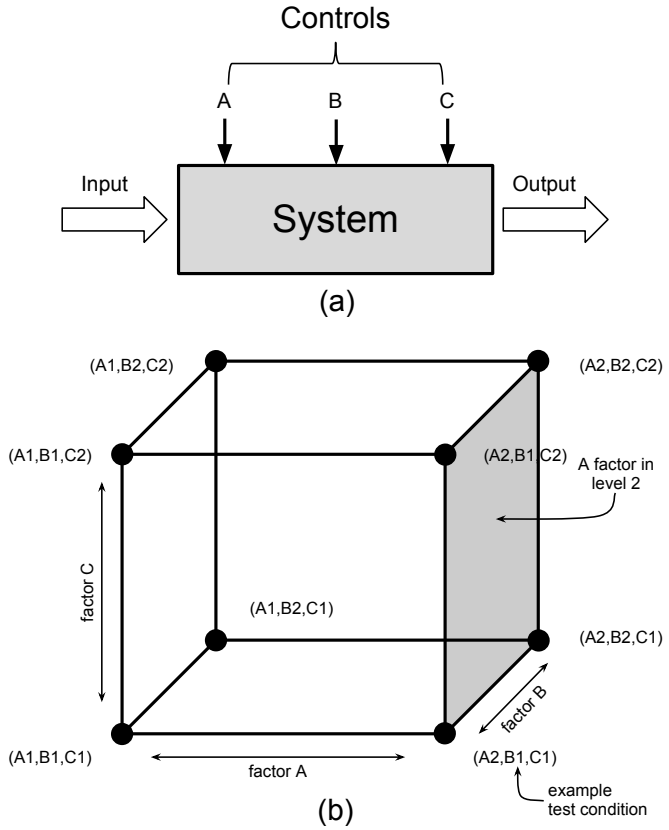

Figure 3.1: Factorial experiment design space for 3 variable with 2 levels each

factors.

In addition to the factorial design, a few standard techniques like blocking, repetition and randomization can be applied to enhance the quality of measurements from the experiments. Blocking is used to improve the precision among selected factors by suppressing unwanted factors. For example, in our experimentation for determining the power model, a deliberate attempt is made to reduce the use of encryption and I/O related functional units to improve the accuracy of estimation of power consumption of the other major functional units. Repetition and randomization are done at the CPU functional unit level to reduce the effect of extraneous noise and thereby avoid any statistical bias on the outcome of the experiments. Once all experiments are carried out, a statistical method like analysis of variance (ANOVA) is applied to filter unwanted noise and determine the

effectiveness of every factor. This statistical analysis is needed when factors are assigned to experimental units by a combination of randomization and blocking to ensure the validity of the results.

3.2 System Details

Fig. 1.1(A) represents the system under observation, a standard chip multiprocessor (CMP). A CMP consists of many function units, with some of them being part of cores, and the remaining are in peripheral units, and are collectively termed as 'uncore'. The classification of the functional units is illustrated in Fig. 3.2. The focus of the DoE is on the functional units shown in Fig. 3.2 as their power consumption is the dominant component of the CMP's power.

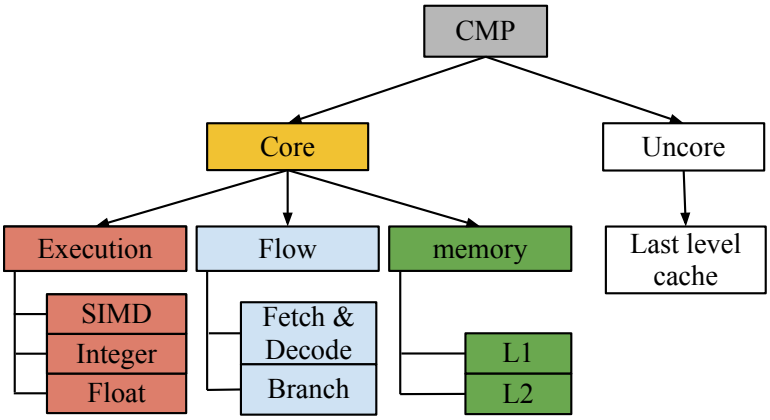

Figure 3.2: Hardware functional unit classification for a CMP

The power consumption of a processor depends mainly on two things: **(1)** the clock frequency of the processor, and **(2)** the subsets of the functional units that are accessed and the rate of their access. This is a characteristic of the program code.

Role of Frequency and Voltage on Processor Power

The power consumption of a processor is typically modeled as the sum of dynamic and leakage power as shown below:

$$P = p_{dyn}^{max} f v^2 + p_{lkg}(v, T), \tag{3.1}$$

where p_{dyn}^{max} is the maximum possible dynamic power consumption, and f and v are the frequency and the voltage of the processor, respectively. The leakage power is denoted by p_{lkg}, which is a function of the operational voltage and the current die temperature. Note that the clock frequency and the voltage in (3.1) are not independent. In general, the maximum frequency, the supply voltage and the temperature are all constrained w.r.t each other. This is because circuit delay increases with temperature due to mobility degradation, which can be compensated by increasing the voltage, which in turn increases the both the dynamic and leakage power, leading to increased temperature. This complex dependency is extremely difficult to model as well as use. Consequently, in practice, for a given clock frequency, the range of operational voltages at which the circuit timing constraints are satisfied at some worst-case corner are determined empirically. This is done for each frequency within some range. As a result, most manufacturers only allow the control of pre-determined 'voltage-frequency' pairs at which the timing constraints are satisfied. These pairs are called *P-states*. P-states are applied globally to all cores (as in Intel Sandy Bridge) or locally per-core (as in Qualcomm Krait) depending on whether each core has a voltage island of its own. Although, per-core DVFS allows for more energy savings over global DVFS, global DVFS has the benefit of reducing expensive voltage islands.

On the other hand, for a given voltage, a processor's clock can be throttled to a desired level to reduce power consumption. This is usually achieved by either inserting halt instructions or through clock modulation. On some platforms this is implemented using clock modulation as shown in Fig. 3.3, and the resulting states are called a *T-state*. Unlike a P-state, a T-state can be changed on a per-core basis. An example of allowed P-states and T-states on Intel Sandy Bridge processor is given in Table 3.1.

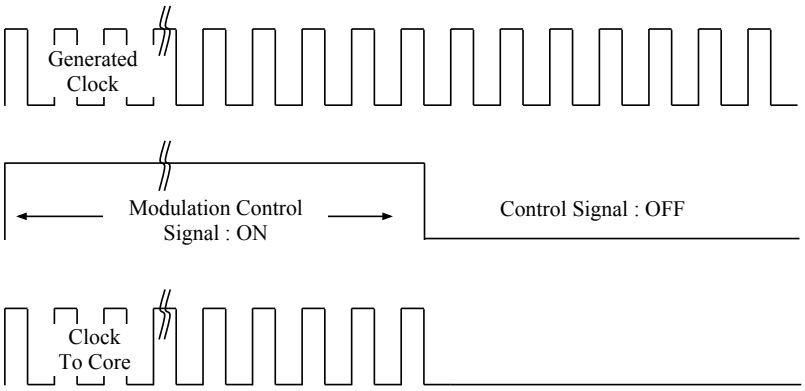

Figure 3.3: T state implementation through clock modulation control

Table 3.1: P-state and T-state specifications for Intel Sandy Bridge processor

P-state	Voltage (V)	F_{max} (GHz)	T-state	% throttling
0	1.000732	2.1	0	00.0%
1	0.975708	2	1	12.5%
2	0.955688	1.9	2	25.0%
3	0.930664	1.8	3	37.5%
4	0.910645	1.7	4	50.0%
5	0.910645	1.6	5	62.5%
6	0.885620	1.5	6	75.0%
7	0.865601	1.4	7	87.5%
8	0.840576	1.3		
9	0.820557	1.2		
10	0.795532	1.1		
11	0.775513	1		
12	0.775513	0.9		
13	0.775513	0.8		

Access rates of functional units

As seen from Fig. 3.2, there are large number of functional units in a processor. Each access of a functional unit consumes a certain amount of energy, which varies greatly among functional units. Hence it is necessary to monitor the access rates of these functional units. Many modern processors provide an option of monitoring such events, also called *performance counters*. A single performance counter can be programmed to capture any one of a large set of hardware events. However, due to the limitations on the

number of available performance counters, only few of the many possible hardware events can be monitored simultaneously. Therefore, it is important to identify the prominent sources of power consumption based on the knowledge of processor architecture, such that these sources can account for most of the processor power consumption. Table 3.2 lists a set of factors that were selected for the experiments on the Intel Sandy Bridge processor. The choice of these is based on knowledge of the architecture, as well as, extensive experimental exploration of the system. Our approach to identifying these factors was based on the classification of the functional units shown in Fig. 3.2. For instance, the events corresponding to the program flow are the number of instructions retired and accesses and misses to the memory hierarchy.

Table 3.2: Processor factors used in modelling processor power

Factor	Granularly	Notation
P-state	chip	ρ
T-state	core	τ
Temperature	core	T
Instructions Retired	core	μ
Core cycles	core	–
Integer operations	core	–
Floating operations	core	–
SIMD operations	core	–
Load and store	core	Δ_l, Δ_s
L1_D accesses/misses	core	–
L1_I accesses/misses	core	–
L2 access	core	Δ_2
Last-level cache access	chip	Δ_3
Memory controller access	chip	Δ_m

In order to conduct a factorial DoE, all factors should be controllable independently. In Table 3.2, except for the P-state and T-state, most events are not directly controllable, but can be indirectly controlled by modifying a program code to selectively effect one functional unit, while blocking all other units. Since temperature contributes to leakage power, which is significant in sub-micron designs, it is also listed in the table.

