Universidade Federal da Bahia
Instituto de Matemática

Programa de Pós-Graduação em Ciência da Computação

PROFILE-GUIDED FREQUENCY SCALING FOR SEARCH WORKLOADS

Daniel Araújo de Medeiros

DISSERTAÇÃO DE MESTRADO

Salvador
12 de agosto de 2020

DANIEL ARAÚJO DE MEDEIROS

PROFILE-GUIDED FREQUENCY SCALING FOR SEARCH WORKLOADS

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Vinicius Tavares Petrucci

Salvador
12 de agosto de 2020

Sistema de Bibliotecas - UFBA

DE MEDEIROS, Daniel Araújo.
Profile-guided frequency scaling for search workloads / Daniel Araújo de Medeiros – Salvador, 2020.
42p.: il.

Orientador: Prof. Dr. Vinicius Tavares Petrucci.
Dissertação (Mestrado) – Universidade Federal da Bahia, Instituto de Matemática, 2020.

1. Escalonamento da tarefas. 2. Processadores de Frequência Variável. 3. Latência de Cauda.. I. PETRUCCI, VINÍCIUS TAVARES. II. Universidade Federal da Bahia. Instituto de Matemática. III Título.

"Profile-guided frequency scaling for search workloads"

Daniel Araújo de Medeiros

Dissertação apresentada ao Colegiado do Programa de Pós-Graduação em Ciência da Computação na Universidade Federal da Bahia, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

Banca Examinadora

DocuSigned by:
Vinicius Petrucci
03537B5F1C89452...

Prof. Dr. Vinicius Tavares Petrucci (Orientador-UFBA)

DocuSigned by:
Daniel Mosse
904259C1B759485...

Prof. Dr.Daniel Mossé (Pittsburgh-PA)

DocuSigned by:
[illegible]
26F6B9686F09424...

Prof. Dr.George Marconi de Araújo Lima (UFBA)

Ad honorem.

ACKNOWLEDGEMENTS

I wish to show my initial gratitude to the thesis committee (Dr. Daniel Mossé and Dr. George Lima) for being present since the proposal submission and for being able to evaluate this thesis and, in particular, to Professor Dr. Vinicius Petrucci, who guided me all the way in these two and a half years.

In addition, I also wish to acknowledge all the support I got from Mr. Denilson Amorim, who helped me with *a lot* of Java code, Mr. Tiago Matos for the eventual brainstorms we had and Dr. Rajiv Nishtala for the great amount of feedback throughout the years.

This project was initially supported by the Barcelona Supercomputing Center, through professors Dr. Xavier Martorell and Dr. Paul Carpenter, which kindly provided permission of use of the JUNO board. Later on, the core computational infrastructure was provided by the Chameleon testbed, supported by the National Science Foundation, United States of America.

"It's the questions we can't answer that teach us the most. They teach us how to think. If you give a man an answer, all he gains is a little fact. But give him a question and he'll look for his own answers."

—PATRICK ROTHFUSS (The Wise Man's Fear)

RESUMO

O escalonamento de frequências é uma técnica essencial para maximizar a eficiência dos recursos computacionais existentes, especialmente em sistemas com arquiteturas capazes de executar tais tarefas em núcleos de frequência variável. Um dos fatores mais críticos capazes de afetar a experiência do usuário em serviços como busca ou redes sociais é a latência de cauda, definida como sendo a latência no 95 ou 99-percentil. Esta latência pode ser fortemente influenciada pela resposta mais lenta de um núcleo de baixo desempenho de um sistema, com seu impacto largamente amplificado quão mais núcleos de baixo desempenho estejam hospedados neste mesmo sistema. Do lado corporativo, reduzir a latência de cauda para a meta desejada é tão necessário quanto a diminuição do gasto energético, haja vista que este é diretamente proporcional aos custos financeiros para a operação do serviço e consequentemente ao lucro. Trabalhos anteriores em escalonamento de frequências para núcleos com capacidade de variação de frequência são de granulometria grossa, no sentido que se observa o estado de toda a aplicação para a tomada de decisões (sem distinção entre aplicação, *threads* ou funções). Tais trabalhos também muitas vezes dependem de um processo externo rodando em paralelo para coletar o comportamento dinâmico de determinada tarefa.

Neste trabalho, propomos uma abordagem de escalonamento de frequências finamente granulada baseada na informação acerca da intensividade de processamento de funções no processamento de tarefas, bem como o comportamento das diferentes frequências. Como prova de conceito, avaliamos nossa abordagem em uma aplicação de busca. Nossa abordagem automaticamente permite o aumento da frequência de operação da *thread* localizada em um determinado núcleo ao chegar em uma função definida como intensiva, aumentando ainda mais após determinado tempo transcorrido, e a diminuição desta ao final da execução da função. Nós implementamos e avaliamos tal metodologia em um sistema multinúcleos real (Intel Skylake) e mostrando que o gasto energético pode ser reduzido em até 28%, em relação ao escalonador padrão do Linux, a depender da carga no serviço, ao mesmo tempo que atinge um tempo de resposta similar.

Palavras-chave: Escalonamento dinâmico de frequência. Processadores de Frequência Variável. Latência de cauda.

ABSTRACT

Dynamic frequency scaling is a technique to reduce the power consumption in computer systems. However, this technique poses challenges when adopted in latency-critical applications. One critical factor that affects user experience is the tail latency, defined as the application response time at the 95th or 99th percentile. Tail latency is heavily determined by the slowest response from a poor-performing processor core, and its impact is greatly amplified at scale as more slower cores are hosted in the same system. Meeting the desired tail latency is as important as decreasing the cost of energy consumption, since the latter are intertwined to financial costs and, subsequently, profit in data centers running cloud workloads. Prior work on dynamic frequency scaling is application agnostic and coarse-granulated in the sense that it considers the entire application process utilization for decision making (without the distinction between individual threads or functions).

In this work, we propose a finer-grained dynamic frequency scheduling approach that leverages information about the computational intensity of certain functions in a latency-critical web search application. First, our approach collects a profile from the running application to identify hot functions for typical workloads. Next, a runtime scheme is devised to adapt the individual core frequency whenever an allocated thread to that core reaches the entry point of a hot function, while decreasing the frequency after the thread reaches the exit point of the hot function. We implemented and evaluated our proposal in a real multicore system (Intel Skylake). We observed energy consumption savings up to 28% when compared to Linux's Ondemand frequency-scaling governor, while attaining acceptable levels of tail latency constraints.

Keywords: Frequency scaling. Frequency-Scaling Processors. Tail latency.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOTATIONS

ACPI	Advanced Configuration and Power Interface	24
DVFS	Dynamic-Voltage Frequency-Scaling	5
JNI	Java Native Interface	7
JNIF	Java Native Interface Framework	21
JVM	Java Virtual Machine	6
JVMTI	JVM Tools Interface	7
MIPJ	Millions of Instructions per Joule	39
OS	Operational System	5
QoS	Quality of Service	39
RAPL	Running Average Power Limit	24
RL	Reinforcement Learning	10
URL	Uniform Resource Locator	7
VSM	Vector Space Model	8

Chapter 1

INTRODUCTION

As modern applications are moving from desktop clients to smaller mobile devices, the processing and storage needs now rely on powerful servers in the cloud (Barroso, Clidaras and Hölzle 2013) once locally present. Examples of such services include web search, video streaming, file sharing, and word processing applications.

