

Energy-efficient Scheduling for Heterogeneous Servers

in the Dark Silicon Era

by

Sankalp Jain

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

Approved April 2015 by the
Graduate Supervisory Committee:

Umit Y. Ogras, Chair
Siddharth Garg
Chaitali Chakrabarti

ARIZONA STATE UNIVERSITY

May 2015

ABSTRACT

Driven by stringent power and thermal constraints, heterogeneous multi-core processors, such as the ARM big-LITTLE architecture, are becoming increasingly popular. In this thesis, the use of low-power heterogeneous multi-cores as Microservers using web search as a motivational application is addressed. In particular, I propose a new family of scheduling policies for heterogeneous microservers that assign incoming search queries to available cores so as to optimize for performance metrics such as mean response time and service level agreements, while guaranteeing thermally-safe operation. Thorough experimental evaluations on a big-LITTLE platform demonstrate, on an heterogeneous eight-core Samsung Exynos 5422 MpSoC, with four big and little cores each, that naive performance-oriented scheduling policies quickly result in thermal instability, while the proposed policies not only reduce peak temperature but also achieve $4.8\times$ reduction in processing time and $5.6\times$ increase in energy efficiency compared to baseline scheduling policies.

Dedicated to my parents and my brother Sambhav

ACKNOWLEDGEMENTS

The thesis not only represents my work towards scheduling algorithms in heterogeneous multi-cores system but also is a record of my work in elab,ASU. I had an amazing experience at elab thanks to several people who have helped me a lot towards my research work.

I would like to express my sincere gatitude to my thesis advisor, Dr. Umit Ogras, for his patience, motivation, advice and immense knowledge. His guidance helped me in throughout my research work and writing this thesis. I would also like to thank Dr. Siddharth Garg for his ideas, insightful comments and encouragement during my research work.Special thanks to Dr. Chaitali Chakrabarti for taking out time and participate my thesis defense committee.

I would like to thank Harshad Navale for his immense help in implementing the algorithms and designing the experiments on multi-core systems. I am profoundly thankful to Bharatwaj Raghu in assisting me in setting up the Clucene search engine and providing daily job arrival rate for servers. Gaurav Singhla helped a lot in the initial board set-up and by the simulating discussions we had. Our weekly elab meetings assisted my research work, namely by Ujjwal Gupta, Spurthi Korrapati, Navyasree Matturu and Rishabh Kesari.

My brother Sambhav has been instrumental in providing feedback on this work as well. My parents deserve most of the credit for this work for their endless love, support and understanding even though I am far away from them.

I would like to thank Dr. Umit Ogras, Dr. Yu Cao and Dr. John Brunhaver for providing financial support during my masters study and giving me many opportunities to teach in classroom. My master's study at Arizona State University was more productive and I learned a lot while being a Teaching Assistant because of aforementioned professors.

TABLE OF CONTENTS

	Page
LIST OF TABLES	vi
LIST OF FIGURES	vii
CHAPTER	
1 INTRODUCTION	1
1.1 Motivation	3
1.2 Proposed Policies	4
2 RELATED WORK	7
3 PROPOSED SCHEDULING POLICIES.....	11
3.1 Quality Metrics	11
3.2 Queue Occupancy Based Threshold Policy.....	12
3.3 Baseline Policy and Motivation for Threshold Policies	14
3.4 Proposed Threshold Policy	15
3.4.1 Addressing SLAs using Execution Time Prediction	16
3.4.2 Using Pareto Frontier for Low-power Deadline Aware Schedul-	
ing Policy	18
3.5 Summary of the Proposed Algorithm	20
4 EXPERIMENTAL EVALUATION	23
4.1 Experimental Setup	23
4.2 Development Platform	23
4.3 Evaluation Methodology	24
4.4 Impact of Threshold on Performance and Energy Efficiency	27
4.5 Comparison of Policies	27
4.5.1 Impact of Dynamic Thresholds	31

CHAPTER	Page
4.5.2 Analysis of Low-power Deadline Aware Scheduling Policy using Pareto Graph	32
4.6 Analysis of Run-time Behavior	33
4.7 Summary of Improvements	35
5 CONCLUSION	37
REFERENCES	38

LIST OF TABLES

Table	Page
3.1 Optimal Scheduling Policies Discussed in the Thesis	22
4.1 Threads Used in the Experiments	23
4.2 Frequency Table for the Odroid XU3 Development Board	25
4.3 Summary of the Improvements for the Different Scheduling Policies	36

LIST OF FIGURES

Figure	Page
1.1 Odroid XU3 Development Board	2
1.2 Thermal Profile for the Baseline and Proposed Policies.	3
1.3 Degrees of Freedom Leveraged by the Proposed Scheduling Algorithms....	4
3.1 Queue Occupancy Based Threshold Policy and Baseline Policies	12
3.2 Optimal Threshold Policy with Varying Arrival Rate	13
3.3 Measured and Predicted Execution Time for Clucene [3].....	17
3.4 Pareto Frontier for Different Configurations using Clucene Search Engine [3] as the Benchmark.	21
4.1 Odroid XU3 Development Board with Power Meter	24
4.2 Average Service Time and Energy Efficiency for Static Out of Order Schedul- ing	26
4.3 Average Service Time Normalized to that of " <i>Little Big Opportunistic (25.2 s)</i> " for the Proposed and Baseline Scheduling Polices.....	28
4.4 Cumulative Distribution Function Plot of the Different Configurations for Service Time.	29
4.5 Total Platform power for the Proposed and Baseline Scheduling Polices.	29
4.6 Energy Efficiency for the Proposed and Baseline Scheduling Polices.	30
4.7 Power Profile of Static and Dynamic Scheduling Polices.	31
4.8 Changing the Number of Active Cores Using Pareto Graph According to the Workload	32
4.9 Temperature and Queue Occupancy for Different Algorithms	34
4.10 Number of Big Cores Active for "Little + Big Opportunistic" and Dynamic Out of Order Time Algorithms.	35

Chapter 1

INTRODUCTION

Technology scaling enables a greater number of transistors, hence processing cores, to be integrated on a single chip by approximately doubling the number of transistors every 1.5 years[19]. However, designs are now primarily limited by power and not area due to increasing power density, which results in the dark silicon problem [16, 40]. *Heterogeneous* or *asymmetric* multicore processors that integrate high-performance “big” and more energy efficient “little” cores on the same chip [27, 33, 5] have been proposed to utilize the abundance of transistors in the dark silicon era. An instance of an asymmetric big-little processor is now available commercially [5, 35]. The goal is to “*deliver peak performance capacity, higher sustained performance, and increased parallel processing performance, at significantly lower average power.* [5]”. The underlying idea is to use the small cores for lightweight tasks, and the big cores for computationally demanding tasks. The problem of determining the type and number of cores as well as their optimal frequencies under general workloads remains an open challenge. Existing solutions implemented in the OS kernels rely on heuristics based on CPU utilization [34, 33]. Determining when exactly to use which core, however, remains an open challenge.

The goal of this paper is to study optimal scheduling policies for heterogeneous processors that are used as Micro-servers A scheduling policy is an online algorithm that assign jobs waiting in a queue to available servers, in this case either to the fast or slow core, as illustrated in Figure 3.1. When the goal is to minimize mean sojourn time (the sum of the time the job spends waiting in the queue and executing on a core), it would seem intuitive to schedule jobs (from the head of the queue) to *any* of the two cores, big or little, that is available to exploit all available computing resources. However, seminal work from Lin

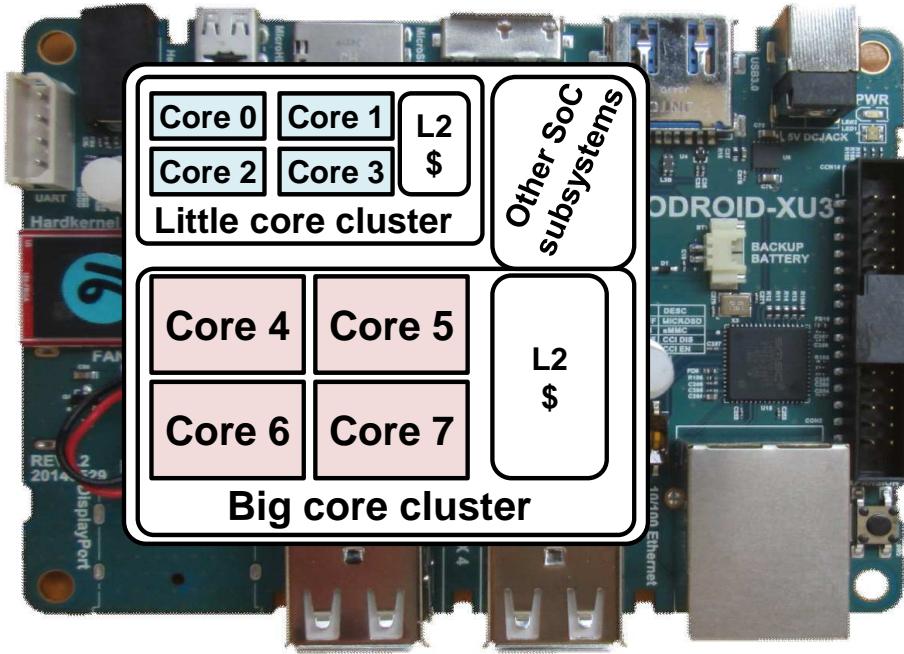


Figure 1.1: The state-of-art big-LITTLE platform [1], with four big and four little cores, used in our experiments.

and Kumar [31] has shown that the optimal scheduling policy is of **threshold type**.

In this paper, we focus on *heterogeneous* Microservers and use web search as our motivational application. Microservers, which are powered by low power processors traditionally used in mobile systems, are increasingly being adopted in datacenters for power and cost reasons [22, 38]. More precisely, we study scheduling policies for heterogeneous processors that serve a stream of search queries with the goal of meeting pre-specified service-level agreements (SLA). This study is indeed timely since power consumption and cost considerations have pushed low power ARM and Intel Atom processors, such as the one used in this work (Figure 1.3(b)), to the *Microserver* domain [22, 38]. Integration of big and little cores on a single chip results in a large dynamic range both in power consumption and performance by dynamically controlling the type, number, and frequency of active cores. Consequently, traditional scheduling policies, which have been typically

limited to *homogeneous* servers with the sole goal of performance optimization, need to be reconsidered by accounting for heterogeneity and energy efficiency explicitly.