In the next chapter, I will identify the relationship of various factors on the processor power, and a method to build a model based on the observed relationships.

Chapter 4

MODEL IDENTIFICATION

4.1 Analyzing variation in power consumption of cores

There can be significant variation in power consumption among different cores. This is especially prominent in heterogeneous multicores like ARM big-little architecture [1]. Even for a homogeneous multicore processor, there can be variation in core power consumption due to process variations. In order to analyze this variation, I designed an experiment, where a known application was executed on one core at a time, leading to n trials for an n core processor, and measured the resulting power consumption. Analysis of the variation in the power consumption of the n trials determines the ratios of power consumption of various cores. Fig. 4.1 shows the scatter plot of the deviation in power consumption of various cores against the mean power consumption of a known application on our reference platform.

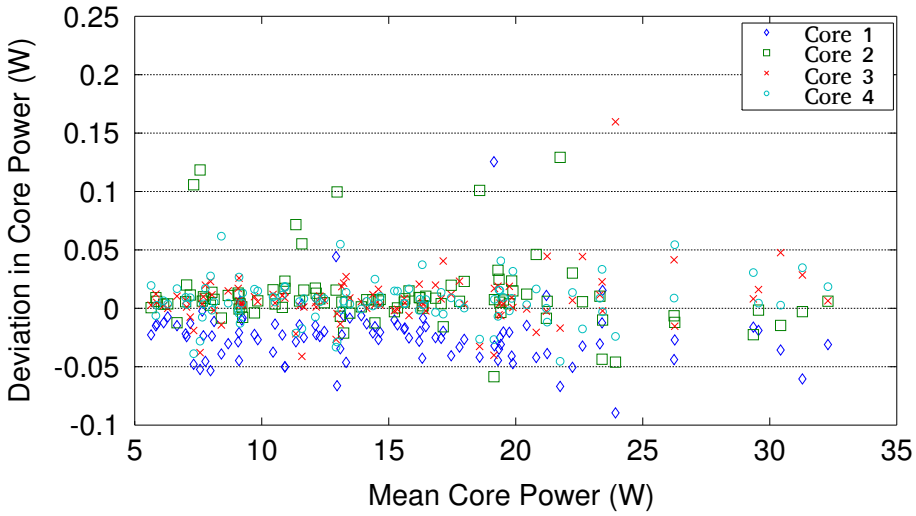

Figure 4.1: Scatter plot of deviation of power of cores from the mean of core powers

In the following sections we analyze the relationship of various factors listed in Table 3.2 w.r.t. power consumption. Let P be the measured total power consumption of the processor. The notation for each of the factors is listed in Table 3.2. A subscript $_c$

denotes the variable is involved with core c. Let $P_\rho|_{\tau,\mu,T,\Delta_l,\Delta_s,\Delta_2,\Delta_3,\Delta_m}$ denote the power consumption of P w.r.t. P-state ρ while keeping other parameters $\tau,\mu,T,\Delta_l,\Delta_s,\Delta_2,\Delta_3,\Delta_m$ constant. I first start with analyzing the effect of P-states and T-states on the total power consumption.

4.2 Effect of P-states and T-states on Power Consumption

From (3.1), we expect P-state to have a cubic effect on dynamic power consumption. However, a closer look at the P-states in Table 3.1 show that while the frequency changes by constant amount, the voltage changes non-linearly. The theoretical cubic relation between power consumption, and frequency and voltage, when characterized according to P-states is closer to being quadratic. This is shown in Fig. 4.2, which plots the power consumption vs. P-states for Intel Sandy Bridge processor.

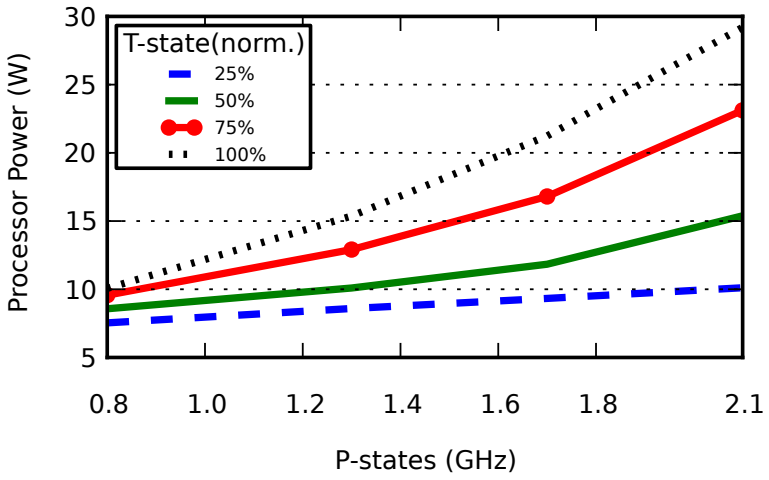

Figure 4.2: Quadratic effect of P-states on processor on power consumption

Equation (4.1) captures the relation between the power consumption of a core and its P-states using a second degree polynomial.

$$P_\rho|_{\tau,\mu,T,\Delta_l,\Delta_s,\Delta_2,\Delta_3,\Delta_m} = \sum_{c=0}^{n}\sum_{i=0}^{2} k_{\rho,i,c}\rho_c^i, \tag{4.1}$$

where ρ_c is the P-state of core c. Note that the association between F_{max} and voltage is a function, while the converse is not. Consequently, we use F_{max} to identify the P-state. The coefficients $k_{\rho,2,c}$, $k_{\rho,1,c}$, and $k_{\rho,0,c}$ are processor specific, which needs to be determined.

T-states are not the same as DFS. Fig. 4.3 shows plots of the processor's power consumption versus the T-states (measured as the % of throttling). For low frequencies, the relation is linear, but changes considerably at higher frequencies.

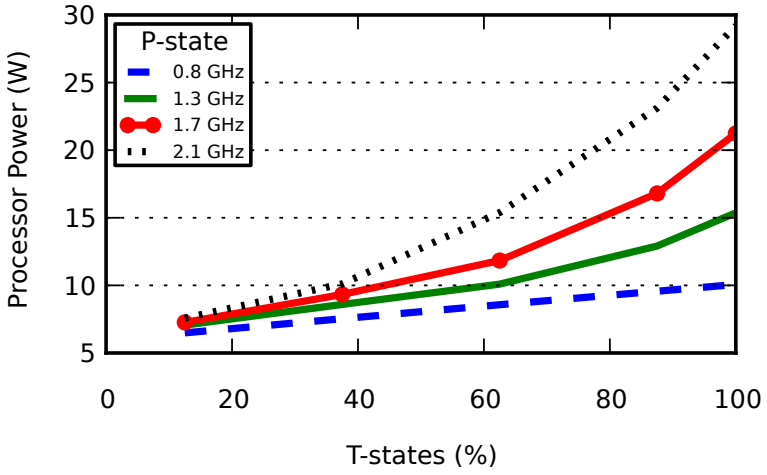

Figure 4.3: Quadratic effect of T-states on processor on power consumption

Similar to (4.1), (4.2) expresses the total power consumption of a core w.r.t. T-states as a quadratic relation.

$$P_\tau|_{\rho,\mu,T,\Delta_l,\Delta_s,\Delta_2,\Delta_3,\Delta_m} = \sum_{c=0}^{n}\sum_{i=0}^{2} k_{\tau,i,c}\tau_c^i, \tag{4.2}$$

where τ_c is the T-state of core c, and $k_{\tau,2,c}$, $k_{\tau,1,c}$, and $k_{\tau,0,c}$ are coefficients to be estimated. The combined effect of P-states and T-states can be determined by finding the coefficients of (4.1), i.e.$[k_{\rho,2,c}, k_{\rho,1,c}, k_{\rho,0,c}]$ for various T-states, and observing the relation between $[k_{\rho,2,c}, k_{\rho,1,c}, k_{\rho,0,c}]$ and T-states. The experiment to determine the combined effects can also be conducted by observing how T-state coefficients ($[k_{\tau,2,c}, k_{\tau,1,c}, k_{\tau,0,c}]$) vary with varying the P-states. From our experiments, $[k_{\rho,2,c}, k_{\rho,1,c}, k_{\rho,0,c}]$ varies quadratically with

T-states. Overall, the combined effect of P-states and T-states can be represented as below:

$$P_{\tau,\rho}|_{\mu,T,\Delta_l,\Delta_s,\Delta_2,\Delta_3,\Delta_m} = \sum_{c=0}^{n}\sum_{i=0}^{2}\sum_{j=0}^{2} k_{\rho\tau,i,j,c}\, s^i_{p,c}\, s^j_{t,c}. \tag{4.3}$$

The above equation contains all possible combinations of P-state and T-state exponent terms from (4.1) and (4.2). Note that in general, $k_{\rho\tau,c,i,j} \neq k_{\rho,c,i} k_{\tau,c,j}$.

For now, we only show the partial relationship of a factor with the total power consumption, and in the end of this section we present a unified power model which includes all the factors. In general, combining relationship of two factors with the total power follows the product rule followed in (4.3), i.e. multiplying the product terms involved in the individual relationship of a factor with the total power. In order to avoid the above representation getting more unwieldy as more and more factors are added, we introduce a simpler representation of a factor's relationship with the total power. This simpler representation takes only the factors considered, and not the rest of the factors that are held constant, e.g. we will represent $P_\tau|_{\rho,\mu,T,\Delta_l,\Delta_s,\Delta_2,\Delta_3,\Delta_m}$ as just P_τ.