In data centers, delivering a satisfactory metric of Quality of Service (QoS) is critical for cloud-hosted services because service delays may affect user experience and impact companies' revenue negatively. A study revealed that a delay of 2000 milliseconds in returning web search results may impact revenue in over 4% per user (Schurman and Brutlag 2009). For large cloud companies, this turns out to be a strong negative impact for their business. At scale, the user-perceived QoS is usually determined by the slowest servers' response — the tail latency, typically the bottom-1% (slowest) distribution of the service's response time (Dean and Barroso 2013).

The implications of tail latency are that in systems where each server typically answers at 10 milliseconds, but with a 99-percent latency of 1 second, every 1 in 100 requests will be affected if the request is handled by a single server. Scaling to 100 servers, 63% of requests may take more than 1 second to respond(Dean and Barroso 2013). While it is possible to guarantee a high quality of service through the sole acquisition of better hardware, this is very costly and may be economically infeasible as the application scales. Thus, several techniques, both in hardware (e.g., out-of-order execution, branch prediction (Yeh, Marr and Patt 1993)) and software-side (e.g., canary requests (Dean and Barroso 2013)), have been developed and deployed in modern cloud systems.

Some modern multi-core systems are designed to explore a technique named Dynamic-Voltage Frequency-Scaling (DVFS), in which the frequency of operation of a certain core may increase or decrease according to the application's demand. Roots of the original DVFS idea can be traced as early as 1994 (Weiser et al. 1994). In that seminal work, Weiser et al. argued that reducing only the clock speed does not reduce the amount of Millions of Instructions per Joule (MIPJ), once the system must run for longer to perform equally. However, at the same time, the authors conclude that by adjusting the clock speed and voltage at a fine-grain, substantial CPU energy can be saved with a limited impact on performance.

In practical settings, DVFS technique is implemented within the Linux's ondemand governor (Pallipadi and Starikovskiy 2006), which attempts to minimize energy consumption by changing the CPU speed according to the actual load, which is based on CPU utilization thresholds. Traditional DVFS techniques do not take in account the application behavior; also, they are unaware of the tail latency constraints.

Alternative scheduling approaches have been developed in the literature exploring techniques like statistical prediction (Kasture et al. 2015), feedback-control state machines (Petrucci et al. 2015), and reinforcement learning (Nishtala et al. 2017) for heterogeneous multicore systems. These works are coarse-grained in the sense that it maps the entire workload onto the heterogeneous cores, and usually require external run-time systems that may incur some overhead in the application itself. They are also not designed for taking into account particular characteristics found in the code structure of the application, such as entry or exit of hot functions or the actual thread state, that could be explored in a more fine-grained frequency scaling.

1.1 THESIS STATEMENT AND CONTRIBUTIONS

This thesis aims to address the issue of energy efficiency and tail latency through the following statement:

Thesis Statement: *Adapting the CPU speed at function level in a cloud application helps to deliver requests within the imposed tail latency while reducing the overall energy consumption in comparison to frequency-scaling methods that rely only on CPU utilization metrics.*

As a corollary of the statement above, our major contributions are essentially in two axes:

- **Energy Efficiency.** We propose a new frequency scaling heuristic that maximizes the resource efficiency by decreasing the energy consumption when compared with existing methods.

- **Tail Latency.** Our scheme aims to maintain the quality-of-service within the 99-percentile according to the desired deadline.

All the developed codes are publicly available and documented at the GitHub repository ⟨https://github.com/raijenki/elastic-hurryup/⟩ under the Apache 2.0 license.

1.2 THESIS STRUCTURE

This chapter presented the motivation of our work and an overview of the problem addressed in this thesis.

Chapter 2 discusses the background of our work and reviews previous works in the literature. Since there are not related works that uses code-level granularity for scheduling, this thesis attains to discuss application-level works using DVFS processors. In

Chapter 3, we introduce our approach for improving energy efficiency with tail latency requirements and discuss the profiling and frequency-scaling framework used by our approach and its implementation. **Chapter 4** presents the results of our implementation comparing it with the Linux's Ondemand DVFS governor. **Chapter 5** concludes the work and discusses suggestions for future works.

Chapter 2

BACKGROUND & LITERATURE REVIEW

This chapter discusses the necessary knowledge to understand this thesis. We first start with introductory concepts, including power governors within Linux. We discuss some aspects of the Java Virtual Machine (JVM) and how a search application works on top of the JVM. This chapter also gives a brief overview of related works.

2.1 BASIC CONCEPTS

The Dynamic-Voltage Frequency-Scaling (DVFS) technique allows the variation of frequency within a processor. This can lead to almost a cubic drop in power consumption, and this occurs because dynamic power (P_{dyn}) is related to a voltage (V) and frequency (f) according to the equation $P_{dyn} \propto V^2 f$ (Guliani and Swift 2019). There are two widely used implementations of DVFS: the first, known as "Global Frequency and Voltage Scaling", scales the voltage and frequency for all the cores within the system simultaneously, and requires only a single voltage controller. The second is known as "Individual Frequency and Voltage Scaling", which allows the per-core changes in frequency and voltage while also increasing hardware complexity by having a per-core voltage regulator (Guliani and Swift 2019).

The Advanced Configuration and Power Interface (ACPI) specification divides the power management technologies into two categories: P-States and C-States. The former category provides a way to scale the frequency and voltage at which the processor runs in order to reduce the power consumption of the CPU. Within the Linux Kernel, the cpufreq interface makes use of the processor-specific ACPI drivers for P-States and frequency management. It includes four different in-kernel governors that allow automatic management of those P-States by the Operational System (OS). Table 2.1 illustrates them.

The highest or lowest available frequencies in Table 2.1 are always within a user-specified range. The userspace governor exports the available frequency information for a user-level interface to permit user-space control. The ondemand governor (Pallipadi and Starikovskiy 2006) measures the per-core CPU usage every sample time

Name	What it does
Powersave	Runs the core at the lowest available frequency.
Performance	Runs the core at the highest available frequency.
Ondemand	Runs the core at a frequency accordingly to CPU usage.
Userspace	Runs the core at the frequency specified by the user.

Table 2.1 In-kernel Linux Governors for the ACPI Driver

and if it's above a certain up threshold, it automatically elevates the frequency to the maximum available. If it's below a certain down threshold since the last check, the frequency scales down by 20%.

There are also C-State mechanisms for power management in the CPU to reduce or turn off some selected functions. Processors can support different numbers of C-states in which various parts of the CPU are temporarily disabled in order to reduce energy consumption. Generally, higher C-states turn off more parts of the CPU, which can significantly reduce the CPU power consumption. However, from the OS's standpoint, the processor is simply idle and can be back to a lower C-State at a very high speed. Also, processors may have deeper C-states that are not exposed to the operating system. Table 2.2 shows the C-states present for the Intel Xeon 6126 and its functions.

Mode	Name	What it does
C0	Active Mode	CPU is fully online.
C1	Halt	Interrupts the principal internal CPU clocks through software; the BUS interface unit and the interrupt controller are online and at maximum speed. Can return to C0 instantaneously.
C1E	Advanced Halt	Same as C1, but also reduces CPU's voltage.
C6	Deep Turnoff	Reduces CPU's internal voltage to any value, including 0 Volts. Saves core state before shutdown.

Table 2.2 List of C-States present in a processor Intel Xeon 6182.

2.1.1 Java Virtual Machine

The Java Virtual Machine (JVM) was introduced in 1994 under the motto of "Write once, Run anywhere". The JVM is detailed by a specification and its most known implementation is the HotSpot, done by the OpenJDK project.

In essence, the JVM is capable of either interpreting or Just-in-time compiling any Java bytecode originated from a Java compiler. The executed code lives at a sandboxed environment within the JVM, which means that there are security mechanisms to avoid application (mainly java web applets) breaking from JVM-space to kernel-space.