1.1 Motivation

Scheduling policies that focus only on performance can quickly exceed the chip thermal budget and result in system shut-down for dark silicon chips. Indeed, the default scheduling policy on the commercial big-LITTLE platform shown in Figure 1.1, opportunistically uses more big cores as jobs arrive, and then turns on little cores when the big cores are fully utilized. Consequently, the core temperature spikes quickly causing the platform shut down *despite the fan*, as depicted in Figure 1.2. An alternative is to utilize the little cores first, and start using the big cores only after all the little cores are fully utilized. Although this prevents system shut-down, transient temperature spikes, shown in Figure 1.2

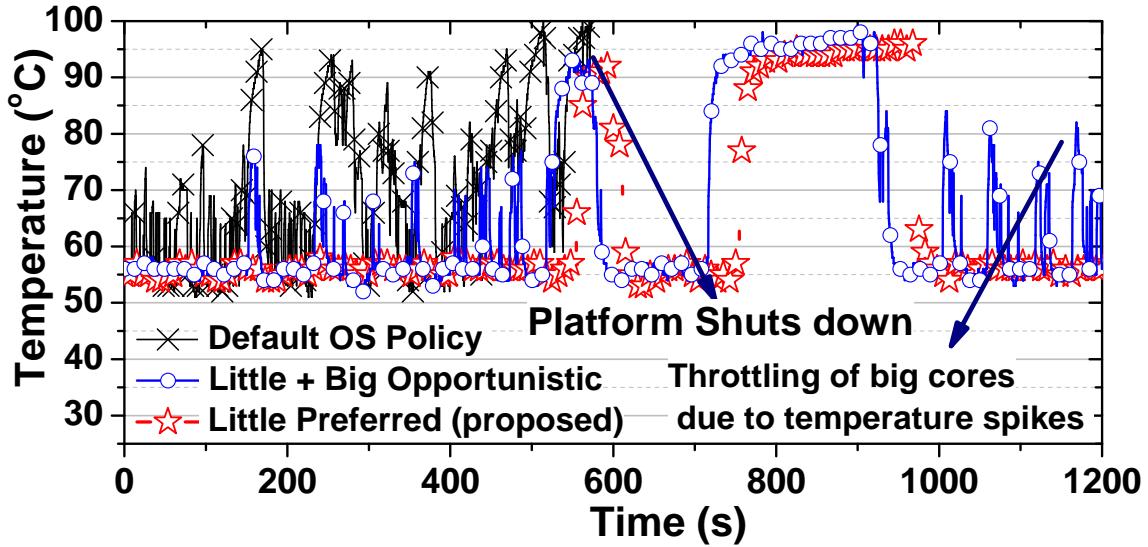


Figure 1.2: The baseline policies (default OS policy and Little + Big Opportunistic) result in either system shut-down or performance throttling due to over-utilization of the big cores. In contrast, the proposed policies results in thermally safe operation with significant benefits in both performance and energy efficiency.

(blue \circ markers), cause the big cores throttle, thus reduce performance. In contrast, the policies proposed in this paper avoid both system shut down and performance throttling by *judiciously* determining when to use big cores, as seen in Figure 1.2 (red \star markers).

1.2 Proposed Policies

The motivational example above illustrates that indiscriminate use of big cores results in thermal instability. The proposed policies activate big cores judiciously based on the following two criteria: (i) The number of outstanding search queries exceeds a certain threshold (these are referred to as *threshold policies* in literature); (ii) There are “critical” jobs in the queue that may miss their deadlines.

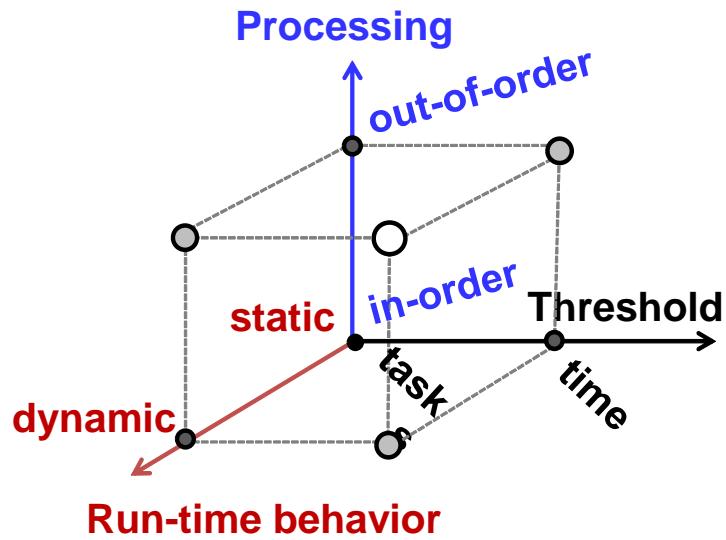


Figure 1.3: Degrees of Freedom Leveraged by the Proposed Scheduling Algorithms.

The queue-based threshold serves as a measure of workload intensity, and allows the scheduling policy to adapt to dynamic variations in arrival rate. When there is only one big and little core, the threshold policy that preferentially uses the big core and only uses the little core when the occupancy exceeds a threshold has been shown to minimize mean ser-

vice time In Figure 1.3 (static,in-order,task), reduces to a special case whose performance optimality is proven theoretically by the seminal work of Lin and Kumar [31]. However, we show that this policy leads to thermal instability. In contrast, we will show that by preferentially using the little cores and judiciously activating the big cores, the proposed policies result in not only thermally-safe behavior, but also significantly improve power and energy-efficiency compared to baseline approaches. What is more, we also account for Service Level Agreements (SLAs) by explicitly checking for deadline violations, and scheduling the critical jobs to the 7big cores. Of note, determining criticality requires a predictor for the execution time of outstanding queries in the queue. Hence, we designed and implemented such a prediction mechanism for web-search queries. In the absence of an execution time predictor, the proposed threshold-based scheme can still be used, and provides significant improvement over the state-of-the-art. In general, the set of policies can operate at all of the vertices of the design space shown in Figure 1.3. Specifically, the intensity of the workload is measured either by the number of jobs in the queue, or by the total predicted execution time for all the pending jobs. The latter, which is more accurate, is feasible when the execution time predictions are viable. Second, smaller tasks could be preferably scheduled to little cores, if out of order scheduling is permitted. Finally, the scheduling threshold, as well as in- or out of order processing choice, can be static or adjusted dynamically depending on the workload (in the server setting, this corresponds to the rate at which jobs arrive for processing). The proposed approach encapsulates these design choices in a single, overarching framework and enables performance, power and energy optimization by leveraging these opportunities. However, as noted in Figure 1.2, policies that only target performance lead to thermal instability for dark silicon chips. Ours is the first work that considers threshold-type policies to optimize energy efficiency for servers with multiple big and little cores, while leveraging there are no known optimality results considering the degrees of freedom in Figure 1.3.

The novel contributions of this paper are:

- We show that using big and little cores opportunistically, like the default Linux policy, quickly raises temperature and causes frequency throttling or worse, system shut-down.
- We propose novel scheduling policies for heterogeneous Microservers that result in thermally-safe behavior while optimizing SLAs including mean response time and percentage of queries that are served by a specified deadline.
- We implement and *experimentally validate* the proposed policies on an octa-core big-LITTLE hardware platform [1] using workload intensity traces from production data-centers [12] and web search benchmark.
- Experiments on real hardware show $4.8\times$ boost in performance and $5.6\times$ increase in energy efficiency.

The rest of the paper is organized as follows. Related work is presented in Chapter 2. The Proposed scheduling policies are explained in Chapter 3.Finally, Extensive experimental evaluation using Samsung Exynos XU3 octa-core chip is performed and shown in Chapter 4 and conclusions appear in Chapter 5.

Chapter 2

RELATED WORK

Multi-cores SoC architectures are fast becoming next big thing in both server and desktop processors. In next 20-30 years, processors with tens and even hundreds of cores on one chip should be seen [9, 17]. The most important part of MPSoCs is parallel processing, which is assigning tasks or jobs to processors. The purpose of task scheduling in MPSoCs are maximizing performance by assigning correct tasks to correct processors and optimizing energy consumption.

On homogeneous multi-cores systems there are several studies done on known scheduling algorithms [24, 28, 11] assuming tasks with a uniform arrival rate. Normally the performance of these popular scheduling algorithms are considered:

- *Min-min* [21]: Min-min algorithm is a simple heuristic, which selects a task with the minimum execution time on any processor from the set of unmapped tasks, and schedules it onto the processor on which it has the minimum completion time. The algorithm is very simple and easy to implement but it requires execution time of all the tasks and the time at which task is going to execute.
- *Max-min* [23]: Max-min algorithm is similar to Min-min algorithm but it gives preference to the task with the longest earliest finish time. If the number of short tasks are much more than long tasks, hen long tasks will be scheduled first and executed with many short tasks. This algorithm has certain limitations which arise if number of short tasks are less or equal to large tasks. [23] shows that this algorithm is better than Min-min algorithm.
- *Chaining* [14]: Chaining algorithm is proposed by Djordjevic and Tasic, which is a single pass deterministic algorithm based on list scheduling techniques except in this case task are mapped irrespective of the task dependencies. The scheduling in this case

is dependent on the time to communicate and execution time. It assumes execution time of each of the tasks at every point is known.

- *A** [29]: The A* algorithm is a best-first search algorithm, originated from the field of artificial intelligence. It is a tree-search algorithm which starts with a null tree and the tree is expanded by allocating a task to all possible processors. It always provides a partial solution and has an exponential execution time.
- *Simulated annealing* [15]: Simulated Annealing is based on Monte Carlo approach on optimization functions. At each stage, a solution is generated by randomly modifying the current solution i.e. remove a task, or switch the two tasks, and it is evaluated based on initial parameters and if considered a better solution the parameters are updated.
- *Deadline based task scheduling policies*: Some of the traditional algorithms are Earliest Deadline First (EDF) [7], Round Robin Scheduling Algorithm (RR) with new algorithms like Prioritized Deadline based Scheduling Algorithm (PDSA) using project management technique for efficient job execution with deadline constraints [2].

The algorithms mentioned above only considered execution time and load balancing but they neglect the energy consumption. In this era, optimization of energy optimization is equally important as we cannot have high power-consuming algorithms. There are several algorithms where energy consumption is taken into account such as [43] which calculates the priority of task based on execution time and system energy consumption, where priority is calculated at every instant making the overhead quite high. It is also expected to know power consumption of tasks beforehand. Energy-efficient algorithms with duplication based method using ant colony algorithm is considered in [46].

In reality clusters in servers and datacenters are *heterogeneous* in terms of the performance, capacity and power consumption in terms of CPU cores. Kumar et al. [27] and by Viniotis et al. [41] are among the first studies to discuss and evaluate the benefits of **asymmetric (or heterogeneous)** multi-core processors for general purpose computing. Since

then several research efforts have been dedicated to determining run-time scheduling techniques to map application threads to cores [8, 26, 39]. However, none of these techniques consider dynamic job arrivals or queueing effects, which are critical in the server/datacenter context. Closer to our work, Gupta and Nathuji [19] have looked at datacenter servers with many little cores and one big core, and have modeled service time using M/M/1 model, i.e., assuming only one job runs on the server at any given point in time. In contrast, our work allows concurrent execution of multiple jobs, for instance, search queries with multiple little and big cores. The work in [37] looks at a similar problem, but assumes only one cluster is on at a given time. However, the entire processor is modeled as a single server running only one job at any given time, where the jobs itself are multi-threaded and the big core is used to accelerate serial sections of the jobs.

With the introduction of commercially available advent of asymmetric or heterogeneous multi-cores like the ARM big.LITTLE architecture, there has been increased interest in designing performance optimal and energy efficient scheduling policies for heterogeneous processors. Recent work has proposed energy efficient scheduling policies for mobile web browsing on an ARM big.LITTLE processor [47, 20]. These techniques schedule jobs to exploit the variations in energy consumption and time required to load different web pages. Power/performance modeling of asymmetric multi-core architectures has been addressed in [36]. The authors present a software-based approach to estimate power consumption and performance for different core types. Likewise, power management of big-little processors is considered in [33], but is focused on dynamic core-count and voltage/frequency scaling during the execution of a single job, but does not model job arrivals or queuing effects. However, this work focuses on controlling the set of active cores, voltage/frequency of the cores and task migration. Similar to the previous papers on scheduling for heterogeneous processors, it does not consider a server setting and hence job arrival process and queuing effects are not modeled.

From a theoretical perspective, Walrand [42] and Koole [25] provide alternative (and simpler) proofs for the results of Lin and Kumar. The threshold based policies are proven to be energy-efficient with N thresholds policy for multiple servers in [6] with comparison of different policies, where it is shown theoretically energy efficiency gain is achieved with 1 or 2 thresholds. Nonetheless, extending the result showing the optimality of the threshold policies to settings with more than two servers has proven surprisingly hard.