4.3 Effect of IPC on Processor Power

The power consumption of a task increases with its *instructions committed per cycle* or IPC. In general, the processor power increases linearly with the IPC, but at a fairly slow rate (see Fig. 4.4). It is true that the IPC can be further refined by distinguishing the different types of instructions (e.g. integer and floating point). However, this did not substantially improve the accuracy of the power prediction. The relationship between the IPC of a task in a core and the total power is given by

$$P_\mu = \sum_{c=0}^{n}\sum_{i=0}^{1} k_{\mu,i,c}\, \mu^i. \tag{4.4}$$

4.4 Analysis of Memory Power

Memory power is the factor that is most difficult to analyze. The primary reason is that it is hard to control various factors of cache and memory controller (MC) from a program.

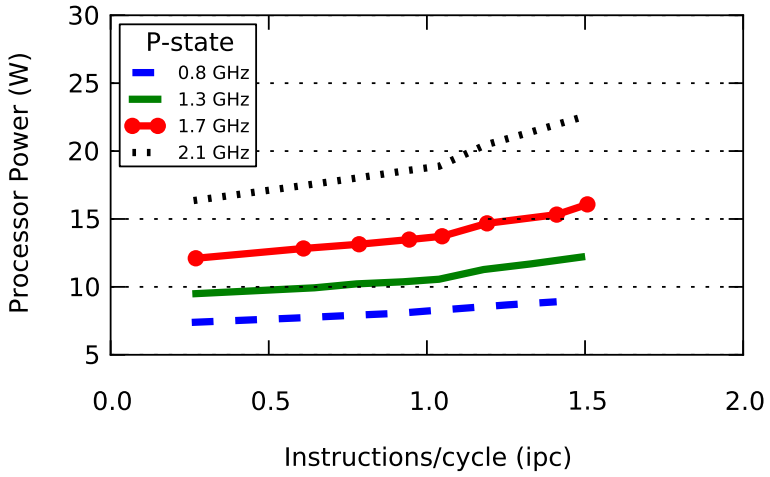

Figure 4.4: Linear effect of increasing IPC over power consumption

The second reason is that caches consume large leakage power. Thus, it is hard to see the effect of accessing any blocks as the change is small. We identified four factors related to memory access that have an impact on the total power consumption. These are (loads + stores)/cycle, L2 accesses/cycle, L3 accesses/cycle, and MC accesses/cycle. The values for the above four factors are obtained by combining various performance counters associated with memory access.

The plots of factors related to memory access along with the total power consumption for various sizes of the memory working-set of the programs are shown in Fig. 4.5. Notice that the number of accesses for each component varies with the working-set size. The plots show that considering only load and store operations does not sufficiently account for the power consumption at large working-set sizes. For this reason, we consider the higher level cache and memory controller accesses. Through experiments it was found that power consumption can be modeled as a linear function of each of these individual factors. The relation between the above memory access factors to the total power consumption is given by

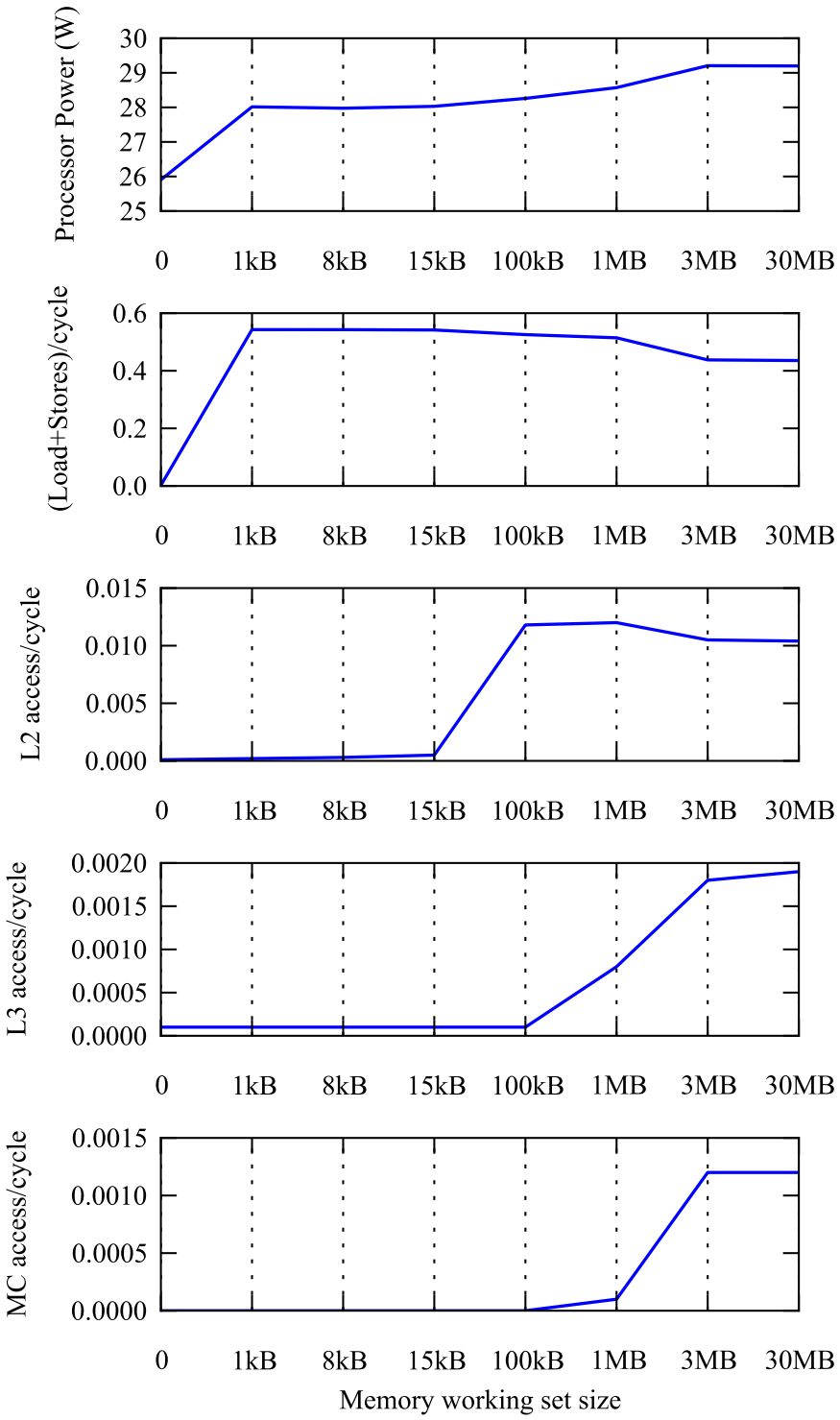

Figure 4.5: Effect of memory working set size on various factors

$$P_{\Delta_l,\Delta_s,\Delta_2,\Delta_3,\Delta_m} = \sum_{c=0}^{n}\sum_{i=0}^{1}[k_{\Delta_l s,i,c}(\Delta_l + \Delta_s)^i + k_{\Delta_2,i,c}\Delta_2^i + k_{\Delta_3,i,c}\Delta_3^i + k_{\Delta_m,i,c}\Delta_m^i]. \quad (4.5)$$

4.5 Effect of Temperature on Leakage Power

Temperature plays a significant factor in contributing to the leakage power. The relationship between temperature and leakage power has been well studied in the literature, and is usually approximated by an exponential relationship [15]. However, the ratio of the leakage power to the dynamic power is low. As a result, this exponential relationship will not be observed in the case when only the total power, which is a combination of both the leakage power and the dynamic power, is measured. In fact, we observed an approximate linear relationship of total power with the mean core temperatures as illustrated in Fig. 4.6. The relationship between a core temperature and the total power is given by

$$P_T = \sum_{c=0}^{n}\sum_{i=0}^{1} k_{T,i,c}T^i. \quad (4.6)$$

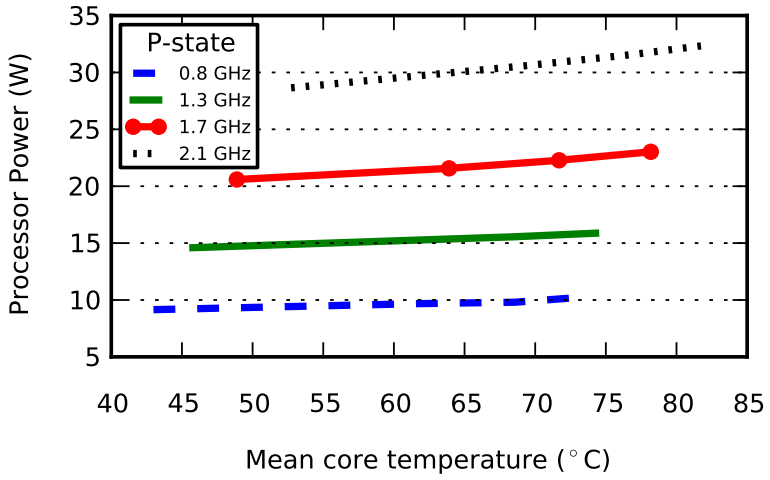

Figure 4.6: Linear effect of increasing core temperatures over power consumption

4.6 Derived Power Model

The combined equation that relates all the factors with the total power consumption is given by

$$P = P_\rho * P_\tau * P_\mu * P_T * P_{\Delta_l,\Delta_s,\Delta_2,\Delta_3,\Delta_m}. \tag{4.7}$$

The coefficients in the above equation is determined using the linear least-square (LLS) method [30] as all the terms have linear coefficients. The LLS equation is given by

$$A\mathbf{k} = \mathbf{P}, \tag{4.8}$$

where $\mathbf{P}$ is a vector of P for various time instants; $\mathbf{k}$ is a vector, where all ks in (4.8) are arranged serially in a decided order; A is a matrix that contains the elements $\rho_c^i \tau_c^j \mu_c^k T_c^l (\Delta_l, \Delta_s, \Delta_2, \Delta_3, \Delta_m)^m$ that matches the arrangement of ks.