Threads are the basis of parallel processing and allow an application to do multiple tasks at the same time. In Linux, the JVM implements virtual threads through native threads, meaning that a Java thread will share the process identifier but also have its unique thread identifier inside the JVM, and it means that a thread may also be assigned to an individual processor core, usually through the `taskset` Linux tool.

Java provides the use of the Java Native Interface (JNI) framework for writing native code (e.g. C and C++), which also enables the usage of platform-specific features, such as POSIX's thread affinity library or a native graphics API. The JNI code is compiled and can not be just-in-time compiled as it happens with the common Java code. Objects can be shared between the JNI code and the actual Java code.

The JVM allows the class code to be executed with an agent. This agent commonly uses the Java Instrumentation API, which enables the modification of existent code, either at startup-time, before any code is executed, or dynamically. Agents can be written using the JVM Tools Interface (JVMTI), which "provides a JVM interface for the full breadth of tools that need access to JVM state, including but not limited to: profiling, debugging, monitoring, thread analysis, and coverage analysis tools"[1].

2.2 SEARCH WORKLOADS

(Brin and Page 1998) and (Coulouris et al. 2012) describe how a web search service works, in particular, Google's search engine[2]. The description below is a high-level overview of a typical search application.

A search engine is composed of three major independent applications: crawling, indexing, and querying/scoring. Crawling is related to the fetching of the contents present in a website. This is done by several automated crawlers which recursively scans a webpage and extracts all links from it. The pages are compressed and sent to a repository. Every web page has an associate ID number called docID, which is assigned whenever a new URL is parsed out of a web page.

Later on, the crawled pages are parsed and each document is converted into a set of word occurrences called hits; these are composed by the word, position in the document, font size, and capitalization.

The indexer distributes those hits into a set of "barrels", creating a partially sorted forward index. The indexer also parses out all the links in every webpage and stores the information about them in an anchor file, which is read by a Uniform Resource Locator (URL) resolver and converted into docIDs. The URL resolver puts the anchor text into the forward index, associated with the docID that the anchor points to, while generating a database of links, which are pairs of the docIDs - used by the ranking phase. The sorter takes the barrels, which are sorted by docID, and resorts them by wordID to generate the inverted index, also producing a list of wordIDs and offsets into the inverted index. An application named DumpLexicon takes this list along with the lexicon produced by the indexer and generates a new lexicon to be used by the searcher. The searcher uses the built lexicon together with the inverted index and the ranking application to answer queries.

Web search is a latency-sensitive application. For complex keywords and/or a large

[1]JVMTI Documentation: https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html
[2]Google Search Engine: https://www.google.com

number of users, the query evaluation procedure might take longer time, affecting user experience. The first Google version, as described by Brin and Page (1998) took between 1 and 10 seconds to answer any query. This is quite different for today's standard, as we expect a complex answer in hundreds of milliseconds for better user experience.

2.2.1 Lucene and Elasticsearch

Before introducing Elasticsearch, we first need to understand the engine used called Lucene. In a nutshell, Lucene is a java library that does the indexing and querying functions. It's fully open-source, cross-platform, and maintained by the Apache Software Foundation.

In Lucene, a document is the unit of searching and indexing. A document is also comprised of fields, which are key-value pairs, similar to most NoSQL databases. A Lucene index is comprised of multiple documents - hence, the indexing process described in the previous section means that a document will be inserted into this Lucene Index.

Search is the retrieval of a sorted list of documents from an index - the ones that are most relevant to the query. This sorting task is known as scoring or ranking, and Lucene uses a combination of a boolean model with the Vector Space Model (VSM) of Information Retrieval. As described by Lucene's own documentation[3], "the idea behind the VSM is the more times a query term appears in a document relative to the number of times the term appears in all the documents in the collection, the more relevant that document is to the query. It uses the Boolean model to first narrow down the documents that need to be scored based on the use of boolean logic in the Query specification. Lucene also adds some capabilities and refinements onto this model to support boolean and fuzzy searching, but it essentially remains a VSM based system at the heart".

Elasticsearch is a Java-written opensource websearch application built atop of Lucene, being mainly responsible for serving the search results through its API while also allowing Lucene to scale among clusters. This means that Elasticsearch acts as a front-end application, responsible for cluster management, thread-poll, queues. and monitoring APIs. Some clients are built to use Elasticsearch, being Elastica and elasticsearch-php the most prominent of all.

Elasticsearch encapsulates and breaks all the Lucene Indexes into Shards. For data redundancy (e.g. prevent fail-overs) and optimizing search times, there are also replica shards which are an exact copy of any shard. Multiple shards are contained into an Elasticsearch node, and usually, a replica shard will not be into the same node as the original.

As a distributed application, a node can easily be associated to a unique machine. Multiple nodes or machines comprise an Elastichsearch cluster, which is the biggest unit in the application. This cluster handles all the non-Lucene tasks, such as query distribution among shards, creation, and replication of shards and maintenance.

Whenever a query reaches an Elasticsearch cluster, it is promptly distributed among search threads. Those are exclusively for searching documents within shards and there's at least one thread per shard. Each shard will make use of the benefits of parallel

[3]Lucene Documentation: ⟨https://lucene.apache.org/core/3_5_0/scoring.html⟩

processing and act independently, and each will have its own return time. As per the tail latency problem, the overall service time is defined by the shard that took the most time to return.

Each search thread scores its own results and returns to the node, that returns to the cluster - which sorts all the obtained results and returns to the user. For optimal results, it's possible to add to the scoring factors like geolocation and synonyms.

2.3 RELATED WORK

Several approaches to this problem were already developed in the literature, exploring different concepts and techniques. Pegasus (Lo et al. 2014), Heracles (Lo et al. 2015), Rubik(Kasture et al. 2015), Hipster(Nishtala et al. 2017), Slow2Fast(Haque et al. 2017), and Parties (Chen, Christina and Martínez 2019) are the ones most closely related to our proposal - as they are all specific for DVFS systems - and will be discussed here. Table 2.3 shows an overview of all the works described below.

Name	AMP or DVFS	Technique	Processor-specific Technology?	Year
Pegasus	DVFS	Statistical Analysis	Yes (RAPL)	2014
Heracles	DVFS	CPU Monitoring / Statistical Analysis	Yes (RAPL)	2015
Rubik	DVFS	Statistical Prediction	No	2015
Hipster	Both	Reinforcement Learning	No	2017
Slow2Fast	Both	Statistical Analysis	No	2017
Parties	Both	CPU Monitoring / Statistical Analysis	No	2019

Table 2.3 A list of previous related works and their major features.

2.3.1 Pegasus, Heracles & Rubik

All three papers in this subsection are regarded for trying to adjust the frequency of DVFS processors according to the usage in order to optimize energy efficiency. The Pegasus paper introduces an energy efficiency policy named "iso-latency", which monitors task latency point-to-point and modifies the energy configurations of all servers to make them reach all deadlines as fastest as possible. For such task, a feature present in Intel-only processors - Running Average Power Limit (RAPL) - was utilized, and it allows the user to set up an energy limit where the CPU never should exceed.

Pegasus (acronym for Power and Energy Gains Automatically Saved from Underutilized Systems) is an effective implementation of the iso-latency policy, being feedback-based and being able to adjust RAPL configurations. In essence, when the data reveals that there's a lot of space for latency, Pegasus decreases the allowed energetic level; on the opposite, when latency is near to service level metric, Pegasus increases the allowed energetic level.