Efficiency of big and small cores in web search has been analyzed in [22]. Finally, we note that heterogeneity in processing capabilities has also been discussed at the full datacenter level [18, 13], instead of a single server/processor level as we do in this paper. These works do model job arrivals and queuing effects, but consider entirely different types of scheduling policies than those that this thesis research consider.

In this dark silicon era, we should also consider temperature aware scheduling algorithms with energy optimizations as thermal constraints play an important role. There are algorithms which do use task migration to get uniform thermal map based on HotSpot thermal model [44] or by using Dynamic Voltage/Frequency Scaling (DVFS) and Dynamic thermal management(DTM) or Heuristic algorithms [45]. Dynamic thermal management primarily by throttle the processor activity by either reducing frequency (or voltage) or restricting the operation of a core.

In the thesis, a family of threshold policies for multiple big-LITTLE servers are developed and the optimality of them for performance and energy optimization is shown in thermally-safe behavior while still meeting SLAs. It can be concluded that starting with low power cores and improving performance is a better strategy than starting with high performance cores. The algorithms are targeted towards heterogeneous cores but they can work for asymmetric cores like Snapdragon™ 800 also as experimentally evaluated.

Chapter 3

PROPOSED SCHEDULING POLICIES

3.1 Quality Metrics

We formally define the quality and SLA metrics before presenting our problem formulation.

Definition 1. *Average service time* is the sum of queuing time and execution time on the server averaged over all queries.

Definition 2. *Energy efficiency (tasks/J)* is defined as the ratio of the throughput (tasks/second) and the CPU power (W).

Definition 3. *Total platform power (W)* is the power consumed by the whole platform including the CPU, fan and peripherals.

Definition 4. The *SLA* refers to an agreement between a user and a service provider (the server), that provides guarantees on quality-of-service (QoS). The SLA that we use in this paper is the percentage of jobs that with a service time less than a pre-specified deadline.

The server platform considered in this paper is shown in Figure 3.1. As new search queries arrive, they wait in a software queue for service. Then, the jobs are serviced by two types of cores which are clustered as “big” (fast, power hungry) and “little” (slow, low power). At any given point in time, a core executes only one search query. When one or more cores are idle and there are outstanding jobs in the queue, the scheduling policy (also implemented in software) decides which job(s) in the queue are allocated to which cores. The servers in the big cluster are faster, hence more power hungry, while those in the little cluster are more energy efficient. This model is general in the sense that big and little clusters can be on a single chip as in [5, 35] or they can be different machines. However,

experimental evaluation in this paper focuses on a single chip implementation.

Next, we present the scheduling policy that begins with a queue-occupancy based threshold to determine activation of big cores, and then incorporate the deadline constraints to meet the SLA.

3.2 Queue Occupancy Based Threshold Policy

We start off with the scenario in which the scheduler knows only the mean execution times on the big and little cores, and services jobs in the order in which they arrive (these

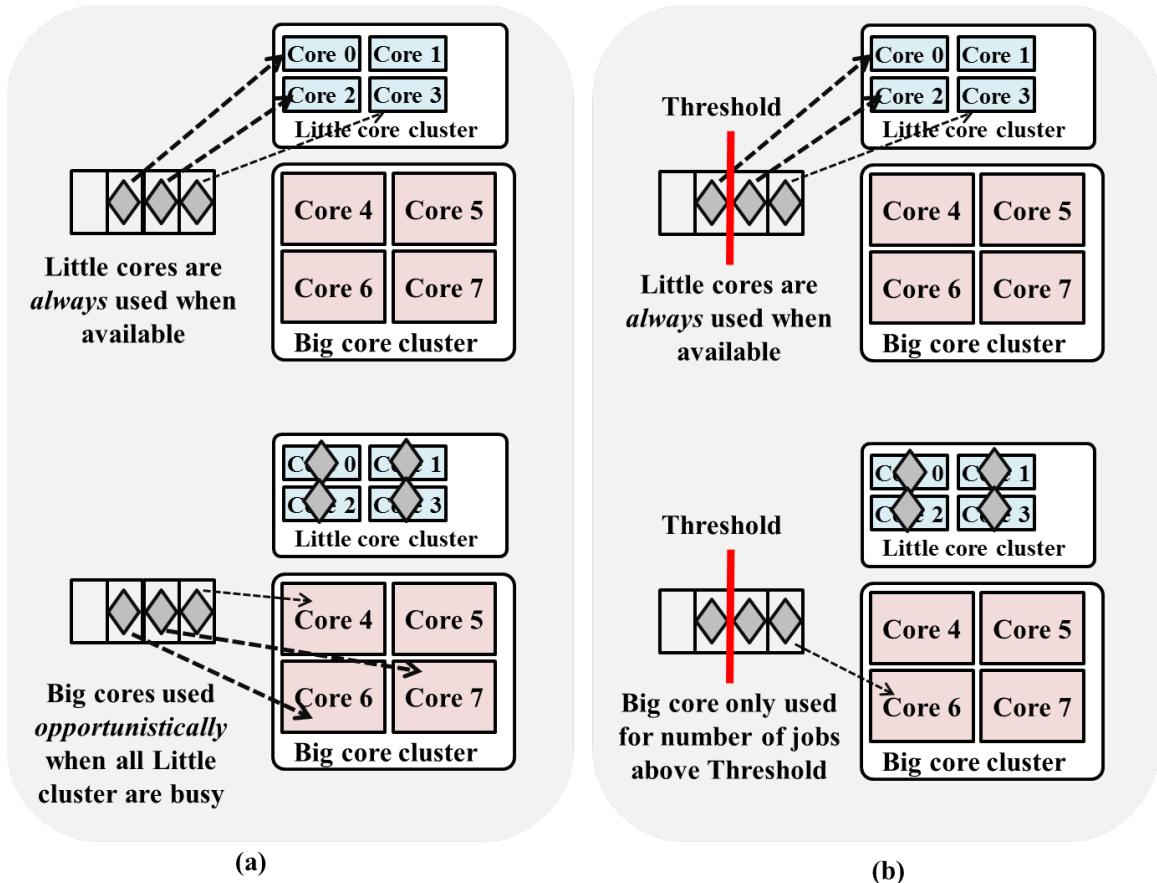


Figure 3.1: (a) Baseline Little+Big Opportunistic Policy: Little cores are used first and big cores when the all little cores are busy. (b) Threshold Policy: Little cores are used first, and big cores only for the number of jobs beyond a threshold.

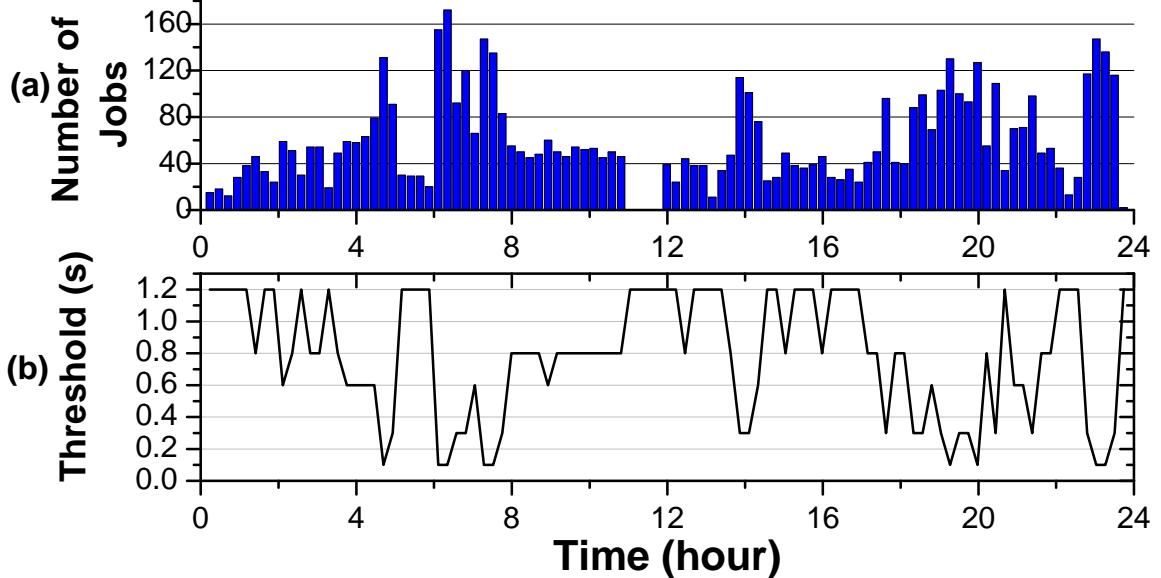


Figure 3.2: Number of Jobs (Histogram) Per Hour Production Datacenter Trace [12], and the Optimal Threshold Policy (Time-Based Threshold) with Varying Arrival Rate.

constraints will be relaxed later). Under this assumption, when there is at least one idle core and one job waiting in the queue, the scheduler can make one of the following two decisions:

- *Send*: The job at the head of the queue is sent to an available core. If both types of cores are available, the scheduling policy decides whether to send the job to the big or the little cluster.
- *Wait*: The scheduling policy chooses to wait rather than scheduling a pending job to an available core.

Definition 5: If there are idle cores/servers of both types, i.e., big and little, the scheduler needs to choose one type over the other. The type which is prioritized by the scheduler is called the *preferred server*. In this paper, all of our proposed policies are “little preferred” policies since we found that no “big preferred” policy is thermally stable.

3.3 Baseline Policy and Motivation for Threshold Policies

Intuitively, it might seem that performance is improved by exploiting any available server, i.e., by always taking the “*send*” action. In other words, whenever the job queue is non-empty, the baseline policy always takes the “*send*” action. When the goal is to minimize the service time (performance), the scheduler can utilize first all the big cores, and then use the little cores — this is, *in essence, what the default OS scheduler does*. Alternatively, the scheduler can prefer first the little cores, if the primary goal is minimizing power. However, we empirically observed that policies in which big cores are preferred are *not* thermally stable. Therefore, the *baseline policy* we consider first utilizes the little cores, and starts using the big cores only after all little cores are busy, as shown in Figure 3.1(a). Meanwhile, any unused core is put to sleep state to save power.

We note that using both cores opportunistically *does not* necessarily provide optimal performance. Suppose that there is a job waiting in the queue and all of the fast servers are busy while there is an idle slow server. Execution time of the slow server can be longer than the waiting time for a fast server plus the execution time in the fast server. Consequently, even a small increase in the service rate, in particular when the arrival rate approaches to the service rate, can shrink the service time significantly. For example, consider the average waiting time in an M/M/1 queue, $W = 1/(\mu - \lambda)$. The change in the waiting time as a function of the service rate can be found as $dW/d\mu = -1/(\mu - \lambda)$. Hence, the savings in waiting time is significant in particular when $\lambda \rightarrow \mu$. Hence, the effective service rate can be improved by waiting for a fast server.

This intuition has been formalized by Lin and Kumar [31] for one fast and one slow server. They showed, under ideal conditions, that the optimal scheduling policy is indeed of “threshold type” wherein the faster server is used opportunistically but the slower server is only used when the number of jobs in the queue exceeds a certain threshold. However,

their results have not been empirically validated so far with real workloads. Moreover, these results have not been generalized to multiple servers, for time-varying job-arrival rates, or taking into account power and energy efficiency as metrics. The framework presented next captures this result as a special case when it is used for a single fast and single small server and the scheduling decision is limited to static, in-order processing with threshold specified in number of tasks (see Figure 1.3). Therefore, we present threshold policies next.

3.4 Proposed Threshold Policy

When the size and execution time of the jobs cannot be predicted before the execution starts, we need to rely on the number of jobs that are already waiting in the queue, and their order of arrival. Hence, our problem formulation is:

Optimize the average service time and energy efficiency ***subject to*** thermal constraints.