Chapter 5

BUILDING A TASK LEVEL POWER PROFILER IN LINUX

In this chapter, I will describe how the task specific power profiles are generated and maintained dynamically for a Linux OS, although the proposed profiler methodology is adaptable for any other OS. The profiling process is especially challenging in a multi-core environment where hardware resources are shared among tasks via time multiplexing, parallel execution on multiple cores and task migration.

To understand task scheduling and management in Linux, consider a standard CMP as an example. In a standard CMP, each core has a set of tasks eligible to run in its private run-queue. A task is a program, and can be defined as a set of instructions being executed in the processor which takes finite amount of time to finish. Tasks are time multiplexed in round-robin fashion on each core, each taken from the corresponding core's run-queue. Generally, the time slice length is proportional to the user assigned priority to the task. A higher priority task runs longer, thus consuming higher CPU resources. On every core, when a task uses up its allocated time slice, the operating system preempts the running task and the next task in the run-queue of the core [29, 31] is assigned to the core.

A task profiler is a module, which can be inside or outside of an operating system. It dynamically monitors and analyzes tasks running on all cores. With support from the hardware platform in terms of performance counters, it is feasible to monitor the dynamic behaviour of programs and profile them with almost negligible overhead. The proposed task profiler is an online, architecture independent software module, which is implemented inside Linux OS to analyze the dynamic behaviour of tasks.

The generation of a task-level power profiler happens in two stages: for every task, the task profiler **(i)** collects performance events pertaining to that task, and **(ii)** uses the collected events to extract power consumption of the task. The performance profile of a

task is a data structure stored within the task that consists of predefined set of performance counter values. It accurately represents the dynamic behaviour of a task. Similar to the performance profile, a power profile is also maintained for every task within the task data structure. The power profile keeps track of the power consumption of a task in the previous few scheduling intervals.

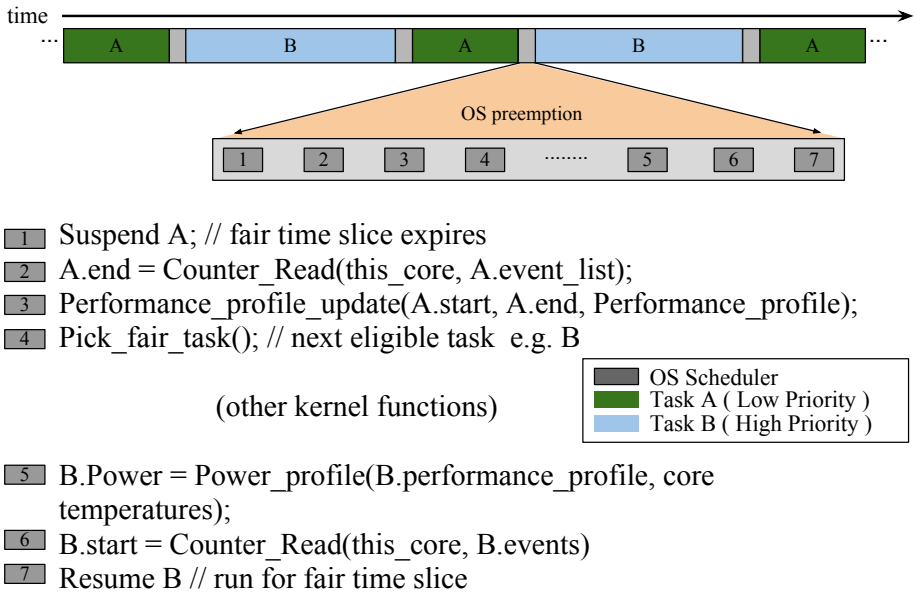

Figure 5.1: An example of the proposed task profiler profiling two tasks A and B

Fig. 5.1 illustrates the implementation of the task profiler within the Linux scheduler. Two tasks A and B are running on one of the cores in a CMP. Task A has been assigned a lower priority than task B, and hence gets a smaller share of the CPU's resources. During the context switching period as highlighted in Fig. 5.1, the task profiler is invoked by the OS. It updates $A.end$ in the task profiler's data structure `task_struct` with the current values of the performance counters. The entry $A.start$ contains the performance counter values stored at the beginning of the OS scheduler time slice. Every task maintains its own $.start$ and $.end$ fields. Let $A.E_i$ represent a hardware event i related to Task A. Then the hardware events that occurred during an interval related to Task A are given by $A.E_{i,cur} = A.start_i - A.end_i$. These event counts need to be filtered as they can be

affected by artificial phase changes in a task due to interrupts and rapid load changes. These glitches in performance counter values can lead to misleading performance as well as power predictions. For smoothening the event counters, we use exponential smoothing filter as shown below:

$$A.E_i = kA.E_{i,cur} + (1-k)A.E_i, \tag{5.1}$$

where $A.E_i$ refers to the value of the performance counter i for the current time; $k \in [0,1]$ (default $k = 0.5$) is the exponential smoothing filter's weight.

The above procedure is repeated at every context switch, while noting the performance counter values that are stored in the previous task's *.end* and for the upcoming task's *.start* fields of their respective data structures. The power profile for a task is computed using the events stored in the task data structure and applying the power models as derived in (4.8). This power number is stored in the task data structure. Since the constructed power model is just an approximation, and never can accurately predict the power consumption, we need to include an adaptive filter that corrects the model parameters with every new data. This is achieved by using recursive least squares (RLS) [4] filter. The goal of the RLS filter is to minimize the weighted error of the estimated total processor power w.r.t. the measured total processor power. The weights are chosen such that the recent values of the estimated power are given more significance.

Migrating tasks is not an issue for the task profiler, as the task profiler runs in a distributed fashion on all cores allowing tasks to migrate from one run queue to another easily. Also, the task's data structure is always kept intact since the data structure is stored within the task's private memory. Another interesting scenario in a CMP system is profiling of multi-threaded programs. In this case, we profile every thread, treating a thread as a standalone program. The proposed task profiler is summarized in Algorithm 1.

Input: Performance event list, no. of cores (C), task run-queues ($\Gamma_i, \forall i \in C$)
Output: Performance and power profile of every task
Generation of power models; (Section 3)
for *every core* $i \in C$ *(in parallel)* **do**
    for *every task* $j \in \Gamma_i$*, at every scheduling interval* $t_s \in [4\ ms, 20\ ms]$ **do**
        if j *is new task* **then**
            Create and initialize task j's performance and power profile within its data structure;
        Build task j performance profile using performance counters;
        Smooth the counter values using (5.1);
        Compute task j's power using (4.8);
        Run RLS to update task t_j's power profile;
        if j *is exiting* **then**
            Delete task j's performance and power profile and its corresponding task data structure;

Algorithm 1: Overall procedure of task-level power profiling in CMPs

Chapter 6

EXPERIMENTAL VALIDATION

6.1 Experiment Platform

- The experiments for validation and case studies are conducted on a 32 nm, quad-core Intel Sandy Bridge processor [26] running SMP Linux kernel 3.5.5 of Ubuntu distribution. Hyper threading or simultaneous multi-threading (SMT) is disabled as hyper threading complicates the derivation of power models. Thus hyper threading is currently not part of the proposed power profiler.

- The power measurements for the processor is done by reading the specific model-specific registers (MSR) [10]. The on-board power measurements can be easily replaced with any external power measurements, which is necessary for processors where such a feature is not available. Power measurements are used for both validation and also for adaptive correction of the model error.

- Hardware events are monitored and collected inside kernel by reading performance counters using Intel performance counter driver. The overhead of reading performance counters is much less inside the kernel than in user space since there is no overhead of using system call or user space library. Each reading of the performance counter measures anywhere between 2 μs and 5 μs.

- The global P-states on our experimental Sandy Bridge processor are changed using the APIs provided by Advanced Configuration and Power Interference (ACPI) [9]; while the per-core T-states are modified by directly writing into specific MSR registers.

- Applications from SPEC CPU2006 [2] and PARSEC-2.0 [2] benchmark suits were used to validate proposed power profiler.

Figure 6.1: Mean error

Figure 6.2: Standard deviation of error

6.2 Profiler Validation

Profiler validation has been carried out in two parts. First set of experiments were done to validate the performance profile of a profiler and then power predictions are validated using on board hardware power estimations.

Performance validation

Before validating the power prediction of the proposed profiler, we need to ensure that the performance events used by the profiler are also validated. Towards this, we validate the recording of the performance events with the standard performance monitoring tools based on *lib-perfmon* [32] library. We observe less than 0.05% error between the values recorded by *lib-perfmon* based profiler and our proposed profiler. Note that this error also includes the fact some of the performance counter events are not completely deterministic, and hence there is some inherent measurement noise.

Analysis of power prediction error

For the purpose of validating the proposed profiler prediction of power consumption, we ran both SPEC CPU2006 and PARSEC 2.0 benchmarks, and varied the execution rates of the benchmarks by randomly varying P-states and T-states. Using the models derived in Section 4, and the RLS method, the task profiler makes a prediction of the current power consumption for every task. Since the only available measurement of power is the total processor power, we summed all the tasks' power consumption in a given time and compared with the total power measurement. Fig. 6.1 and Fig. 6.2 plots the mean and the standard deviation of the prediction error of our proposed power profiler against the measurement of the total power consumption.

The mean error refers to the average prediction error on the entire run of a benchmark, which in ideal scenario should be zero; while the standard deviation of error refers to the average deviation of a prediction from the actual measurement for every

sample, which cannot be lesser than the measurement noise of the sensor. The power consumption of SPEC and PARSEC benchmarks varied from 10 W to 40 W.