Heracles is more sophisticated than Pegasus. While it also implements the iso-latency policy, it provides a controller that manages both the network, the CPU power utilization, and the cores/memory performance counters. As it uses online monitoring and also offline profiling information, it is capable of identifying when shared resources become saturated and are likely to cause service time violations; hence, the controller is able to configure the appropriate isolation mechanism to proactively avoid that from happening

Rubik is another DVFS approach, but without utilizing that processor-specific feature (RAPL). The kernel of the idea is a statistical model that utilizes a dynamically collected service time distribution in order to handle the uncertainty from the computational needs of individual requests. It allows the distribution predicting which is the lowest frequency that will not violate the imposed deadline. Each time a new task arrives, a new prediction is made and frequency is changed. Since there's an usual correlation between frequency and energy consumption/service time, results can be good.

2.3.2 Hipster

Hipster utilizes a hybrid approach of Reinforcement Learning (RL), which consists of a feedback-based controller that allocates tasks dynamically into heterogeneous cores while selecting optimized DVFS parameters. The RL problem is formulated by a Markov Decision Process, in which the algorithm is rewarded by a point if it's correct and punished if wrong. Regarding the implementation, Hipster has two variants: HipsterIn, for optimized energy efficiency of latency-critical only tasks, and HipsterCo, which allows running latency-critical along with other jobs for better server resource management.

Hipster uses a Quality of Service (QoS) monitor that collects data regarding all executed task set. This data allows the mapping decision of the thread to the processing core. In order to operate Hipster, there are two phases: the learning phase and the exploitation phase. In the learning phase, there's a state machine with a feedback-control loop and the actual state identifies the core configuration: the DVFS configurations and the number/types of cores to be used. There's a predefined order based on energy efficiency.

In its exploitation phase, Hipster maps the tasks to the core based on the reward mechanism (HipsterIn or HipsterCo) and updates the lookup table. If the QoS guarantee falls below the target deadline, Hipster goes automatically back to the learning phase.

2.3.3 Adaptative Slow2Fast

The **Adaptative Slow to Fast** (AS2F) algorithm aims to explore the heterogeneity on a finer-grained manner in both DVFS and heterogeneous systems. Differently of Rubik and Pegasus, AS2F uses task progress instead of prediction alongside competition management in order to prioritize longer tasks. There are two components: offline and online. The first component has a feedback-based control that computes the threshold migration time based on the measured tail latency, the target migration time and the system load, while the online phase consists on thread mapping based on thresholds and task progress.

2.3.4 Parties

Parties can be easily compared to Heracles. In a similar fashion, the scheduler consists of a controller that monitors per-application tail latency, memory capacity and network bandwidth usage, and uses that information to determine appropriate resource allocations, and enforcing them using isolation mechanisms for Core, Memory or Disk.

The controller samples every 500 ms and based on the results, those resources are adjusted depending on each application Tail Latency slack. If one or more running applications has its QoS about to be violated, Parties will assign more resources. However, in the opposite case - if all applications are satisfying their target QoS - Parties will reduce the available resource.

Chapter 3

FREQUENCY SCALING SOLUTION

This chapter introduces the main elements of our solution. To motivate our design, we will first discuss in Section 3.1 the empirical observations found in our work. Our frequency scaling policy is described in Section 3.2.

3.1 EMPIRICAL OBSERVATIONS

Our approach works by observing that individual requests can demand distinct CPU processing time. Thus, the optimal frequency for improved energy usage can be adjusted to match the request demands, as we show in the following empirical observations. The specific software and server configurations used in our experiments are described in Section 4.1.

To understand the impact of the keyword size on the response time, we group queries with 4 keywords or less as "light queries" (or low constraint environment) and the ones ranging from 12 to 18 onward as "heavy queries" (or high constraint environment).

Observation 1. *Given multiple search requests of different key lengths, running the requests on a single CPU with the highest operating frequency will finish not later than a request that ran with a lower CPU frequency.*

This observation shows that the overall service time is influenced by the operating frequencies. We can demonstrate this by running multiple requests at different frequencies. In our case, we performed 3 runs of 5000 requests for each available frequency - there was no distinction between keyword length. The results are shown in Figure 3.1. We can see that the service time of the requests running at 2.6 GHz (the maximum available frequency of the CPU) is the fastest. In particular, the difference in service time between the frequencies of 1.0 GHz and 2.6 GHz is nearly three-fold.

Observation 2. *Given two different types of requests considered "light" (4 keywords) and "heavy" (12-19 keywords), the "heavy" requests will take longer to finish.*

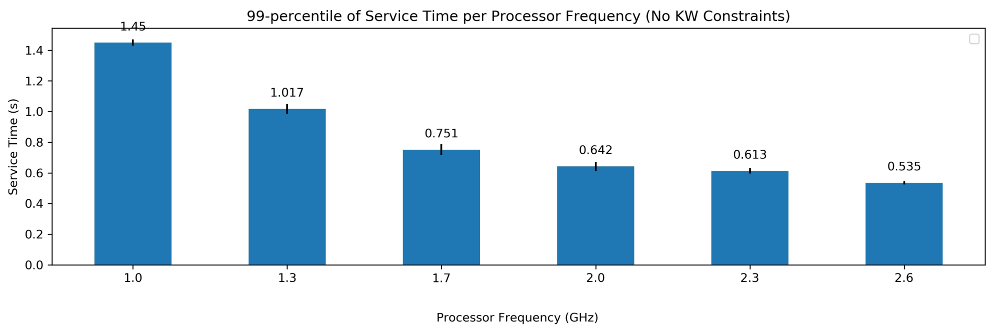

Figure 3.1 Comparison between different frequencies for mixed queries.

This experiment shows that requests with more keywords require more processing capacity than requests with fewer keywords. To show this, we fixed the keyword length considering two different frequencies: 1.0 GHz and 2.6 GHz, the minimum and maximum available in the processor. Figure 3.2 shows the results in which 2.6 GHz is always faster than 1.0 GHz for both cases, but “heavy” requests are significantly slower than the “light” ones.

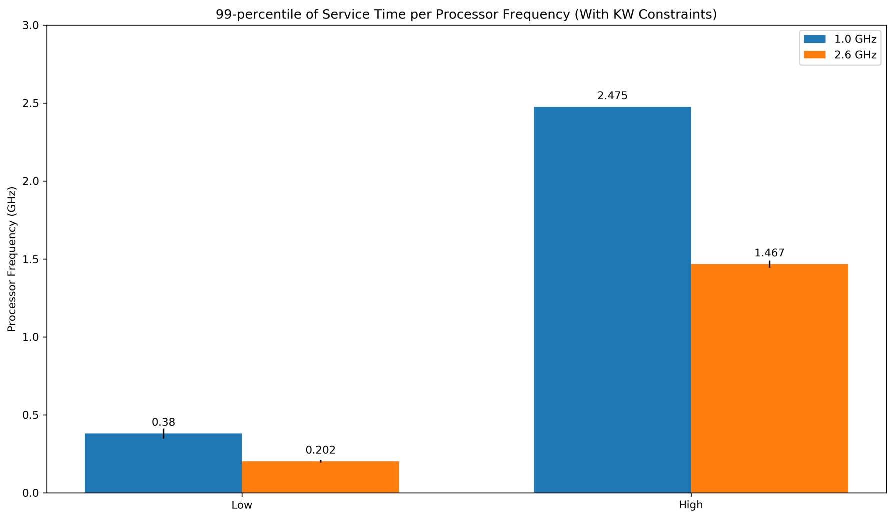

Figure 3.2 Comparison between different frequencies for different keyword length.

In Elasticsearch, each request is scored to bring the most relevant search results. Scoring is the most compute-intensive phase that involves more processing operations for requests with more keywords as the number of results to score is usually higher.