In the proposed scheduler, we set the little cores as the preferred server to address temperature constraints and energy-efficiency (see Algorithm 1). Since (for now) we assume that the scheduler cannot predict the execution time of search queries in advance, it simply schedules jobs in the order in which they arrive.

The proposed threshold policy is illustrated in Figure 3.1. If there is a job waiting at the head of the queue *and* a little core is available (*Case A*), the job is *always* sent to the little core. If all little cores are busy *and* there is at least one more job waiting in the queue, the job is sent to a big core ***only if*** the queue length is greater than a threshold t . Otherwise, the job is held in the queue. With respect to Figure 1.3, this corresponds to a threshold in terms of number of tasks.

According to the theoretical analysis of Lin and Kumar theory [31], threshold policy is optimum under constant arrival rate. However, real life data shows significant variations as depicted in Figure 3.2 (a). Therefore, we also consider a set of dynamic threshold policies where the thresholds are updated dynamically as a function of job arrival rates. To this

Algorithm 1 Pseudo-code for the task based threshold scheduling policy.

```
1: procedure SCHEDULETASK PreferredServer
2:   PreferredServer  $\leftarrow$  idle
3:   NonPreferredServer  $\leftarrow$  idle
4:   while TaskQueue is not empty do
5:     if PreferredServer is idle then
6:       Schedule the next job to the PreferredServer
7:
8:     if (TaskQueue is not empty) AND
9:       (NonPreferredServer is idle) AND
10:      (There is no thermal violation) AND
11:      (TaskQueueSize  $\geq$  Threshold) then
12:        Schedule the next job to the NonPreferredServer
```

end, we first characterized the optimal threshold for each algorithm as a function of a constant rate, and stored this information. Since the optimal threshold does not change abruptly, the optimal thresholds for five different arrival rates that cover the whole range in the empirical data as shown in Figure 3.2 (b). Then, the arrival rate tracked at run-time is used to dynamically control the threshold.

3.4.1 Addressing SLAs using Execution Time Prediction

For many workloads including web search, the execution time of a job can be predicted ahead of time based on its characteristics. For instance, a search query execution time predictor, that makes predictions based on the number of keywords in the search query, is presented in [10]. Following this approach, we performed an offline analysis by sweeping the number of keywords in Clucene search engine [3] and recording the execution time.

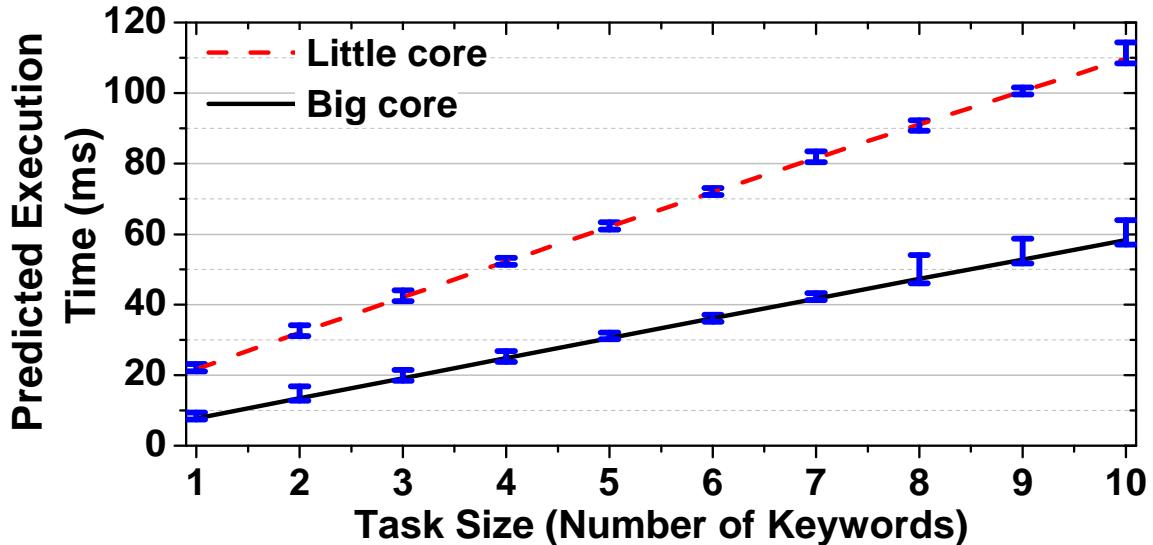


Figure 3.3: Measured and Predicted Execution Time for Clucene [3].

Then, we constructed an accurate linear predictor for execution time as a function of the number of key words as shown in Figure 3.3.

Execution time prediction enables two additional optimizations, as detailed below.

- *Time (instead of Task) Based Threshold:* So far, we have expressed the threshold in terms of the number of tasks waiting in the queue. Using execution time prediction, the threshold can be expressed in terms of absolute time instead of number of tasks. In other words, the total execution time of jobs is computed and compared with the time based threshold to determine when to schedule jobs on the big cluster. The time-based threshold is set, as before, to minimize mean response time for a given arrival rate, and can be dynamically varied to respond to varying arrival rates. To the best of our knowledge, time-based thresholds have not been discussed in literature.
- *Deadline-Aware Out-of-Order Execution:* Assuming that the scheduler is allowed to execute jobs out-of-order, execution time prediction can be used to identify the most critical task in the queue. This is the task with the smallest difference between the deadline on the one hand, and the sum of its current waiting time and predicted execution time on the

Algorithm 2 Pseudo-code for the Deadline-Aware Out-of-Order scheduling policy.

```
procedure SCHEDULETASK PreferredServer
2:   PreferredServer  $\leftarrow$  idle
    NonPreferredServer  $\leftarrow$  idle
4:   while TaskQueue is not empty do
    if PreferredServer is idle then
6:     Schedule the next job to the PreferredServer
8:     if (TaskQueue is not empty) AND
        (NonPreferredServer is idle) AND
10:      (There is no thermal violation) AND
        (Total expected Execution Time  $\geq$  Threshold) then
12:       Schedule the most critical job to the NonPreferredServer
```

other. Thus, when scheduling a task to the big core, the scheduler preferentially executes the most critical job instead of the job at the head of the queue to maximize the number of tasks that meet the SLA. More precisely, the scheduling policy can process a job out of order, if it is more likely to miss its deadline than those in front of it. We note that there is no starvation, since execution on the little core is still in order.

3.4.2 Using Pareto Frontier for Low-power Deadline Aware Scheduling Policy

Pareto optimality [32, 30] is widely used in the world of engineering to find optimality in a multi-objective function. Generally, there can be no single global solution and therefore it is often necessary to determine a set of points that all fit a predetermined definition for optimality. Pareto optimality is the concept of defining a point in the Pareto frontier curve, which can be stated as follows

A point $x^* \in X$, is Pareto optimal iff there does not exist another point $x \in X$, such that

Algorithm 3 Pseudo-code for the Deadline-Aware scheduling policy using Pareto Frontier.

procedure DEADLINE AWARE PARETO CONFIGURATION

Starting Algorithm with 1st Configuration from Pareto Frontier graph

3: $Td \leftarrow$ Deadline for all jobs

$Tb[i] \leftarrow$ Time of execution of ith job *in Big core*

$Tl[i] \leftarrow$ Time of execution of ith job *in Little core*

6: $Tq[i] \leftarrow$ Time already spent in queue of ith job

for all Task in Queue **do**

9: $\Delta[i] \leftarrow Td - Tq[i]$

 Based on current *configuration* and *threshold*,

 it can be determined whether job will run in *Big or Little core*

12: $\Delta[i] \leftarrow \Delta[i] - (\sum_{j=0}^i (Tb[j]/Tl[j]))$

if $\min(\Delta) \leq 0$ **then**

 Go to a *higher* configuration using the Pareto frontier curve

15: Schedule the *most critical* job to a big core

if $\min(\Delta) \geq Margin$ **then**

 The system can go to lower configuration

$F(x) \leq F(x^*)$ and $F_i(x) \leq F_i(x^*)$.

Using the definition given above, a Pareto frontier graph is made by running Algorithm 2 for different configurations of Big and little cores as shown in Figure 3.4. The total CPU power is obtained using internal sensors and the average service time is reported using a multi-threaded pthread program running search Clucene queries.

The Pareto frontier curve in Figure 3.4 is used to determine how many cores should be used, so that deadline for all jobs is maintained and minimal power should be used.

A separate thread is used to constantly update the configuration, with respect to SLA of the jobs in queue, as shown in Algorithm 3. By knowing the *current configuration and threshold*, it can be known on which core job will run. For every job in the queue, Δ is calculated, by subtracting the time the job has already spent in queue and summation of execution time of all the jobs *prior* to it plus its own execution time from the *deadline*. If for any job Δ becomes negative then a higher configuration should be used as a job is going to break the SLA, otherwise a lower configuration can be selected. By accurately predicting when a job is going to break deadline, a Pareto optimal point is selected and when not required extra CPU cores are essentially turned off. This way both client is satisfied and temperature of the board under better control.

3.5 Summary of the Proposed Algorithm

Algorithm 1 & Algorithm 2 shows the pseudo-code of a family of scheduling policies that arise as a consequence of the ideas discussed above. The precise scheduling policy can be configured depending on the following choices:

- Big or little preferred, i.e., the variable *PreferredServer*,
- Static or dynamic threshold, i.e., in line 11 if a static threshold is used or dynamically updated,
- Task or time based threshold, i.e., whether the threshold is in terms of number of tasks or absolute execution time,
- In-order or out-of-order execution, i.e., whether the next job (lines 6 and 12) is scheduled from the head of the queue, or based on criticality to meet SLA.

We note that the in-order version of the algorithms are explicitly targeted towards minimizing mean response time, although empirically they also result in a significant increase in the fraction of queries serviced within the deadline. The out-of-order versions additionally aim to optimize for the SLA, but can only be used if the execution time can be predicted.