6.3 Accuracy Analysis and System Overhead

Reading and maintaining performance counters, synchronizing between cores, evaluating power models and maintaining power prediction history, generates additional overhead. The power profiler is integrated within the scheduler and as described earlier in Chapter 5, the profiler is called by the scheduler on every context switch. The frequency of context switching can be as high as 250 times per second. Maintaining low overhead restricts the computational complexity and so the model accuracy. Even though profiler generates less then 1% overhead, performance counter values were maintained with high accuracy. Similarly, power predictions are in within 4% error even with a noisy temperature and power sensors.

Chapter 7

USE CASES

In this chapter, we discuss the use cases of my proposed profiler for two emerging architectural techniques in commercial platforms: fine grained per-core DVFS control and architecturally heterogeneous cores. Although the above architectural techniques have shown promise in reducing power consumption, they are yet to be seamlessly integrated with the system software, partially due to the lack of a detailed task-level power profiler. In the following sections, we discuss how the proposed profiler can efficiently bridge the gap between available hardware techniques and system software to improve energy efficiency of future processors.

7.1 Task priority-aware fine grained DVFS including soft power and thermal capping

The DVFS techniques currently adopted for power and thermal management are task agnostic. The DVFS decisions made purely on system-level power and temperature information might achieve the desired power savings for the processor, but can be unfair w.r.t. a task, if its priority is not taken into consideration. This is also true w.r.t. hardware power capping. As an example, consider the plot of *performance/Watt* (PPW) of two SPEC CPU2006 benchmarks running on Intel Sandy Bridge as shown in Fig. 7.1. PPW is an effective metric representing task's energy efficiency, accounting power and performance of a task. Figure shows that the optimal frequency of execution to achieve maximum PPW is not same for all applications, but depends on the application requirements of the hardware resources, e.g. a high IPC application will need a higher operating frequency to improve energy efficiency than a low IPC application as seen from the figure. Such observations are not possible without the use of a task-level power profiler.

An accurate power model at the task-level is necessary to ensure that a task is not unfairly throttled and to complete its execution in a reasonable time. Also, accurate power

models can help in enforcing soft power and thermal capping, where the capping can be relaxed for a short time to boost performance. This is similar to turbo mode in Intel processors [26]. Recent architectural improvements have enabled low overhead per core DVFS, which has increased the effectiveness of DVFS compared to single voltage domain. Such a hardware feature can be used to improve system energy efficiency, while at the same time respecting task priorities.

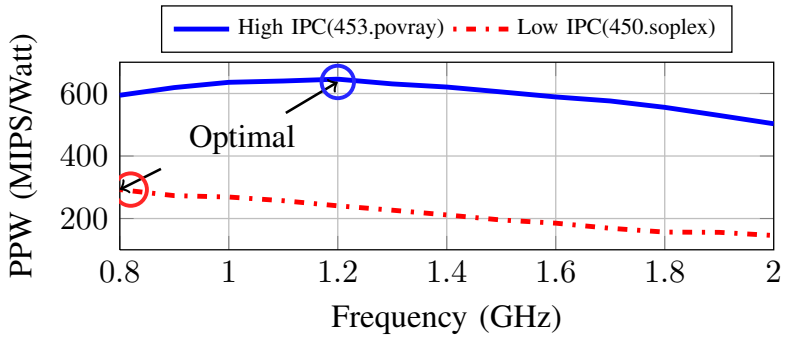

Figure 7.1: Different applications have different optimal DVFS operating points for maximum energy efficiency

7.2 Heterogeneous task-to-core mapping

Asymmetric multicores have recently gained popularity as they provide more opportunities to improve energy-efficiency than standard homogeneous multi-cores. A typical heterogeneous chip multiprocessors (HMP) consists of two different set of cores on a single die with both having the same instruction set architecture (ISA). Of the two set of cores, one set of cores is designed to deliver high performance, and the other set offers high energy efficiency, but with reduced performance. In other words, both set of cores differ in terms of their power consumption and performance, e.g. ARM's bigLITTLE [1]. With a power model that accounts for this kind of heterogeneity among cores, one can develop intelligent task-level, energy aware load balancing and scheduling techniques to improve energy efficiency of a processor.

Using a task-level power profiler, one can derive a metric to effectively represent

runtime energy efficiency of a task. This metric can be used to select most suitable type of core to run a task. This dynamic task-to-core mapping is important to improve energy efficiency under changing thermal and workload conditions.

To demonstrate the importance of a task profiler for heterogeneous systems, we emulated a four core heterogeneous system on a four core SMP system. This system has two performance oriented cores and two lower performance, but with energy efficiency. Heterogeneity is emulated by adopting clock modulation using ACPI T-states to produce low performance cores as shown in Fig. 7.2. Both high and low performance cores have voltage-frequency controls and performance counters support. As we are mimicking heterogeneity through clock modulation, no modification in the operating system is required. In order to do power and thermal profiling on architecturally different cores, one must develop separate power model parameters for different types of cores. For architecturally different, real heterogeneous cores, we can apply our proposed model building methodology described in Section 3 for both types of cores. In the current scenario, the low and the high performance cores differ only by the T-states they operate at.

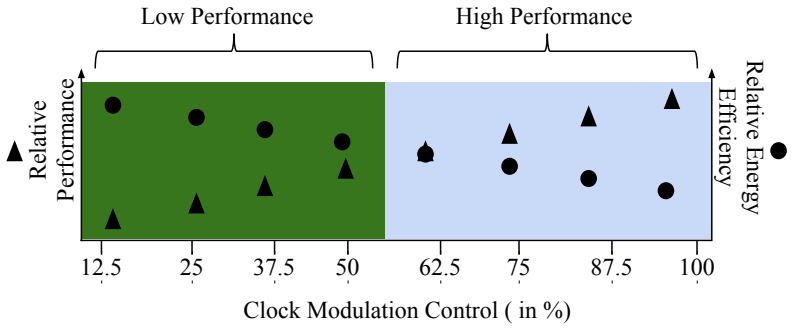

Figure 7.2: Emulating heterogeneous cores through core clock modulation

One of the primary goals in introducing heterogeneous cores is to achieve better energy efficiency by performing runtime task-to-core mapping. Without the help of a task profiler, the task-to-core mapping is very challenging. In order to demonstrate that there

are many situations in which such runtime task-to-core mapping is essential, we executed SPEC CPU2006 benchmark 473.astar on emulated heterogeneous core system. The performance/Watt (PPW) results are plotted in Fig. 7.3. Notice the dynamic change in the energy efficiency of the task running on high performance vs. low performance core. There are some segments where high performance core gives higher energy efficiency, and some other segments, where low performance core gives higher energy efficiency. Without a task-level power profiler, such changes in energy efficiency over heterogeneous cores cannot be detected, and benefits offered by heterogeneous cores will be wasted.

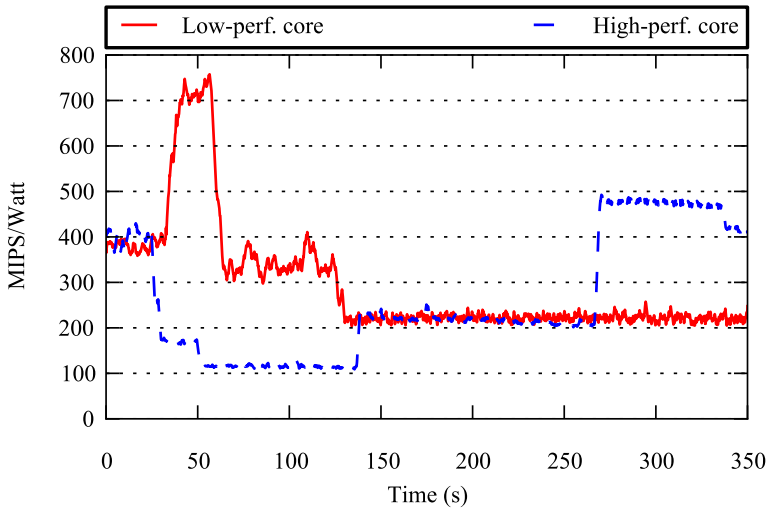

Figure 7.3: Energy efficiency comparison of execution of SPEC CPU2006 benchmark 473.astar on high and low performance cores. The task needs to be migrated from either core to provide high energy efficiency at various times.

Chapter 8

RELATED RESEARCH WORK

The fundamental CGRA architecture utilizes multiple processor elements (PEs) connected to the data memory via a 2D mesh network. In contrast to a general purpose processor, CGRAs are much more configurable in terms of what each PE does every cycle. Unlike a FPGA, which provides a finer-grain static reconfigurability, CGRA's coarse yet dynamic reconfigurability can be utilized by a compiler to map an application efficiently. CGRA's coarse reconfigurability greatly reduces the delay, area, power and configuration time with compared to a FPGA. Other features include predictable execution timing, a small instruction storage space and flexible topology. It is a promising architecture to deliver higher performance and better energy efficiency. Fig 1.2, compares legacy architecture models.

Since 1994, many researchers have developed variety of systems using CGRA as a centrepiece of their design [6]. Kress Array [7], REMARC [18], Matrix [17] are some of the popular on-chip CGRA designs. These earlier architectures use MIPS, PowerPC or ARM based cores to help the CGRA with more general tasks like memory management and operating systems. These designs have one or more high bandwidth on-chip scratch-pad memory units associated with the CGRA. The idea of using CGRA with processor cache memory is present in MorphoSys [27] and Garp [8]. In addition to hardware design space exploration, a lot of work went into analyzing the scheduling required to maximize utilization of CGRAs.