Observation 3. *Heavy requests tend to consume more energy than light requests when running at the same frequency.*

While it is a common sense that higher frequencies consume more energy, we need to check if there's any meaningful difference in energy consumption between the light and heavy requests. The results can be seen in Figure 3.3.

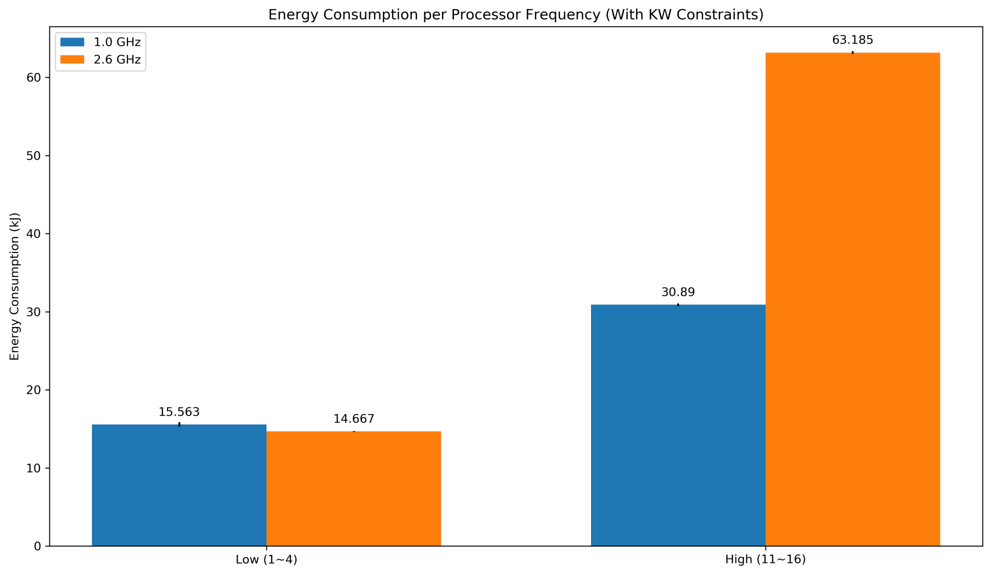

Figure 3.3 Energy consumption given processor frequency and query size.

We observe that heavy requests consume more energy than their light counterpart at the same frequency. Heavy requests makes the energy consumption over fourfold higher at the 2.6 GHz frequency. It is interesting to notice that energy consumption at 1.0 GHz is higher than 2.6 GHz. Reducing only the clock speed does not reduce the energy consumption, once doing the same work the system must run longer (Weiser et al. 1994).

Observation 4. *The best energy efficiency for light requests is not necessarily the lowest frequency.*

We show this by running only light requests while varying the available frequencies. Figure 3.4 shows that the energy consumption starts following a downward trend and later an upward one. The inflection point - at 2.0 GHz - is the most relevant here, as it has the lowest energy consumption for this case while also providing similar service time results when compared to its neighbors (1.7 and 2.3 GHz) - hence, it's 2.0 that has the best Millions of Instructions per Joule (MIPJ) here.

3.2 THE THREE STATES POLICY

From our empirical observations, we noticed that running most queries at 2.0 GHz is the most efficient way. During a query search, some queries may need to be accelerated and

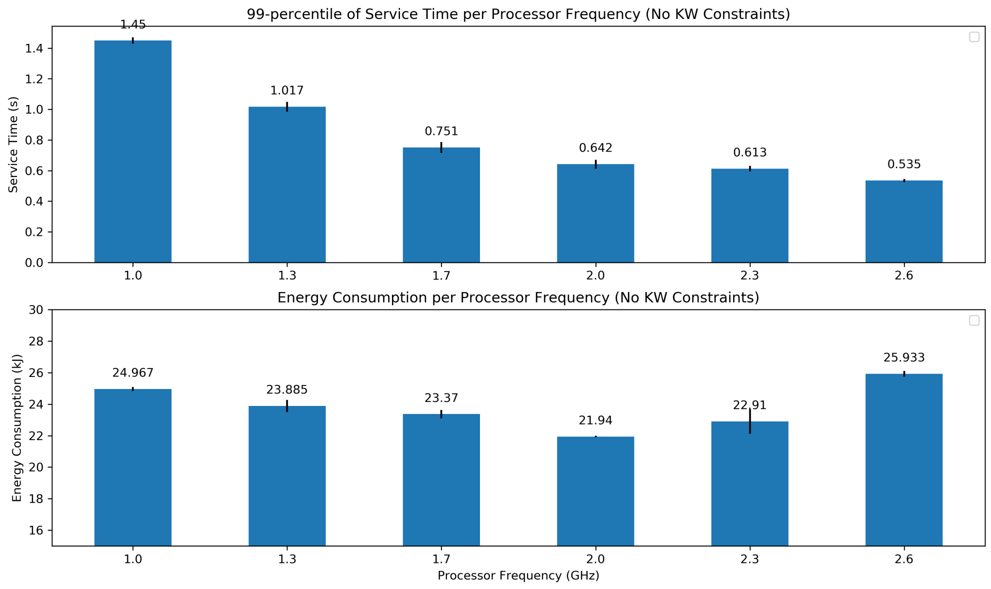

Figure 3.4 Energy consumption per processor for light requests.

others don't - because the ones that need to be accelerated would miss the deadline if at the 2.0 GHz frequency, while the latter ones wouldn't.

Hence, the basis of this algorithm involves tracking threads to check their status and take action based on this returned state. The status we are checking will answer the following: "is it executing the hot function?" and "for how long has it been at hot function?". As it will be shown below, the name "three states policy" is because the search function - which is largely responsible for the service time - also dominates the CPU usage when Elasticsearch is running, and we'll divide the application states into "idle", "low/mid intensity" and "heavy intensity".

3.2.1 Profiling Elasticsearch

To build our scheduling policy, we first need to discover the application hot function. This is done through the usage of a profiler during the application execution.

This work used two profilers: the YourKit Java Profile[1] and Linux's default tool, perf. After Wikipedia dataset was already indexed by Elasticsearch, we started running only search queries. Both profilers showed that the function:

`org.elasticsearch.search.query.QueryPhase.execute` dominated about 70% of the software execution, while having a reasonable number of calls in comparison to its later branches - which were mainly derived from the Apache Lucene engine. A call graph generated with the java-callgraph tool from YourKit is shown in Figure 3.5.

Although the hot function was chosen with some degree of subjectivity, since no

[1]YourKit Java Profiler: https://www.yourkit.com

quantitative metric for hot functions could be established, later results have shown that using this function as the choice for Three States Policy has compatible results with the baselines established in Chapter 4.

3.2.2 Frequency Scaling Algorithm

This section is an overview of how our frequency scaling works at a high-level. Implementation details can be seen at the next section below.

Our frequency scaling governor is mostly reactive to function call events. Every entry or exit to/from the hot function generates an event that is pushed into the main queue. The frequency governor sleeps for a certain time (empirical tests have shown that 50 milliseconds is a reasonable time) and, after awaking, it consumes all the events at the queue and checks it own structure for possible changes.

Each event has its own information: the thread id, core id, the clock time, and if it's entering or leaving the hot function. When consuming each event, the frequency governor modifies its own auxiliary structure in order to keep track of the thread status. This auxiliary structure exists because it's necessary to keep track of the difference between the entry time and the elapsed time - if it's bigger than a certain threshold, the thread is promoted to an increased frequency; if it's not, it's probably a small request that would reach the deadline nonetheless, and we keep it at the optimal MIPJ frequency that is set at the entry event. The checking of difference for elapsed time is done only after all the new events are consumed. If it's necessary to change frequency, either to the minimum or to maximum, the position of the core at another auxiliary ordered array is altered - either to 0, 1, or 2 - and a frequency change function is called.