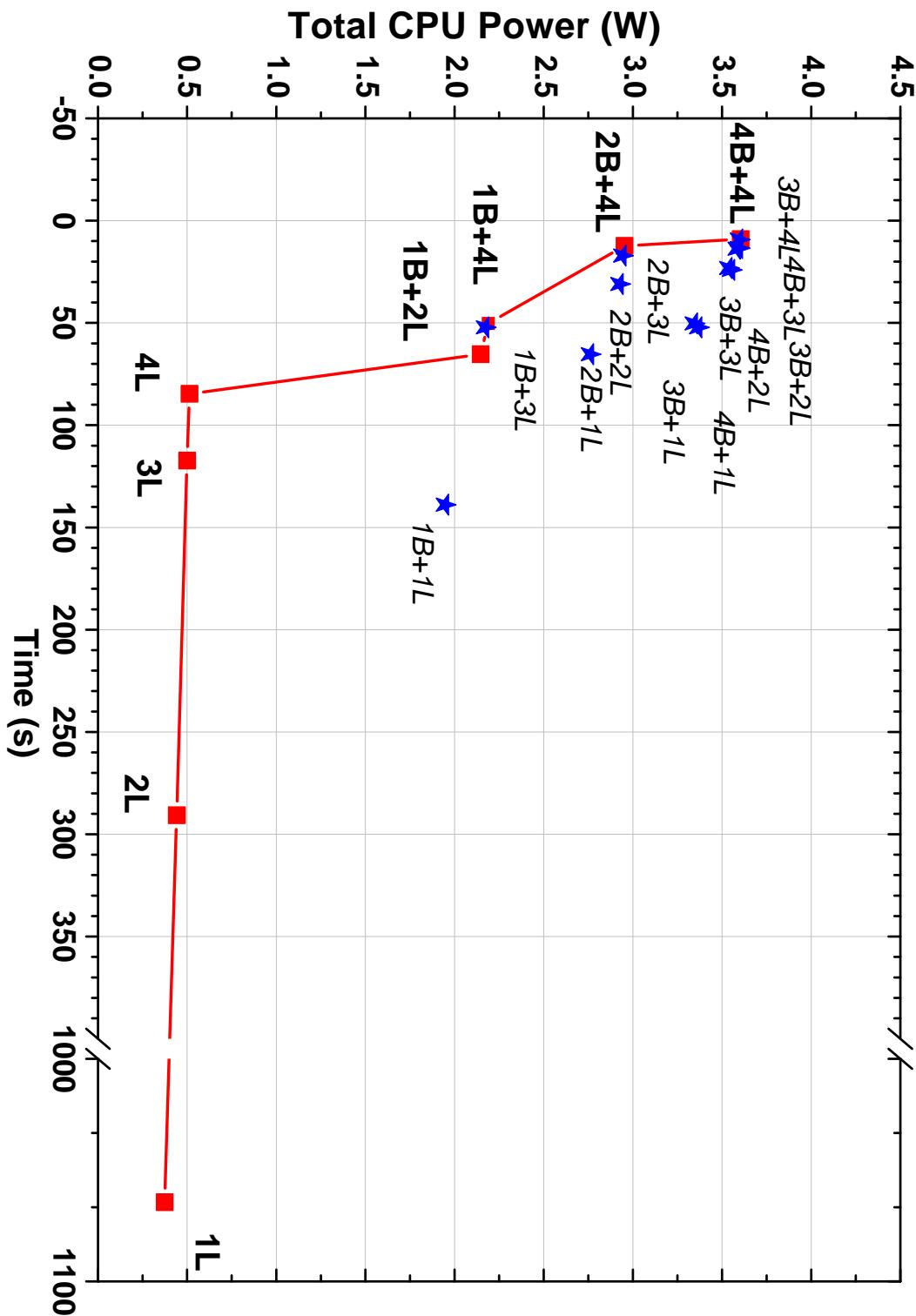


Figure 3.4: Pareto Frontier for Different Configurations using Clucene Search Engine [3] as the Benchmark.

Algorithm 3 shows a method to power consumption and thereby increase energy efficiency while maintaining SLA for all the jobs. The Pareto graph as shown in Figure 3.4 gives optimal points for multi-objective function where we have to decrease power consumption and at the same time decrease average service time.

As a final note, the proposed scheduling policies are both implicitly and explicitly thermally constrained. Implicitly, the power governor on the chip throttles the big core or shuts off the platform in case of thermal emergencies. Explicitly, in line 10 of the algorithm, we only schedule jobs on the big core if the current system temperature is below a safe value.

Each instance of the scheduling policies mentioned above are summarized in Table 3.1. We note that each one has a static and dynamic implementation.

Table 3.1: The different optimal scheduling algorithms discussed in above.

No.	Policy	Scheduling	Threshold
1	In Order, task	In Order	Number of tasks in the queue
2	In Order, time	In Order	Execution time
3	Out of Order, time	Out of Order	Execution time

Chapter 4

EXPERIMENTAL EVALUATION

4.1 Experimental Setup

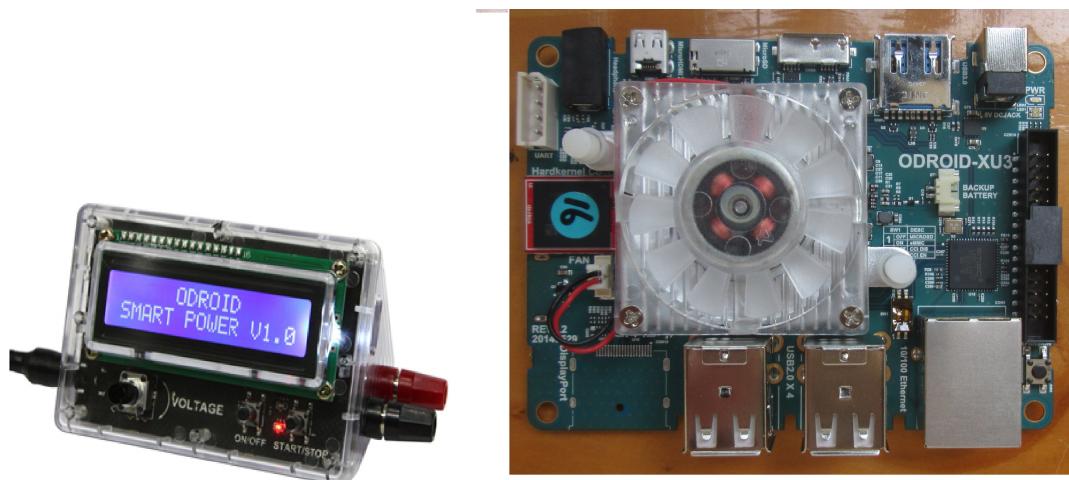
We implemented a multi-threaded program using the POSIX pthread library to evaluate the scheduling policies studied in this paper. Thread 1 generates new jobs, and then inserts them into a queue of size 512. Whenever the task queue is non-empty, Thread 2 schedules the task to one of the servers following the scheduling policies described in Chapter 3. The core implementing the scheduler goes to sleep between scheduling different jobs, and has less than 5% utilization. Finally, threads 3-6 and threads 7-10 implement the slow and fast servers, respectively, as described in Table 4.1.

4.2 Development Platform

This program is executed on a Odroid XU3 development board [1] running Linux kernel 3.10.9 on Samsung Exynos 5422 MpSoC. Samsung Exynos 5422 MpSoC has a heterogeneous multi-processing (HMP) solution with which it can simultaneously use both Big (4

Table 4.1: The threads used in the experiments, the cores they are mapped to, and their description.

Thread	Core	Description
1	0 (A7)	Generates and enqueues the tasks
2	7 (A15)	Schedules the tasks to slow and fast servers
3-6	0-3 (A7)	Slow server
7-10	4-7 (A15)	Fast server



Power meter used for measuring Total Platform Power.

Samsung Exynos 5422 with 4 A15 and A7 cores each with internal temperature and power sensors.

Figure 4.1: Odroid XU3 Development Board with Power Meter

A15 cores) and Little (4 A7 cores) clusters with internal power and temperature sensors.. The processor supports DVFS and it has default governor as *interactive*. The Big and Little cores are symmetric, meaning all cores in same cluster can have same frequency. The processor is widely used commercially in Android mobile phones and tablets. In our experiments, we measured the power consumption and temperature using the built-in sensors.

4.3 Evaluation Methodology

The enqueue thread generates search queries with varying job arrival rates taken from a commercially deployed datacenter server [12] for an entire day, as shown in Figure 3.2. The jobs stored in the queue are the number of keywords in search queries. We incorporated a keyword search application into our multi-threaded program using “Clucene search engine API” [3]. Clucene is a C++ port of Java based Lucene, open-source high-performance text search engine API, used in a lot of high traffic websites such as Twitter, Wolfram Research and LinkedIn [4]. Upon receiving a new job, the servers search as many keywords as

Table 4.2: Frequency table for the Odroid XU3 big.LITTLE development platform [1]

No.	Frequency of Big Core (A15) (MHz)	Frequency of Small Core (A7) (MHz)
1	2200	1400
2	2100	1300
3	2000	1200
4	1900	1100
5	1800	1000
6	1700	
7	1600	
8	1500	
9	1400	
10	1300	
11	1200	

specified in the job size in a 500 MB database from Wikipedia, already indexed using Clucene API. It is also used to build search capabilities for various applications such as Eclipse IDE, Nutch and companies like IBM, AOL and Hewlett-Packard. Clucene is used as it is faster than Lucene and as its written with a flexible CMake build system, cross-compilation for ARM architecture was possible. We made a database search engine using CLucene search engine library. It was incorporated into the program and cross-compiled for the ARM architecture. For each experiment, we generated jobs using entire day's job arrival data, and ran the system until all keyword searches are completed.

When we make the fast server (big core cluster) “preferred” the temperature rises steadily during high activity periods until a forced shut-down (as shown in Figure 1.2). Therefore, we evaluated the proposed scheduling policies when the slow server (little core

cluster) is preferred. We also compared the proposed policies against two baseline cases: (i) *Only Little*: The scheduling threshold is set to infinity such that only the little cluster is used, and (ii) *Little Big Opportunistic*: The threshold is set to zero such the big cores are used opportunistically whenever the little cores are fully utilized (Figure 3.1(a)).

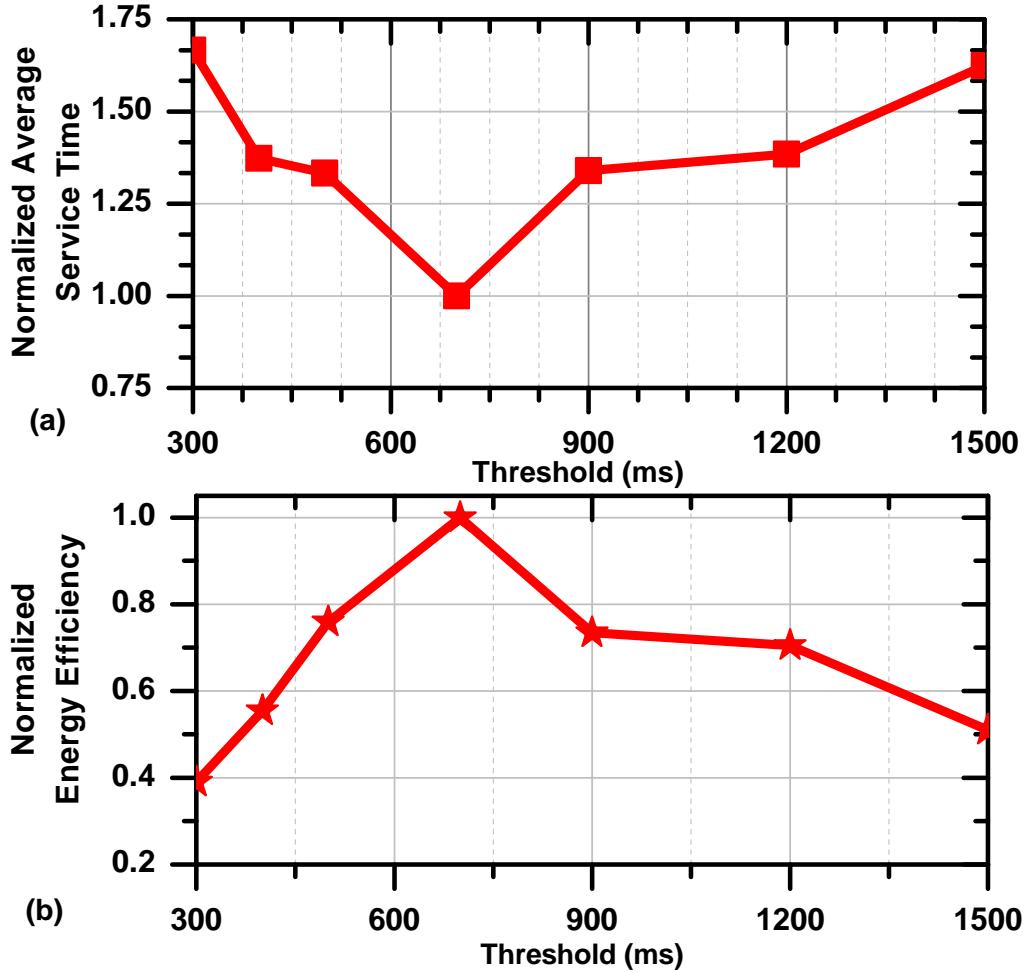


Figure 4.2: **(a)**:Normalized average service time and **(b)**: Energy efficiency for static out of order scheduling as a function of threshold expressed in absolute time with the optimal points as 10 s and 94 tasks/J respectively. Both plots demonstrate the optimal nature of the "threshold" based scheduling algorithms.