Architecture of the coarse-grained reconfigurable arrays required a deep understanding of the underlying architecture by the application developers, which made the CGRA less attractive. The modulo graph embedding technique for finding a more effective schedule was able to reduce the amount of extra routing to map the instructions

properly [21]. It was focused on the interprocess communication and assumes that the data will be available in the memory. In 2009, the same group of researchers introduced a new algorithm called edge-centric modulo scheduling which increases performance by 25%over traditional modulo scheduling and achieves 85-98% of the performance compared to a state-of-the-art simulated annealing technique [22]. However, this was also focused more on the routing of the instructions and output registers than how the memory referencing can affect the CGRA architecture. Unlike general purpose processors, the compiler and software programming model for the CGRA is still a long way from maturing and has a good scope in future research.

Evidently, the CGRA architecture is an interesting and a challenging research problem for both hardware as well as software developement. It can be a potential candidate as a center piece for accelerator rich future system-on-chip. Here in this research, I am proposing a flexible framework to perform design space experiments using CGRAs at software as well as hardware level. This will facilitate researchers to do hardware-software co-design at early design stage of developement thus reducing development time for a CGRA based systems.

Chapter 9

COMPONENTS OF THE FRAMEWORK

9.1 CGRA- future of programmable accelerators

Components of the proposed framework can be classified into two broad categories **(i)** Hardware Models in to the simulator and **(ii)** Software modules. For this work, hardware models including the CGRA model are part of the architectural simulator. It is a software tool which simulates pre-configured processor using its hardware models to mimics the actual hardware. This simulated hardware is accurate enough to boot unmodified Linux operating system. Once simulator is updated with the CGRA hardware model, we need to add few software components to the OS to utilize newly added hardware models. This operating system components include device drivers to handle CGRA accelerator and user space applications. Following this chapter, first we will see hardware components added in the simulator and later we will see software modules added in the OS in details.

9.2 Hardware Components

GEM5 is an open source, cycle accurate architectural simulator [3]. It is based on the discrete event driven simulation principle. GEM5 is capable of simulating various popular architecture like ARM, x86 etc. In addition to support multiple architectures, it can run unmodified operating system with reasonable speed and fairly good accuracy. Since it is an open source software, adding hardware models or modifying existing ones is not difficult. Such attributes make GEM5 an attractive platform for this research work where design space exploration requires adding and altering the hardware models. Overview of the GEM5 simulator based system is shown in Fig. 9.1.

Embedded and low power system has hard constraint on power consumptions and battery life. For demonstration purpose in this work, I selected the ARM 32 bit, in-order microprocessor based system to integrate the CGRA as an off-chip accelerator. CGRA is attached to the IO bus of the processor and to the ARM DMA PL081 [25]. The DMA

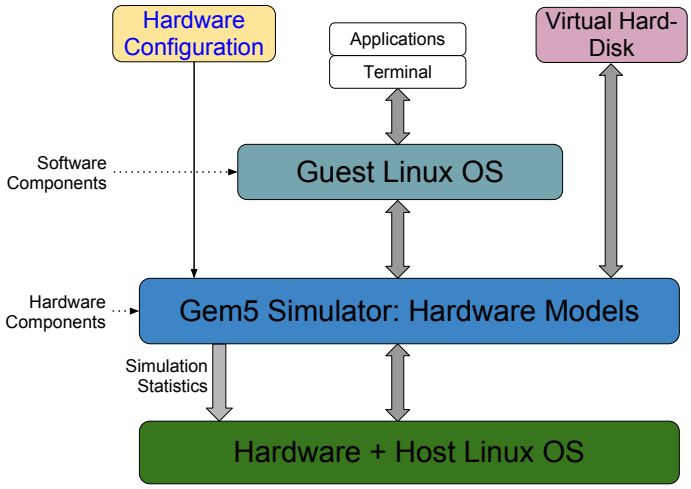

Figure 9.1: GEM5 simulation environment

controller is used to efficiently offload memory transfer request between the main memory and the CGRA memories. In this section, we will see the CGRA hardware model and its interface with IO bus and ARM DMA controller in detail.

CGRA Model and interfaces

The CGRA is a promising architecture capable of delivering high throughput at the greater energy efficiency than a standard processor. Its 2D mesh like array architecture as shown in Fig. 9.2 connects processing elements called PEs. Each PE is connected to its neighbors with a private interconnect and connected to a data memory with a shared bus. The PE typically consists of set of functional units capable of performing basic integer arithmetic and logical operations. The instruction supplied to CGRA is determined by the compiler at a compile time. This static mapping removes a need for complicated dynamic scheduling inside PE and thus improving energy efficiency. Reduced PE complexity also reduce decoder complexity lead to power saving in decoding. Every PE also has a small register file and a shared element which can be accessed by neighbor PEs.

This simple 2D structure has a private instruction and a data memory. The instruction memory has one port for each PE to supply 32 bit wide instructions every

cycle. Unlike the instruction memory, the data memory has only 4 ports and all PEs in a row share the data memory port. The instruction and the data memories are treated as scratch pad memories further eliminating need for power hungry, complicated tag store and compare hardware circuits. Just like any other scratch pad memory, CGRA instruction and data memories have to be managed by software during the program execution. In addition to that, from system point of view, CGRA is connected to IO bus and thus being treated as standard peripheral device. DMA is used to manage data and instruction memories of CGRA, just like any other peripheral device. This is a standard solution adopted by many other off-chip accelerators.

CGRA model developed here as shown in Fig. 9.2, features 16 PEs. Each PE consists of a basic ALU and a register file. On every cycle, PE fetches and decodes 32 bit instruction. As all the instructions are mapped statically by the compiler, CGRA eliminates need of complicated hardware dynamic scheduling e.g. out-of-order processing. This leads to better hardware utilization and performance at lower power consumption. CGRA's PE instruction encoding is shown in Table 9.1. As shown in the table, it has 3 bit encoding for Op-code and Multiplexer selection. The register file has 2 bit register selection fields, meaning each PE has 4 registers in their register file. 3 bit opcode and the multiplexer selection fields are explain in the Table 9.2. Each PE supports 8 preliminary arithmetic and logical operations selected by the bit-field `Op-Code`. In the table we see all 8 different opcodes supported by the PE. Each opcode consumes two 32 bit operand and produces 32 bit output. Selection of both input operand source is done by the instruction bit fields called `Left` and `Right`. These bit field control the select pins of the multiplexers present on both of the ALU inputs. Both input multiplexers can select operand from the neighbour PEs, data memory, immediate and register file as shown in Table 9.2. Similarly, the destination for output operand can be controlled through the output multiplexer. Setting the `Write Enable` bit will write output to the register file

and `Data` bit field can be used to when to send the data on the data bus for either to data memory or other PEs. It also has a `predicate` field which can be used by the compiler in various ways while dealing with if-then-else type conditional program mapping. Lastly, instruction also has a 12 bit unsigned `immediate` field, allowing constants from 0 to 2^{12-1} inside instruction. Communication with data memory is done by putting a address on data memory address bus by selecting bit 13(`Addr`) and if store, selecting bit `Data` in the same cycle by some other PE in the same raw. But if load, data memory will put data on the same port in the next cycle, available to all PEs sharing the bus.

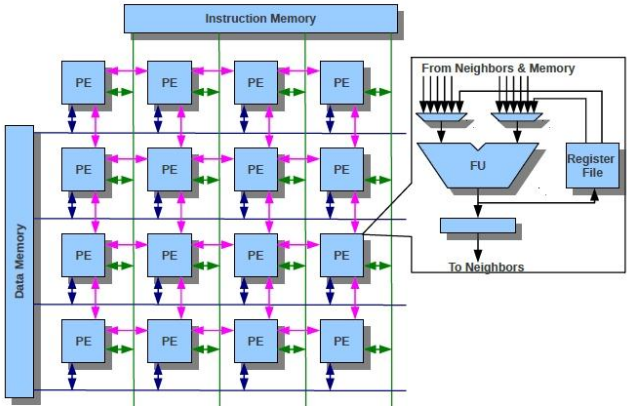

Figure 9.2: Architecture overview of CGRA

Table 9.1: Encoding of 32 bit PE Instruction

31-29	28,27	26-24	23-21	20,19	18,17	16,15	14	13	12	11-0
Op code	pred-icate	Left Mux	Right Mux	Reg. A	Reg. B	Reg. O	Write Enable	Addr Bus	Data Bus	Imm

CGRA relies on the DMA controller for any data or instruction memory transfer to or from the main memory. The DMA used here is the ARM PL081 controller. Programming the DMA controller is integrated into the master-slave programming paradigm used for CGRA. DMA model developed is a minimalistic dual channel, single direction DMA engine mimicking PL081's features and programmability. It is capable to

Table 9.2: Specifications of PE op-code and multiplexer selection in CGRA Model

Encoding	Op Code	Mux Select
000	Add	Register
001	Sub	Left
010	Mult	Right
011	And	Up
100	Or	Down
101	Xor	DataBus
110	Asr	Immediate
111	Asl	Invalid

handling memory-to-memory, memory-to-peripheral and peripheral-to-memory modes. Modes involving peripherals are especially useful to transfer data and instruction from main memory to CGRA memories. It also support read, write competition interrupts to communicate with ARM core. The CGRA is connected to the IO bus (also known as peripheral bus). Thus can not access any system memory without a DMA and vice versa. It can not program the DMA either but it can communicate with ARM or master core though interrupt. This interrupt is also used by the CGRA device driver to maintain state of the CGRA.

All the hardware models discussed in this section are written in C++ and are part of the GEM5 simulator and can easily extended for more features. In following section, we will discuss about the software modules present in the Linux kernel to utilize the CGRA hardware and how it facilitate users to use CGRA from user space.

9.3 Software Components

The GEM5 simulator boots a guest Linux OS using the processor and memory hardware models. In order to utilize new hardware models added in the simulator, software support has to be added in OS e.g. device drivers and interface with kernel subsystems. This section will focus on necessary software modules added and modified to support CGRA accelerator in the guest OS. This includes CGRA device driver, ARM DMA device driver, and user space applications. Fig. 9.3 shows the overview of interconnection and

communication between user application, device drivers and hardware models.