This frequency change function simply checks if this array is 0, 1, or 2. 0 means that it is not on the hot function, thus decreasing the frequency. 1 means that the frequency should be increased to 2.0 GHz. 2 means that the frequency should be increased again, this time to 2.6 GHz. After any change in frequency, the function itself keeps an auxiliary structure to ensure that it doesn't change again until the frequency governor asks for it. Table 3.1 shows the composition of each structure while Algorithms 3.1 and 3.2 show the pseudo-language for execution. Uppercase denotes the full structure, while lowercase letters denote the elements of this structure. For readability purposes, we used the terms INTENSITY_LOW, INTENSITY_MID and INTENSITY_HIGH instead of 0, 1 and 2.

3.2.3 Implementation Details

Our implementation was done in a JVMTI agent in two parts. First, we implemented an event generator for capturing function entry and exit. Second, we designed and implemented the frequency governor that consumes those events. Figures 3.6 and 3.7 provides, accordingly, an overview of each part.

Whenever a request arrives at Elasticsearch, the request is split among the shards (in our case, five shards). Elasticsearch uses five threads to process that particular request across the shards. At a certain point, each thread will enter the hot function, where a signal will be sent to the JVMTI Agent (or Event Generator) whenever it reaches either the entry or exit point. After all the shards' results are scored and ranked jointly, the

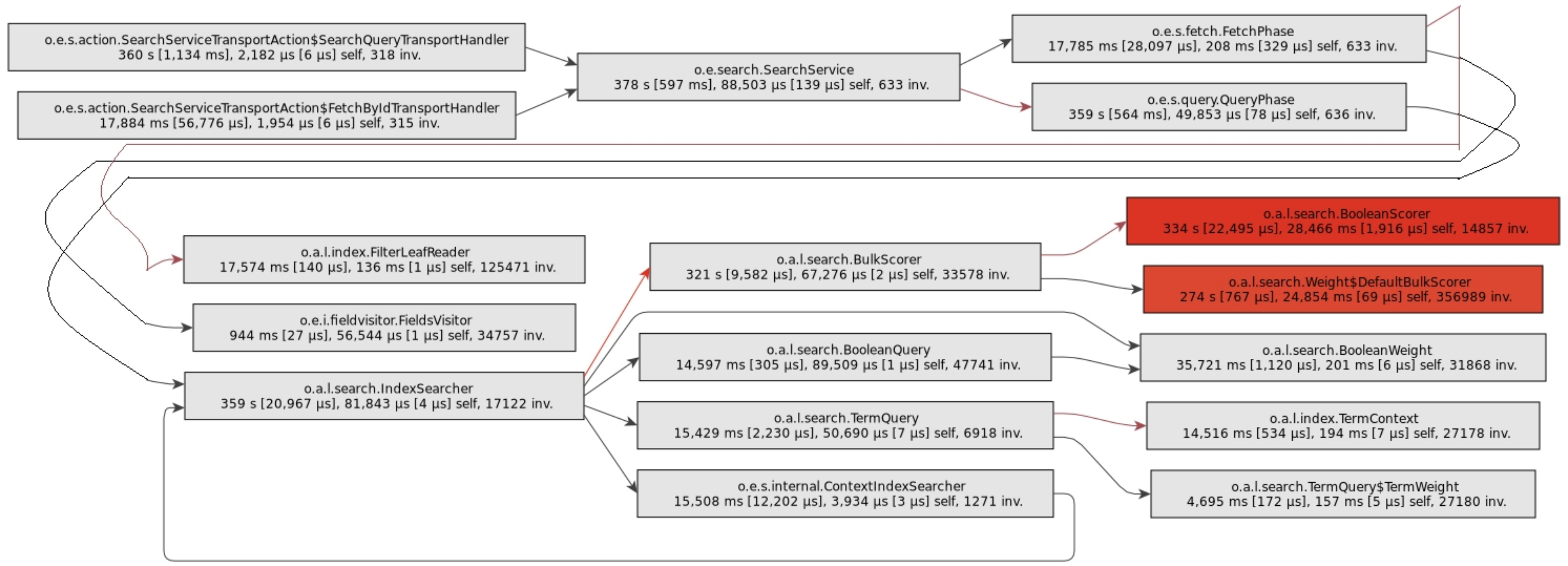
o.e.s.action.SearchServiceTransportAction$SearchQueryTransportHandler
360 s [1,134 ms], 2,182 µs [6 µs] self, 318 inv.
o.e.s.action.SearchServiceTransportAction$FetchByIdTransportHandler
17,884 ms [56,776 µs], 1,954 µs [6 µs] self, 315 inv.
o.e.search.SearchService
378 s [597 ms], 88,503 µs [139 µs] self, 633 inv.
o.e.s.fetch.FetchPhase
17,785 ms [28,097 µs], 208 ms [329 µs] self, 633 inv.
o.e.s.query.QueryPhase
359 s [564 ms], 49,853 µs [78 µs] self, 636 inv.
o.a.l.index.FilterLeafReader
17,574 ms [140 µs], 136 ms [1 µs] self, 125471 inv.
o.e.i.fieldvisitor.FieldsVisitor
944 ms [27 µs], 56,544 µs [1 µs] self, 34757 inv.
o.a.l.search.IndexSearcher
359 s [20,967 µs], 81,843 µs [4 µs] self, 17122 inv.
o.a.l.search.BulkScorer
321 s [9,582 µs], 67,276 µs [2 µs] self, 33578 inv.
o.a.l.search.BooleanQuery
14,597 ms [305 µs], 89,509 µs [1 µs] self, 47741 inv.
o.a.l.search.TermQuery
15,429 ms [2,230 µs], 50,690 µs [7 µs] self, 6918 inv.
o.e.s.internal.ContextIndexSearcher
15,508 ms [12,202 µs], 3,934 µs [3 µs] self, 1271 inv.
o.a.l.search.BooleanScorer
334 s [22,495 µs], 28,466 ms [1,916 µs] self, 14857 inv.
o.a.l.search.Weight$DefaultBulkScorer
274 s [767 µs], 24,854 ms [69 µs] self, 356989 inv.
o.a.l.search.BooleanWeight
35,721 ms [1,120 µs], 201 ms [6 µs] self, 31868 inv.
o.a.l.index.TermContext
14,516 ms [534 µs], 194 ms [7 µs] self, 27178 inv.
o.a.l.search.TermQuery$TermWeight
4,695 ms [172 µs], 157 ms [5 µs] self, 27180 inv.

Figure 3.5 Call graph used to identify Elasticsearch's hot function. We chose *org.elasticsearch.search.query.QueryPhase* based on the average time spent by a thread in this function.