4.4 Impact of Threshold on Performance and Energy Efficiency

The threshold parameter, either in terms of number or predicted execution time of outstanding jobs, is a critical knob that determines the utilization of the big cores for our proposed policies. Figure 4.2 shows the performance and energy efficiency of the time-based threshold policy as a function of the threshold. Increasing the threshold results in lower utilization of the big cluster and should, in theory, cause the average service time to *increase* monotonically. However, note that increasing the scheduling threshold from 300ms to 700ms *improves* the average service time. This is because high utilization of the big cluster at lower thresholds results in thermal spikes that cause the big cluster frequency to be throttled, hence degrading performance. Further increasing the threshold beyond 700 ms hurts performance because of decreasing big cluster utilization. At the same time, the average power consumption decreases monotonically with threshold. Therefore, the energy efficiency peaks at the same threshold as performance, increasing from only 36 tasks/J at a threshold of 300 ms to 93 tasks/J at the optimal threshold.

4.5 Comparison of Policies

Next, we compare the proposed policies to the two baseline policies in terms of performance, power and energy efficiency.

Performance Average service time for the baseline and proposed policies is plotted in Figure 4.3. Using only the little cluster results in the highest average service time per task, while additionally using the big cluster opportunistically when the little cluster is occupied reduces the average service time by almost $6\times$. We observed *even further boost in performance by using the proposed threshold policies*, even though our proposed policies utilize the big cluster *less* than utilizing the big cores opportunistically. This is because the frequency of the big cores in the baseline policy is periodically throttled in response to thermal

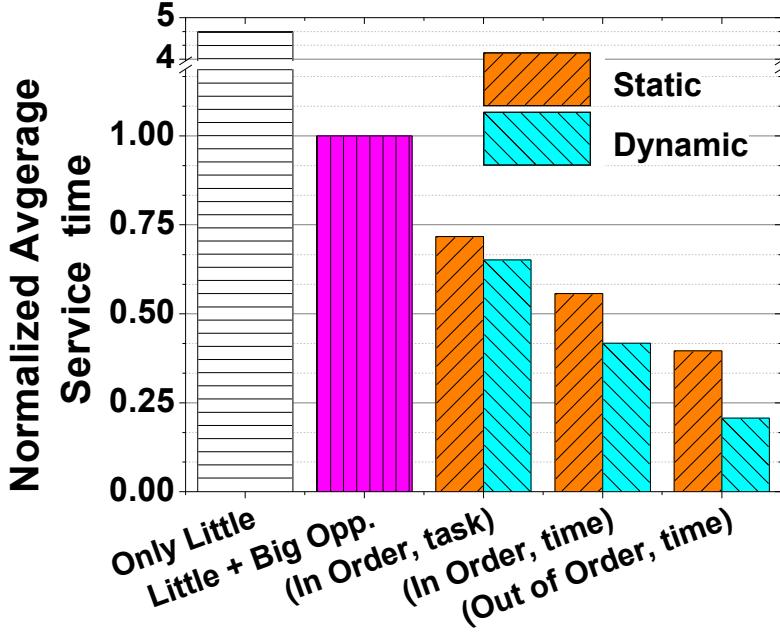


Figure 4.3: Average Service Time Normalized to that of "Little Big Opportunistic (25.2 s)" for the Proposed and Baseline Scheduling Policies.

spikes generated by the power consumption of big cores. We observed that even static in-order scheduling with task-based thresholds, the most constrained policy results in 18.1 s average service time. This results in 1.4 \times reduction in average service time compared to little+big opportunistic. Our best policy (dynamic out-of-order scheduling time-based thresholds) reduces average service time by almost 4.8 \times over little+big opportunistic.

To quantify the benefits of the proposed scheduling policies in meeting SLAs, we plot in Figure 4.4 the cumulative distribution function (*cdf*) of service time for the different policies. Observe that for any deadline constraint, our best performing policy significantly increases the fraction of jobs that meet the deadline compared to the two baseline policies. We also note that compared to in-order scheduling policies, the *cdf* curve for out-of-order scheduling has a shorter tail, illustrating that it is indeed effective in ensuring that critical jobs meet their deadlines.

Power Consumption Compared to the lowest power policy that utilizes little cores only

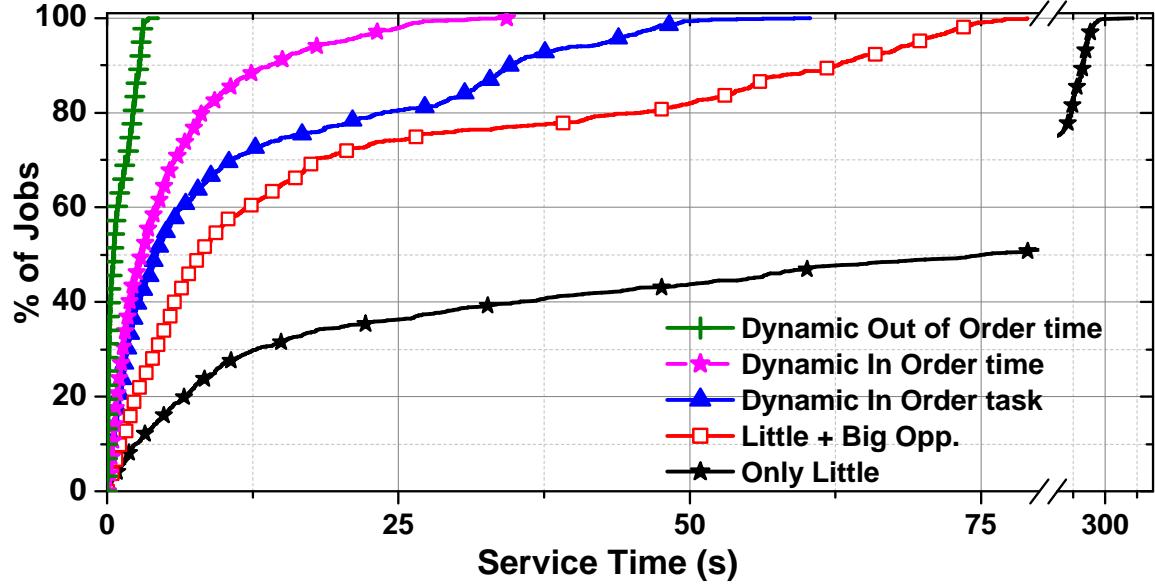


Figure 4.4: Cumulative Distribution Function Plot of the Different Configurations for Service Time.

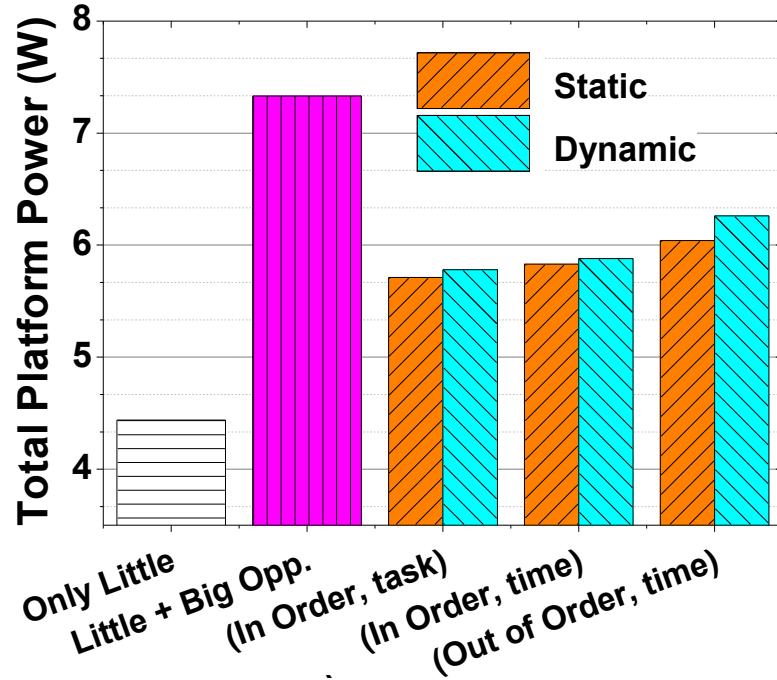


Figure 4.5: Total Platform power for the Proposed and Baseline Scheduling Polices.

(4.44W), our proposed policies increase the power consumption marginally to 6.23 W (Figure 4.5), while providing 22 \times reduction in average service time. At the other extreme, uti-

lizing little+big opportunistic consumes almost 7.33 W, yet has $4.82 \times$ lower performance than our proposed policies. That is, the proposed policies achieve both better performance and lower power than opportunistically using both clusters by utilizing the big core only when needed. Note that these results are for total platform power consumption — the trends for CPU-only power consumption are similar.

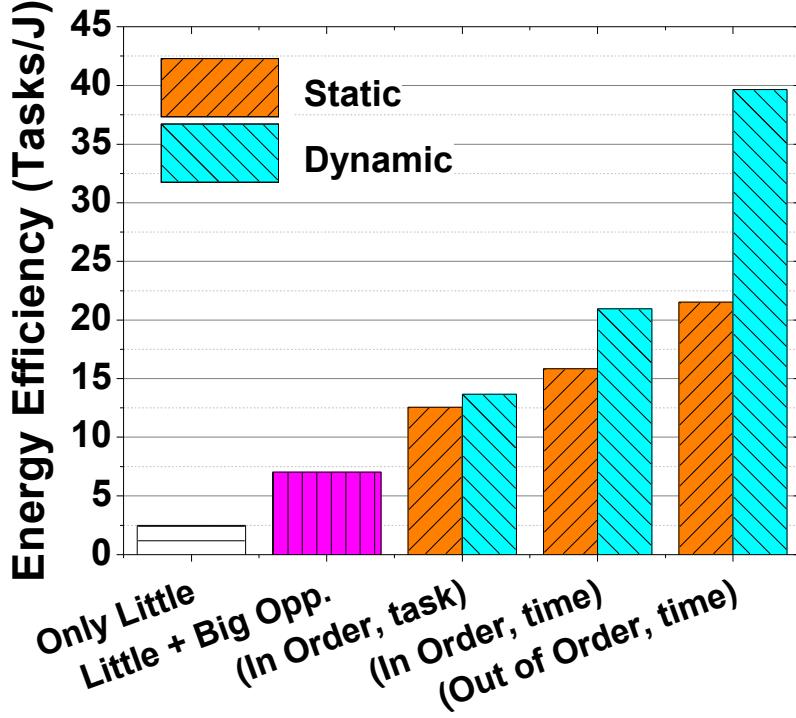


Figure 4.6: Energy Efficiency for the Proposed and Baseline Scheduling Policies.

Energy Efficiency Finally, the energy efficiency of each policy is summarized in Figure 4.6. We observed that the proposed policies achieve substantially better energy efficiency than both extremes, *i.e.*, using only the little cluster and using both clusters opportunistically. In particular, all of the static policies achieved around 15 tasks/J, which is $6.38 \times$ larger than using little cluster alone, and $2.25 \times$ larger than using both clusters opportunistically. We also noted that dynamic scheduling helps in particular with out of order scheduling and time based threshold, *i.e.*, when there are more degrees of freedom.