Just like any other standard peripheral device, the CGRA accelerator is also memory mapped and DMA has to be used in order to copy data to and from CGRA memories. The memory mapping of CGRA is shown in Fig 9.4.

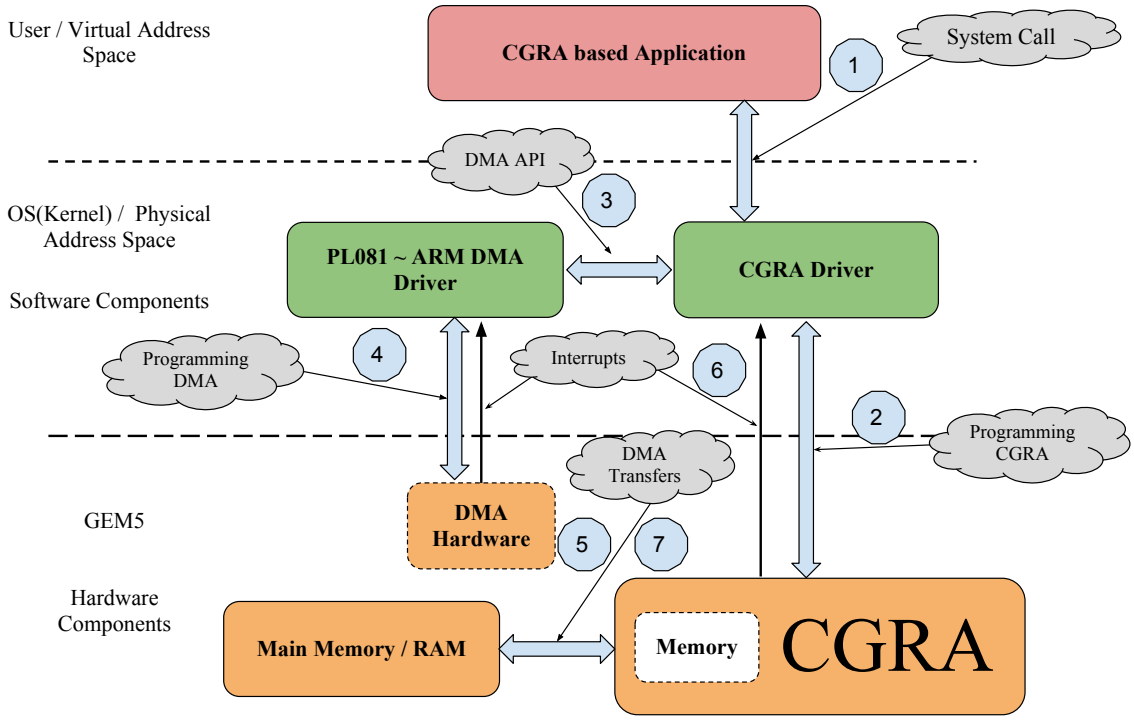

Figure 9.3: CGRA Hardware-Software Stack

User Interface and Device Drivers

As seen in Fig. 9.3, the user application directly communicate with the CGRA device driver through system calls. Once programmed, and initiated CGRA operation, user blocks itself and sleeps until CGRA driver signals the application. At the application level, communication with the CGRA through the CGRA driver is very straight forward, and described in Algorithm 2. First, the user application checks the availability of the accelerator. Once available, it obtains the ownership of the CGRA and locks it. Then, the actual transfer of user space pointers of .data and .text sections of CGRA binary and their sizes is done. Note that this step does not involve any actual copy of the data or

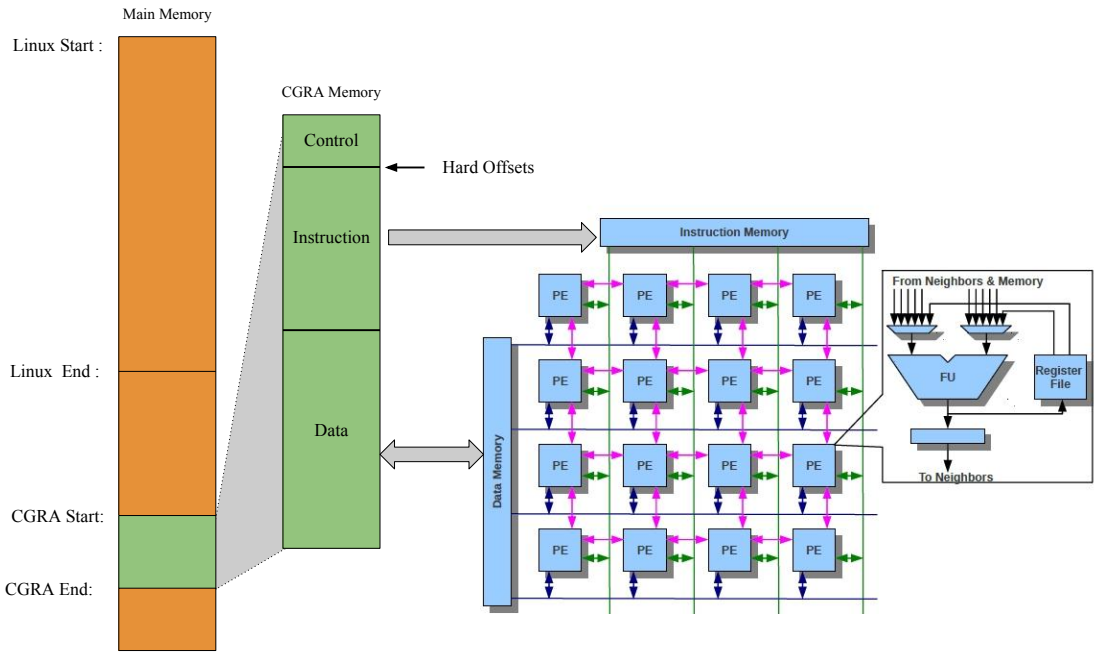

Figure 9.4: System memory mapping for CGRA

instruction. Actual data and instructions are transferred using DMA and handled by the CGRA driver. This data movements are completely transparent to the user application.

Input: CGRA Instructions and Data buffers (Binary: .text, .data sections)
Output: Processed data buffer(.data section or results)
if *CGRA is available* **then**
    Obtain CGRA Handler;(Ownership)
    Transfer CGRA instruction to CGRA Driver;(No actual copy)
    Transfer CGRA data to CGRA Driver;(No actual copy)
    Program CGRA control memory;
    Start CGRA;(Blocking Call);
    Use Results;
else
    Try after some time;

Algorithm 2: Procedure of CGRA User Application

CGRA device driver is a fundamental block of this framework. This software module is responsible to provide user level abstraction of the CGRA hardware and

controlling it. It also provide system call interface to the CGRA, allowing users to utilize the CGRA with minimal hardware knowledge.

Here follows the main features of the CGRA device driver, **(1)** Handling the DMA requests and maintaining the DMA queue, **(2)** Configuring the PL081 DMA controller for every DMA transfer, **(3)** Handling the virtual memory to the physical memory mapping and the page table entries, **(4)** Handler for PL081 and CGRA interrupts, **(5)** Configuring CGRA as per user application, **(7)** Maintaining the CGRA state and handle data and instruction flow with limited CGRA memory and lastly **(8)** Providing uniform interface to user space to utilize CGRA.

The internal workings of the CGRA device driver is shown in Algorithm 3.

Input: CGRA System Calls: Ownership, Memory transfer to and from user, Start, Stop, Release
Output: CGRA status:running, complete, available
Start: status=available; **if** *User System Call: Get Status* **then**
    return status;
if *User System Call: Transfer .text section* **then**
    Obtain pointer and size of the section; (virtual pages)
    Perform page table walk to obtain physical frame mapping;
    transfer all frames to CGRA instruction memory: **DMA_transfer;**
if *User System Call: Transfer .data section* **then**
    Obtain pointer and size of the section; (virtual pages)
    Perform page table walk to obtain physical frame mapping;
    transfer all frames to CGRA instruction memory: **DMA_transfer;**
if *User System Call: configure CGRA* **then**
    copy control data structure from user space;
    program CGRA control memory;
    start CGRA;
    status=running
Function DMA_transfer(Source address, Size, Direction [to/from], CGRA Memory Offset)
Add DMA Controller transfer setting in a DMA request Queue;
Perform Copy; (Non Blocking)
return;
Function CGRA_Interrupt_handler()
Update CGRA State: Complete;
Transfer all result frames to user: **DMA_transfer;**
Update CGRA State: available;
return;

Algorithm 3: Procedure of CGRA Driver

Chapter 10

USING THE FRAMEWORK

In the previous chapter, we saw the internals of the framework and the user space interface for application developement. In this section, we will see an example application development on CGRA from a Linux OS running on the 32 bit, in-order ARM processor. Assuming the framework is already installed in the guest Linux OS and it is running on GEM5 simulation platform with this CGRA hardware models available and ready to use. As explained in previous section, CGRA driver exposed few standard system calls to Linux user space. First, we will use those system call and develop a parallel, memory efficient matrix multiplication algorithm running on the CGRA accelerator. Then, we will compare the performance of the same algorithm running on an ARM processor with that running on the CGRA.

10.1 Programming CGRA

For an application to use CGRA, first it needs to provide .data section and .text section of the CGRA binary. This sections are nothing more than a buffer in a memory which can be generated dynamically using standard ARM libraries. As, the compilers for the current hardware accelerator are very premature and this work is primarily focused on dynamic software-hardware framework, I implemented .text and .data section in CGRA assembly language. It is reasonable to assume that a mature compile can automatically produce the CGRA instructions and data automatically without user to get familiar with hardware details or assembly language programming of the CGRA.