Algorithm 1: Frequency Governor Algorithm

Input: EventQueue (EQ) consisting of entry/exit events; Period (p) which the scheduler sleeps after its execution and threshold for frequency change.
Data: FuncState describes the status of the hot function per core; CoreChanges is a 2-D matrix to keep track of changes made to core allocations

```
1  foreach event q in EQ do
2      if q event is already recorded in FuncState then
           /* Register the entry and the timestamp */
3          if q is function ENTRY event then
4              Set FuncState[q.coreId].status as IN_HOT_FUNCTION;
5              Set FuncState[q.coreId].timestamp as q.timestamp;
6          end
7          if q is function LEAVE event then
8              Set FuncState[q.coreId].status = NOT_IN_HOT_FUNCTION;
9          end
10     end
11     else
           /* q is not registered in FuncState. Record it for the first
              time. First event of a thread is always an entry at hot
              function. */
12         Set FuncState[q.coreId].timestamp as q.timestamp;
13         Set FuncState[q.coreId].status as IN_HOT_FUNCTION;
14     end
15 end
16 Set t as current system timestamp;
   /* No event in Queue anymore. Iterate over FuncState. */
17 foreach element s in FuncState array do
       /* s is at hot function */
18     if s.status == IN_HOT_FUNCTION then
19         set CoreChanges[s.coreId][0] as INTENSITY_MID;
20         if t - s.timestamp >= threshold then
21             set s.status as OVER_THRESHOLD;
22             set CoreChanges[s.coreId][0] as INTENSITY_HIGH;
23         end
24     end
       /* Not in hot function */
25     if s.status == NOT_IN_HOT_FUNCTION then
26         set CoreChanges[s.coreId][0] as INTENSITY_LOW;
27     end
28 end
29 Call frequency change module;
30 Sleep for p interval;
31 Repeat the algorithm from the beginning;
```

Algorithm 2: Algorithm for Frequency Change

```
Input: Bi-dimensional Matrix (CoreChanges) where the first position's index is
       the core number and the the second position indicates whether or not
       the change was persisted. The matrix's content records the frequency
       level the core is running at.
1  for every element e of CoreChanges do
       /* If the first and second elements are equal, you don't need to
          change frequencies                                              */
2      if CoreChanges[e][0] != from CoreChanges[e][1] then
3          if CoreChanges[e][0] == INTENSITY_LOW then
4              change frequency to 1.0GHz;
5          end
6          if CoreChanges[e][0] == INTENSITY_MID then
7              change frequency to 2.0GHz;
8          end
9          if CoreChanges[e][0] == INTENSITY_HIGH then
10             change frequency to 2.6GHz;
11         end
           /* Persist frequency change                                    */
12         Set CoreChanges[e][1] as CoreChanges[e][0];
13     end
14 end
```

Structure	Elements	Description
EventQueue	core_id	The physical core identifier in which the element is assigned, ranging from 0 to 23.
	timestamp	Timestamp of the entry or leave event.
	is_hotpath	Can assume two values. 1 if entry event and 0 if leave event.
FuncState	core_id	The physical core identifier in which the element is assigned, ranging from 0 to 23.
	timestamp	Timestamp of the entry or leave event.
	status	Enum. NOT_IN_HOT_FUNCTION if not in hot function, IN_HOT_FUNCTION if in hot function, and OVER_THRESHOLD if it's at hot function and above the threshold. Values outside this range are for avoiding reprocessing.
CoreChanges	Matrix	Matrix ranging from 0 to 23 on the first position's index. On the second position, there are two elements per index for comparison in order to avoid rewriting frequencies into files. E.g.: If the corechanges[0][0] is equal to INTENSITY_MID and corechanges[0][1] is INTENSITY_LOW, there'll be a frequency change as they are different - else, do nothing.

Table 3.1 Description of the structures used in our implementation

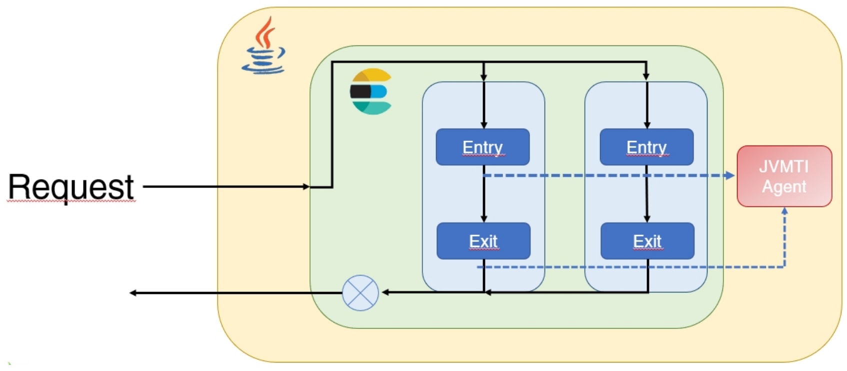

Figure 3.6 Overview of the Proposed Governor. Entry and exit moments are intercepted at execution time by the JVMTI agent.

request returns to the user.

The JVMTI Agent runs a separate process from Elasticsearch in the Java Virtual Machine. The event generator is responsible for receiving the entry or exit events signals and pushes them into a global queue. The scheduler, which is responsible for increasing or decreasing the speed of the functions, is the only point-of-contact with the cpufreq interface on Linux. Each time it is triggered, it executes the Algorithms 1 and 2 (Section 3.2.2), consuming the events located in the queue.

3.2.4 Design Options

There were two designs attempts for the event generator: a sampler and an event-based one. The idea of the former was essentially sampling all the search threads at every defined time-space and checking if they were at the designated hot function or not. If the thread was at a hot function, our heuristic would be executed to increase the running core to the maximum available frequency. For this, the JVM provides an internal function called `AsyncGetCallTrace` which allows the asynchronous collection of stack points from Java and works within a signal handler. However, our experiments showed that although the service time overhead was minimal, the `AsyncGetCallTrace` calls impacted the system's energy consumption, so we discarded this approach.

A similar situation was reported by the Java Native Interface Framework (JNIF) (Mastrangelo and Hauswirth 2014) tool for sampling. While the service time overhead was also minimal, the energy consumption was impacted as much as AsyncGetCallTrace. In the end, we opted for a mix of using JNIF and the common Java Native Interface code for the event generator. In this model, JNIF is responsible for building an entire stack of thread information. While initializing the agent, we use the `SIGBLOCK` function for threads that are not considered in "search pool". Thus, we only collect information from the threads that aren't blocked to receive signals.

The search threads, which are the responsible for running the compute-intensive hot function, have their affinities set for a single core. We mapped 12 search threads to the 12 available cores; that is, 1 search thread per core. The JNIF tool allowed us to instrument

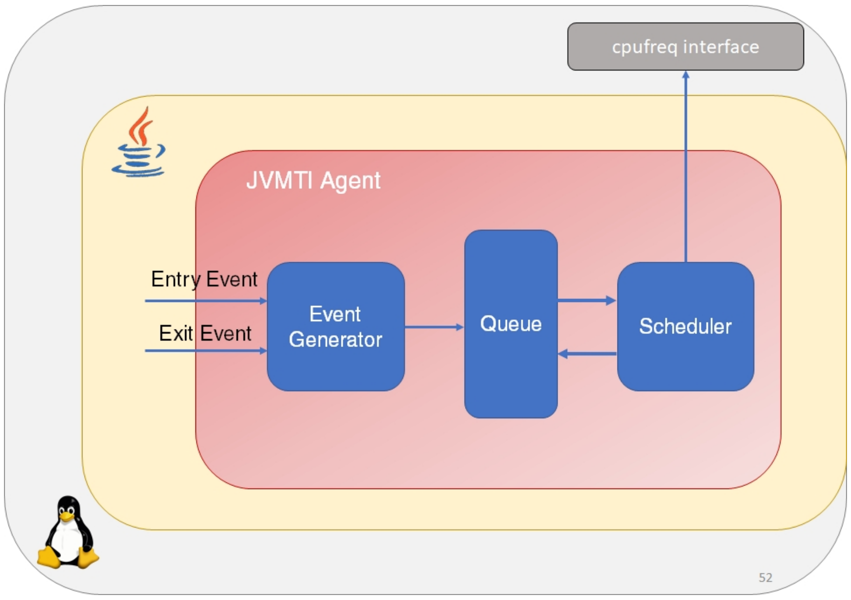

Figure 3.7 Overview of the JVMTI Agent. The Event Generator sends all the Entry/Exit Events to the Queue, which are stored there until consumed by the Scheduler - which is also responsible for the processor frequency change.

the identified hot function so as to execute our frequency scaling algorithm whenever there's a hot function enter or leave event. At each event, a line is added at a general queue with information regarding thread id, core id, and if the thread entered or left the hot function. At each designated time, the frequency governor reads this queue and takes action based on this, as seen in the previous section.