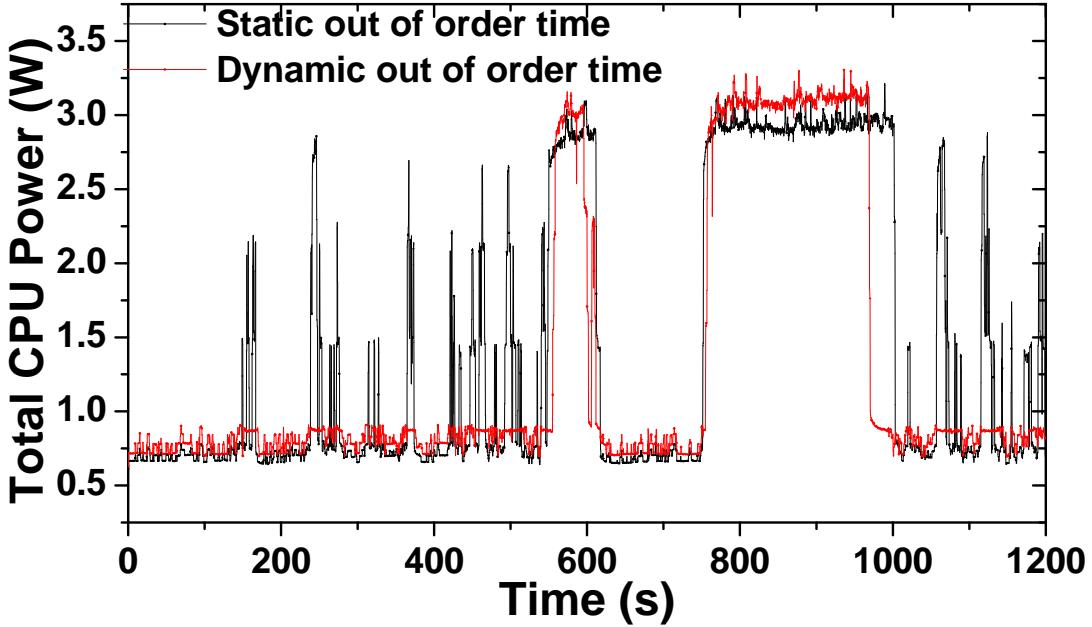


Figure 4.7: Power Profile of Static and Dynamic Scheduling Polices.

4.5.1 Impact of Dynamic Thresholds

As explained in Chapter 3, the job arrival rates change dynamically, it is beneficial to also change the threshold to adopt to the workload. Therefore, we implemented dynamic version of each algorithm and compared them with the static thresholds. Figure 4.3 shows that dynamically changing the thresholds indeed improves the average service time for all the algorithms. For instance, the optimal average service time of (*In order, task threshold*) reduces further from 18.1s to 16.4s by controlling the threshold dynamically. What is more, we achieve an impressive $4.82 \times$ reduction in the optimal average service time for (*Out of order, time*) over the baseline algorithm and $1.98 \times$ reduction in average service time over its static version, which was shown to be superior to other algorithms. More importantly, we can see a reduction of total power consumed by using dynamic thresholds as for (*Out of order, time*) a improvement of $1.45 \times$ is observed for the dynamic run-time behavior over its static version. This results in $1.89 \times$ improvement in energy efficiency by dynamically

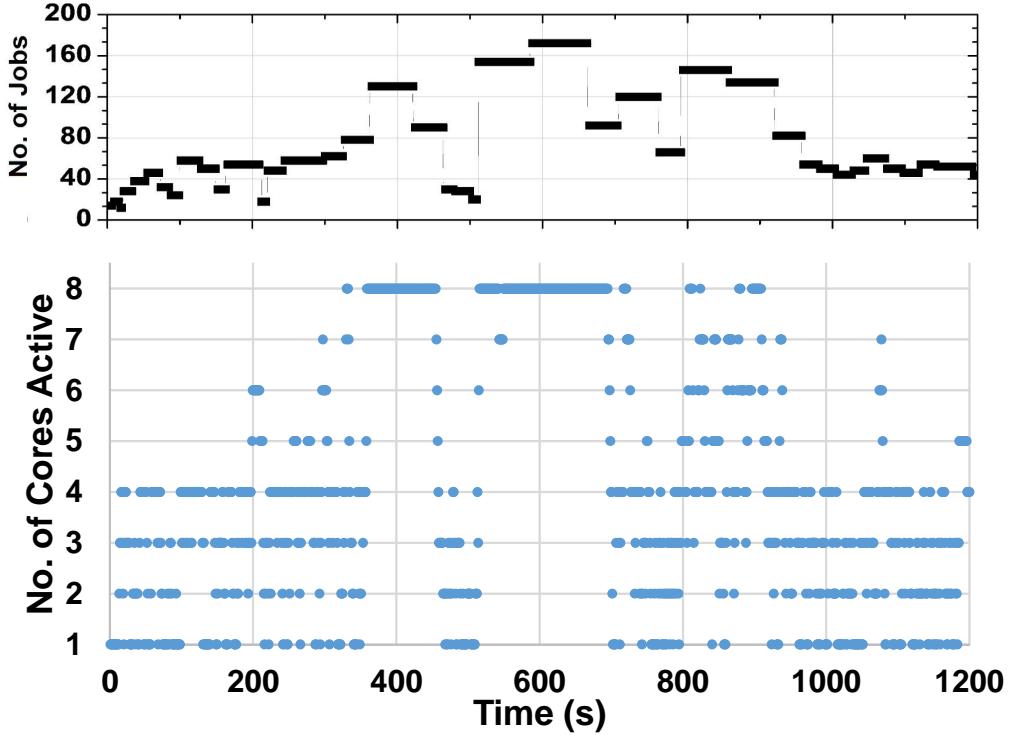


Figure 4.8: Changing the Number of Active Cores Using Pareto Graph According to the Workload

changing the thresholds and an improvement of $5.64\times$ over baseline policy for (*Out of order, time*). The advantage of dynamic thresholds can be seen clearly in Figure 4.9, as the queue occupancy for dynamic (*Out of order, time*) is the least among all the other algorithms. From Figure 4.7 it can also be observed that when using absolute time as the threshold instead of number of tasks in queue is much better as power spikes are decreased and only when needed then big cores are used. Therefore by using out of order processing and absolute time as threshold with dynamic run-time behavior we achieve maximum degrees of freedom as shown in Figure 1.3.

4.5.2 Analysis of Low-power Deadline Aware Scheduling Policy using Pareto Graph

As explained in Section 3.4.2, by using Pareto frontier graph as shown in Figure 3.4, the Deadline-Aware Out-of-Order Execution scheduling policy is modified to decrease power

consumption. As all the cores are not used throughout the run-time as shown in Figure 4.8, a increase in average service time is seen from 5.2s to 5.6s but total platform power consumption decreases from **6.26W** for Deadline-Aware Out-of-Order Execution, the most optimal scheduling policy, to **5.8W** and therefore the energy efficiency nearly remains same to 40 tasks/J. So for low-power needs Algorithm 3, should be used so that SLA is also met and make the power consumption bare minimum.

4.6 Analysis of Run-time Behavior

Figure 4.9(b)-(d) show the frequency of the big cluster, measured temperature and queue occupancy as a function of time for a 20 minute window. The corresponding rate at which search queries arrive for processing is shown in Figure 4.9(a).

We identified the optimal threshold as a function of job arrival rate empirically. Then, we analyzed the job queue occupancy as a function of time for the policies under study. Figure 4.9 shows that all policies are able to maintain a small queue utilization when the job arrival rate is small. However, running only the little cluster suffers from even small bursts in the job arrival process, as demonstrated by the peak at 400s. What is more, the queue occupancy blows as the input data becomes more intense.

In agreement with Figure 1.2, all of the proposed policies maintained a smaller queue occupancy. Note that the baseline policy in which both big and little cores are used opportunistically has the highest peak temperature and several short temperature spikes, because of the frequent utilization of three or even all four big cores, as seen in Figure 4.10. In response, the power governor scales the frequency of the big cluster from 2.2GHz to as low as 1.8GHz. In contrast, the proposed policies run cooler with 10°C-30°C lower peak temperature, since they keep the big cores dark more often (see Figure 4.10). This allows the big cluster continues to run at peak frequency throughout as shown in Figure 4.9(b) , since there are no temperature spikes. Finally, the baseline policy that uses only little cores

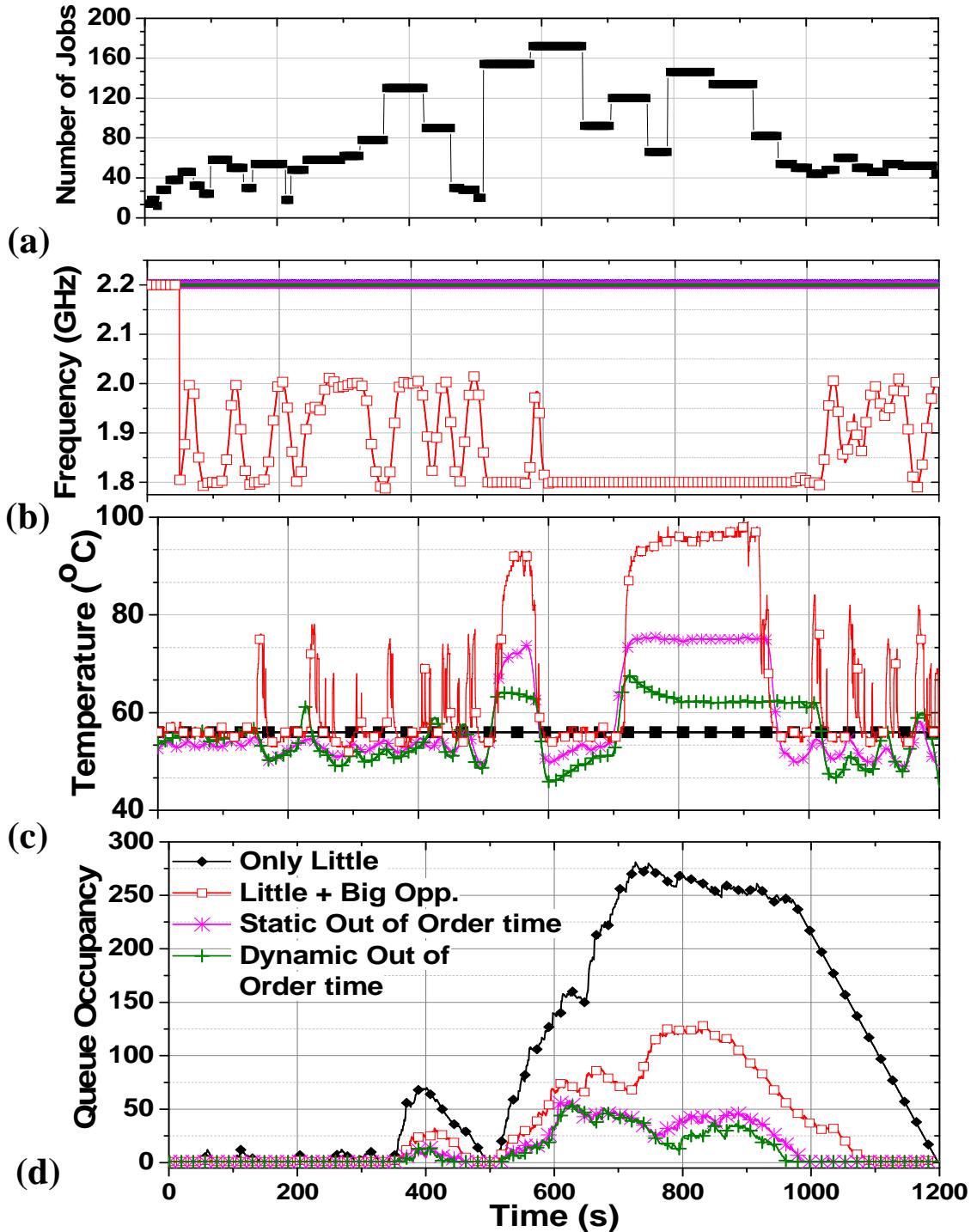


Figure 4.9: (a): Number of Jobs per 42s intervals for workload under consideration (b): Frequency of Big core (A15) (c): Temperature of the Big core (A15) (d): Queue Occupancy for the different algorithms.