10.2 Benchmark: Matrix Multiplication

Since there is no compiler support available for now to test the framework, developing handwritten assembly benchmark is necessary. To validate functionality of the proposed framework and demonstrate usefulness of MIMD accelerator, I chose to implement a variable size matrix multiplication using Cannon's [14] memory efficient matrix

multiplication algorithm.

Here, the multiplication of two matrices, each of size MxM where, $M = 2^n, n \in N$ is done using a CGRA of size, P=16. To minimize the inter PE and memory communication, matrices are processed in blocks of size $N/\sqrt{P}xN/\sqrt{P}$. According to the Cannon's algorithm on the 2D array architecture, matrices are required to be skewed first and perform a circular shifts after every cycle. It keeps the partial results in the PE registers eliminating need of data movements. At the end of multiplication, PEs store the complete result ready to be written back to memory. It is highly memory efficient algorithm suitable for symmetric 2D array architecture. Implementation on 4 by 4 CGRA to finish one 16 by 16 matrix multiplication, one block for larger matrices takes about 35 cycles and it supports matrices upto 128 by 128 by partitioning them into 16x16 blocks. Keeping PE utilization to the highest and minimizing inter iteration delay are standard measures to improve quality of mapping. The average loop utilizes on average 80% of CGRA PEs during entire loop which has inter iteration delay of 35 cycles.

10.3 Results: ARM vs CGRA

For simplicity, the CGRA model performs multiplication and addition in single cycle and runs as fast as standard ARM in-order processor. Similarly the DMA also transfers data at 1 bytes/cycle speed. As this is complete simulation framework, bandwidth and latencies can be easily adjusted.

'Naive' implementation on the ARM in-order processor performs matrix multiplication sequentially. To exploit the parallel nature, the CGRA based multiplication is done on 2D mesh of distributed PEs in parallel. Inherently parallel applications like matrix multiplication which utilizes PEs well should give high performance boost and the expected speed-up is about P times but due to DMA transfer overhead associated with CGRA gives less than expected performance boost. CGRA memory limitations and without compiler support, hiding DMA latency by pipe-lining DMA transfers is not

possible. This puts DMA transfer operands from main memory to CGRA memory and results from CGRA to main memory in critical path.

Preliminary results are presented in the Fig. 10.1. As you can see, due to DMA overhead the benefit of CGRA is not as good as 16 times better as it should have been. Also lower utilization of 80% affect the performance benefits. For large matrices, ARM suffers cache misses which increases execution time which is not the case for CGRA. The power and energy efficiecny numbers are not available at this time of the development.

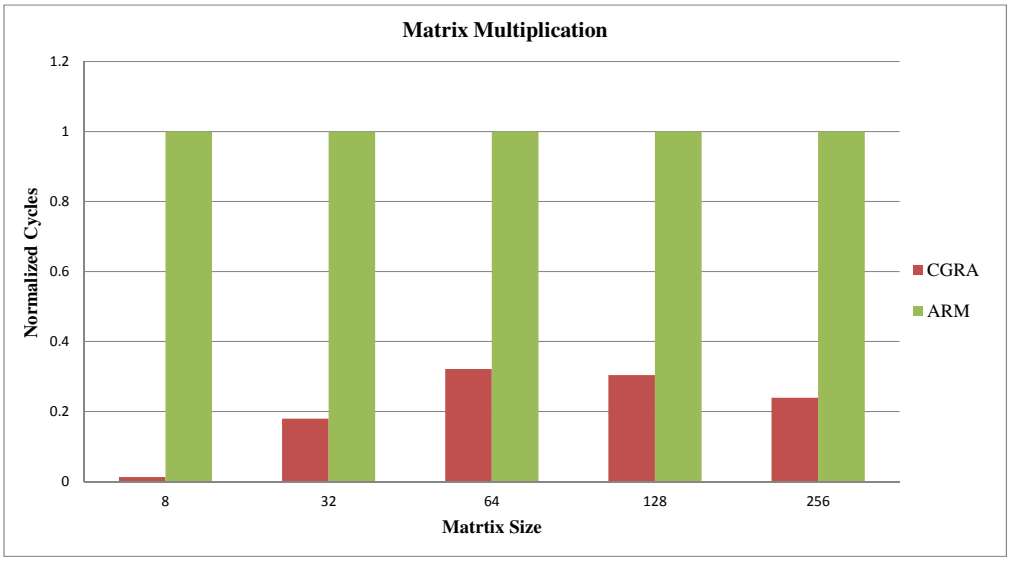

Figure 10.1: CGRA and ARM matrix multiplication comparison in cycles

Chapter 11

CONCLUSION

Intelligent use of architectural support for reducing power consumption requires a detailed task-level power profiler. In this paper, we proposed an online task-level power profiler, and discussed an experiment methodology to develop detailed power models. This methodology eliminates the need to have a detailed chip layout to get build accurate power models. The proposed profiler was integrated within Linux OS and validated with Intel Sandy Bridge processor with more than 96% accuracy and with less than 1% overhead. We also demonstrated the importance and flexibility of proposed tool to facilitate – (1) task characterization for improved energy efficiency in heterogeneous architectures, and (2) intelligent fine grained, per core DVFS features. Coarse Grain Reconfigurable Architecture provides much desired combination of performance and energy efficiency. In future SoCs where the power consumption will be of prime concern, the CGRA can play an important role to achieve higher energy efficiency. The CGRA features like enhanced compiler control, simple hardware design and inherent parallelism motivates researchers to do many architectural and software level research. Towards achieving this and to facilitate hardware designers to evaluate different architectural design evaluation and software developers to develop most suitable programming model, I developed a full system hardware-software co-design framework for CGRAs. Proposed framework is capable to detecting the CGRA as an accelerator in the Linux with help of the CGRA device driver I developed. It also allows efficient memory transfers using the peripheral DMA. I demonstrated its usefulness with a parallel matrix multiplication as an example application.

REFERENCES

[1] ARM. Whitepaper: Big.little processing with arm cortex-a15 and cortex-a7, 2012.

[2] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[4] J Cioffi and Thomas Kailath. Fast, recursive-least-squares transversal filters for adaptive filtering. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 32(2):304–337, 1984.

[5] Jason Flinn and Mahadev Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA'99. Second IEEE Workshop on*, pages 2–10. IEEE, 1999.

[6] Reiner Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the conference on Design, automation and test in Europe*, pages 642–649. IEEE Press, 2001.

[7] Reiner W Hartenstein and Rainer Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *Design Automation Conference, 1995. Proceedings of the ASP-DAC'95/CHDL'95/VLSI'95., IFIP International Conference on Hardware Description Languages; IFIP International Conference on Very Large Scale Integration., Asian and South Pacific*, pages 479–484. IEEE, 1995.

[8] John R Hauser and John Wawrzynek. Garp: A mips processor with a reconfigurable coprocessor. In *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 12–21. IEEE, 1997.

[9] Intel. Whitepaper: Advanced configuration and power interface specification.

[10] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, August 2012.

[11] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In

Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pages 359–370. IEEE Computer Society, 2006.

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

[13] Aman Kansal and Feng Zhao. Fine-grained energy profiling for power-aware application design. *ACM SIGMETRICS Performance Evaluation Review*, 36(2):26–31, 2008.

[14] Hyuk-Jae Lee, James P Robertson, and José AB Fortes. Generalized cannon's algorithm for parallel matrix multiplication. In *Proceedings of the 11th international conference on Supercomputing*, pages 44–51. ACM, 1997.

[15] W. Liao, L. He, and K. M. Lepak. Temperature and Supply Voltage Aware Performance and Power Modeling at Microarchitecture Level. *IEEE Trans. Computer-Aided Design*, 24:1042–1053, 2005.

[16] Andreas Merkel and Frank Bellosa. Task activity vectors: a new metric for temperature-aware scheduling. *ACM SIGOPS Operating Systems Review*, 42:1–12, 2008.

[17] Ethan Mirsky and Andre DeHon. Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pages 157–166. IEEE, 1996.

[18] Takashi Miyamori and Kunle Olukotun. Remarc: Reconfigurable multimedia array coprocessor. *IEICE Transactions on information and systems*, 82(2):389–397, 1999.

[19] Douglas C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[20] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *Networking, IEEE/ACM Transactions on*, 1(3):344 –357, jun 1993.

[21] Hyunchul Park, Kevin Fan, Manjunath Kudlur, and Scott Mahlke. Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures.

In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 136–146. ACM, 2006.

[22] Hyunchul Park, Kevin Fan, Scott A Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 166–176. ACM, 2008.

[23] 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 *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42. ACM, 2012.

[24] Abhinav Pathak, Y Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, pages 153–168. ACM, 2011.

[25] DMA PrimeCell. Controller (pl081) technical reference manual, 2005.

[26] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro*, 32(2), 2012.

[27] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, 49(5):465–481, 2000.

[28] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[29] S. Wang, Y. Chen, W. Jiang, P. Li, T. Dai, and Y. Cui. Fairness and interactivity of three cpu schedulers in linux. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA'09. 15th IEEE International Conference on*, pages 172–177. IEEE, 2009.

[30] Peter Whittle, Peter Whittle, Peter Whittle, Peter Whittle, New Zealand Mathematician, and Great Britain. *Prediction and regulation by linear least-square methods*. English Universities Press London, 1963.

[31] C.S. Wong, I. Tan, R.D. Kumari, and F. Wey. Towards achieving fairness in the linux scheduler. *ACM SIGOPS Operating Systems Review*, 42(5):34–43, 2008.

[32] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Accuracy of performance counter measurements. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 23–32. IEEE, 2009.

[33] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114. ACM, 2010.