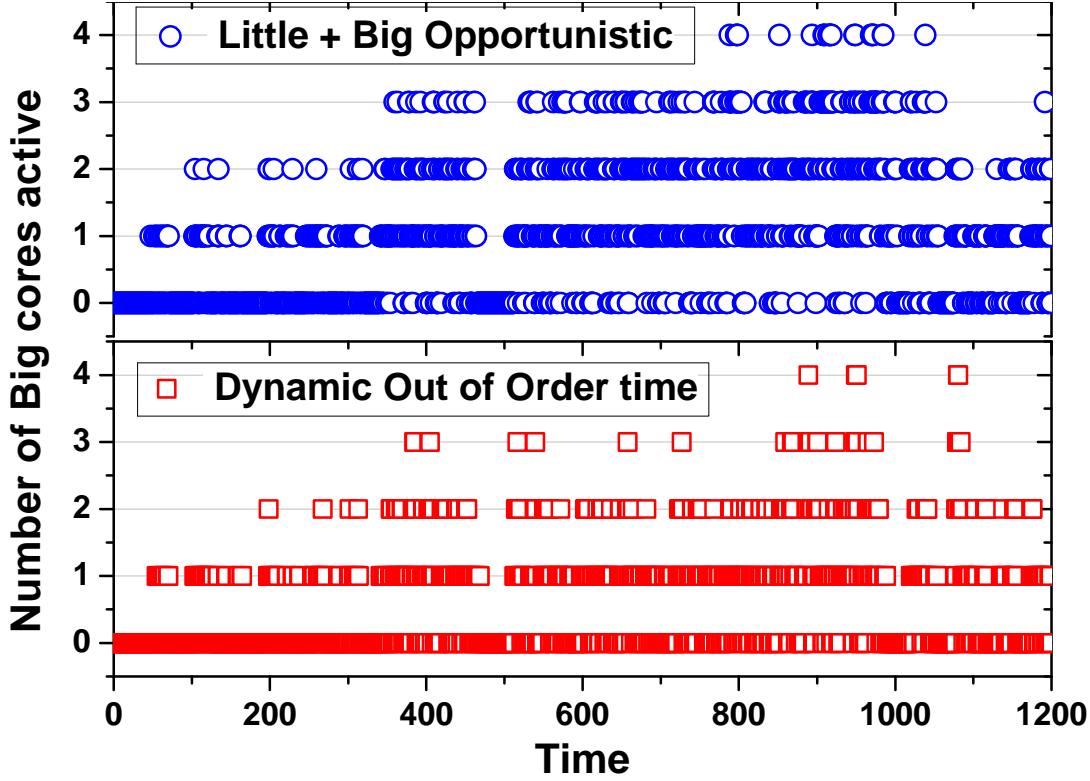


Figure 4.10: Number of Big Cores Active for "Little + Big Opportunistic" and Dynamic Out of Order Time Algorithms.

has the lowest peak temperature as expected. However, this comes at the expense of more than an order-of-magnitude performance loss, which can be explained by the large queue sizes that are observed when only little cores are used (see Figure 4.9(d)). In particular, dynamic out of order scheduling with threshold in absolute time achieved the best results, just like in performance. By utilizing the big cores very efficiently, it can maintain the queue utilization below 40 jobs even during the most intense period (800s-1200s).

4.7 Summary of Improvements

Before concluding the paper, we summarize the improvement over the baseline policy. We first note that out of order processing results in both better performance and but increase

in total power consumption. Likewise, expressing the threshold in time rather than number of tasks provides better performance and energy. Overall, (Out of order,time) scheduling policy gives the best performance and energy efficiency. Finally, we observe that dynamic threshold is superior to static threshold across all algorithms as expected. By using only static thresholds the average service time is reduced by $2.52\times$ compared to the baseline, whereas dynamic threshold policy delivers as much as $4.8\times$ improvement over baseline policy.

Table 4.3: Summary of the Improvements for the Different Scheduling Policies

Scheduling Policies	Static	Dynamic
In order,task	1.4x	1.53x
In order,time	1.8x	2.4x
Out of order,time	2.5x	4.8x

Chapter 5

CONCLUSION

Asymmetric processors that integrate cores with varying power and performance operating points are becoming increasingly popular in the dark silicon era. Based on a state-of-the-art asymmetric multi-core processor with four big and four little cores, this paper The underlying idea in combining cores with different power/performance profiles is to use the small cores for lightweight tasks, and the big cores for computationally demanding tasks. My thesis experimentally demonstrates that naive scheduling policies based solely on performance lead to thermal violations, which eventually leads to either system shut-down or frequency throttling. We address this problem by presenting a *family* of threshold-type scheduling policies for big-LITTLE servers that judiciously manage the activation of big cores to jointly optimize power and energy efficiency, can run both types of cores simultaneously using heterogeneous multi-processing. Our policies provided $4.8\times$ improvement in performance and $5.6\times$ reduction in energy-efficiency, while keeping the peak temperature 30°C lower than naive solutions.

REFERENCES

- [1] ODROID – XU3. <http://www.hardkernel.com/main/main.php>.
- [2] H. A. Abba, S. N. M. Shah, N. B. Zakaria, and A. J. Pal. Deadline based performance evaluation of job scheduling algorithms. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2012 International Conference on*, pages 106–110. IEEE, 2012.
- [3] Apache Software Foundation. Lucene Search Engine. <http://www.ibm.com/developerworks/library/os-apache-lucenesearch/>.
- [4] Apache Software Foundation. Websites Powered by Lucene Search Engine. <http://wiki.apache.org/lucene-java/PoweredBy>.
- [5] ARM. big.LITTLE Processing. <http://www.arm.com/products/processors/technologies/biglittleprocessing.php>.
- [6] N. M. Asghari, M. Mandjes, and A. Walid. Energy-efficient scheduling in multi-core servers. *Computer Networks*, 59:33–43, 2014.
- [7] T. P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 120–120. IEEE Computer Society, 2003.
- [8] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proc. of Conference on Computing Frontiers*, pages 29–40, 2006.
- [9] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner. Platform 2015: Intel processor and platform evolution for the next decade. *Technology*, page 1, 2005.
- [10] E. Bragg, M. Guevara, and B. C. Lee. Understanding query complexity and its implications for energy-efficient web search. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 401–401. IEEE Press, 2013.
- [11] T. D. Braun, H. Siegal, N. Beck, L. L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, et al. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, pages 15–29. IEEE, 1999.
- [12] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 390–399. IEEE, 2011.
- [13] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proc. of ASPLOS, 2013*, pages 77–88, 2013.

- [14] G. L. Djordjević and M. B. Tošić. A heuristic for scheduling task graphs with communication delays onto multiprocessors. *Parallel Computing*, 22(9):1197–1214, 1996.
- [15] R. Eliasi, T. Elperin, and A. Bar-Cohen. Monte carlo thermal optimization of populated printed circuit board. *Components, Hybrids, and Manufacturing Technology, IEEE Transactions on*, 13(4):953–960, 1990.
- [16] H. Esmaeilzadeh et al. Dark silicon and the end of multicore scaling. In *Proc. of Int. Symp. of Computer Architecture*, pages 986–994. IEEE, 2011.
- [17] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.
- [18] M. Guevara et al. Navigating heterogeneous processors with market mechanisms. In *Proc. of High Performance Computer Architecture*, pages 95–106, 2013.
- [19] V. Gupta and R. Nathuji. Analyzing performance asymmetric multicore processors for latency sensitive datacenter applications. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8. USENIX Association, 2010.
- [20] T. Heath, B. Diniz, E. V. Carrera, W. Meira Jr, and R. Bianchini. Energy conservation in heterogeneous server clusters. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 186–195. ACM, 2005.
- [21] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM (JACM)*, 24(2):280–289, 1977.
- [22] V. Janapa Reddi et al. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *ACM SIGARCH Computer Arch. News*, pages 314–325, 2010.
- [23] R. JeminaPriyadarsini and L. Arockiam. Performance evaluation of min-min and max-min algorithms for job scheduling in federated cloud. *International Journal of Computer Applications*, 99(18):47–54, 2014.
- [24] S. Jin, G. Schiavone, and D. Turgut. A performance study of multiprocessor task scheduling algorithms. *The Journal of Supercomputing*, 43(1):77–97, 2008.
- [25] G. Koole. A simple proof of the optimality of a threshold policy in a two-server queueing system. *Syst. & Cont. Letters*, 26(5):301–303, 1995.
- [26] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems*, pages 125–138. ACM, 2010.
- [27] R. Kumar and F. et. al. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proc. of Int. Symp. on MICRO*, pages 81–92, 2003.
- [28] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

- [29] Y.-K. Kwok and I. Ahmad. On multiprocessor task scheduling using efficient state space search approaches. *Journal of Parallel and Distributed Computing*, 65(12):1515–1532, 2005.
- [30] H. Li and G. Tang. Pareto-based optimal scheduling on cloud resource. In *High Performance Networking, Computing, and Communication Systems*, pages 335–341. Springer, 2011.
- [31] W. Lin and P. Kumar. Optimal control of a queueing system with two heterogeneous servers. *IEEE Trans. on Autom. Control*, 29(8):696–703, 1984.
- [32] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.
- [33] T. Muthukaruppan et al. Hierarchical power management for asymmetric multi-core in dark silicon era. In *Proc. of DAC*, 2013.
- [34] V. Pallipadi and A. Starikovskiy. The Ondemand Governor. In *Proc. of the Linux Symp.*, volume 2, pages 215–230, 2006.
- [35] A. Peter Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7, 2011.
- [36] M. Pricopi et al. Power-performance modeling on asymmetric multi-cores. In *Proc. of CASES*, pages 1–10, 2013.
- [37] B. Raghunathan and S. Garg. Job arrival rate aware scheduling for asymmetric multi-core servers in the dark silicon era. In *Proc. of CODES*, 2014.
- [38] Rao, Anil. System Overview for the SM15000 Family. http://www.seamicro.com/sites/default/files/SM_TO02_64_v2%205.pdf.
- [39] D. Sheleporov et al. Hass: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review*, 43(2):66–75, 2009.
- [40] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 205–218. ACM, 2010.
- [41] I. Viniotis and A. Ephremides. Extension of the optimality of the threshold policy in heterogeneous multiserver queueing systems. *IEEE Trans. on Autom. Control*, 33(1):104–109, 1988.
- [42] J. Walrand. A note on optimal control of a queuing system with two heterogeneous servers. *Systems & control letters*, 4(3):131–134, 1984.
- [43] L. Wenjing and W. Lisheng. Energy-considered scheduling algorithm based on heterogeneous multi-core processor. In *Mechatronic Science, Electric Engineering and Computer (MEC), 2011 International Conference on*, pages 1151–1154. IEEE, 2011.

- [44] G. Wu, Z. Xu, Q. Xia, J. Ren, and F. Xia. Task allocation and migration algorithm for temperature-constrained real-time multi-core systems. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 189–196. IEEE Computer Society, 2010.
- [45] Y. Xie and W.-L. Hung. Temperature-aware task allocation and scheduling for embedded multiprocessor systems-on-chip (mpsoc) design. *Journal of VLSI signal processing systems for signal, image and video technology*, 45(3):177–189, 2006.
- [46] J. Zhao and H. Qiu. Genetic algorithm and ant colony algorithm based energy-efficient task scheduling. In *Information Science and Technology (ICIST), 2013 International Conference on*, pages 946–950. IEEE, 2013.
- [47] Y. Zhu and V. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *Proc. of Int. Symp. on HPCA*, pages 13–24, 2013.