For readability purposes, our frequency governor uses a vector of structs for storing and processing the events. It uses a monotonic clock for time comparison between the arrival time and elapsed time. Frequency increases or decreases are performed through changes at the CPUFreq files, which are open when the agent initiates and are closed when the agent shutdowns. This was done to avoid the IO overhead of frequently opening/closing files.

Chapter 4

EVALUATION

This section shows and discusses the experiments that were carried out using the implementation described in Section 3.2.3. We call our implementation "Hurry-Up". We experiment with Hurry-up using the Three States Policy running on the server system. We also present sensitivity analysis in a more high-stress environment.

To establish baselines as a comparison point, we show how the Ondemand governor performs against the Powersave and the Performance governors. All response time results in this chapter refer to the 99-percentile of response time - which is essentially the tail latency of the service time. For comparison purposes, we'll establish this 99-percentile deadline as one second, and use the Ondemand in-kernel governor as the baseline. In the following figures, the imposed deadline is always shown as a black dotted line, and the discussed service time is always within the 99-percentile.

Next, we describe the hardware and software stack used in our experiments.

4.1 EXPERIMENTAL SETUP

We conducted our experiments on a "baremetal" server instance, provided by the Chameleon Cloud initiative [1] service, with the following specification:

CPU	Intel Xeon Gold 6126
RAM	196 GB
SSD	220 GB

Table 4.1 Server configuration used in our experiments.

We explore a CPU that is DVFS-capable, allowing the operating frequency to be changed on a per-core basis. The operating frequency and voltage can range between 1.0

[1]Chamelon Cloud: https://www.chameleoncloud.org, a service provided by the National Science Foundation (USA) which allows the creation of baremetal instances, and specifications like processor and network can be choosen.

and 2.6 GHz for each individual core. Moreover, the CPU has 24 physical cores equally divided between two sockets. Intel's Hyperthreading was turned off, so we only consider the physical core count. We used socket 0 for the server-side applications and socket 1 for the load generator. The energy measurement was done through Intel's Running Average Power Limit (RAPL) interface[2]. We measure the energy consumption counter at the beginning of the experiment and again at the end; the difference between those two values is considered the energy consumed for that particular experiment.

The software stack is listed below.

Ubuntu	19.10
Linux Kernel	5.3.0-51
d**Elasticsearch**	6.5.4

Table 4.2 The software stack used.

Instead of using Intel's PState driver, we used the Advanced Configuration and Power Interface (ACPI) driver since it allowed a more effective control of frequencies through its userspace governor. All the frequencies changes were done via the CPUFreq driver. To change the core frequency, we simply change the frequency values at the corresponding files the driver provided. The CPUFreq-info tool estimates that the maximum transition latency between frequencies is 10 microseconds.

Java 11 was the choice because it was the minimum recommended by Gradle[3] to compile Elasticsearch. In addition, although the most recent version of Elasticsearch is 7.3, the choice for the version 6.5.4 was because the latter one supports a plugin used for indexing the Wikipedia dump; once there are APIs changes between each major version, the plugin wouldn't work with the newest Elasticsearch version. By indexing Wikipedia, Elasticsearch has 5 shards of 8 gigabytes each.

There is a myriad of ways to configure Elasticsearch. There are several Wikipedia dumps to choose from. Figure 4.1 shows that the configuration of 5 shards of 8 GB each is better than 1 shard with 40 GB or 5 shards of 15 GB each. This is due because, at each request, Elasticsearch spans 1 thread per virtual shard. The 5x8GB gets the benefit of parallel processing over the 1x40GB, while the 5x15GB configuration is simply too big. Figure 4.1 shows a benchmark comparison.

After its initialization, Elasticsearch spans over 70 java threads. While there's a fixed formula for the number of search threads, we configured that in such a way that it creates 12 threads, one thread per core. This is because we wanted to simplify our design and to avoid possible core-sharing interference between search threads, which happens to be the most CPU-intensive when executing at the production-level.

In order to properly measure energy, the entire Elasticsearch application was initialized with the numactl[4], which made all threads with affinity to the socket 0 (even cores). After running the first query, the jstack dump reveals which threads are the search threads - and we set their affinity to an even core each.

[2]RAPL Power Meter: https://01.org/rapl-power-meter

[3]Gradle Build Tool: ⟨https://gradle.org⟩

[4]Documentation available: https://linux.die.net/man/8/numactl

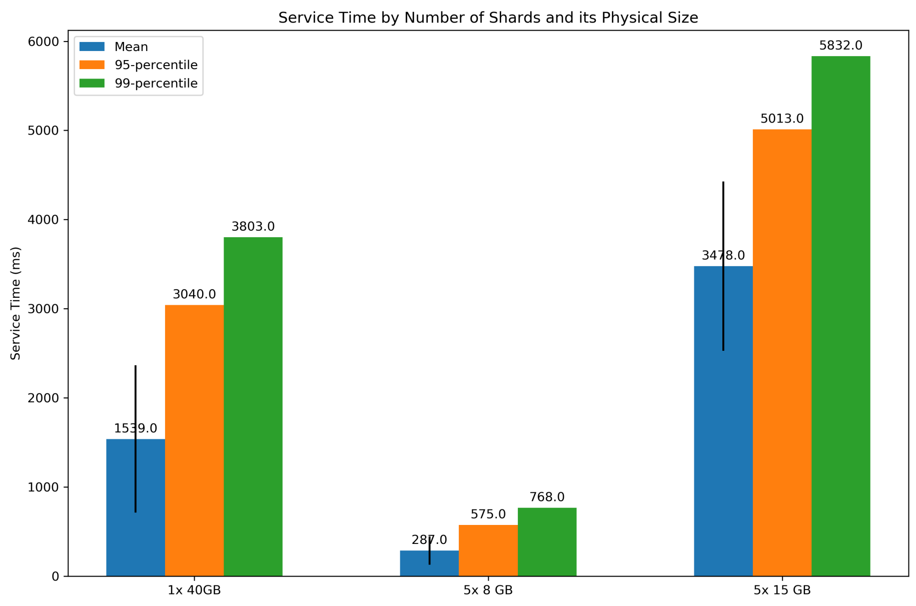

Figure 4.1 Comparison between different Elasticsearch shards configurations.

In order to mitigate network transmission issues in our experiments, we preferred hosting both server and client locally. In order to draw a boundary line between the server and the client in the same machine, the client had its entire thread affinities set to socket 1 (odd cores). The client is responsible to generate search requests with randomized keywords, which ranges between 1 to 18 keywords. The client is also responsible to measure the response time.

For the client application, we used a modified version of FABAN workload generator from Cloudsuite(Ferdman et al. 2012). The Web search benchmark from Cloudsuite was originally built for Apache SOLR, another Lucene-based search platform. So, we modified FABAN (the load generator) to issue search requests in the format specified by Elasticsearch. We use the same algorithm from Cloudsuite to generate the length of keywords, a Ziphian distribution. In practice, this means that there will be more requests of small sizes than larger sizes.

4.2 BASELINE

As explained in Chapter 2.1, the performance governor runs at the highest frequency, the powersave governor uses the lowest available frequency, and the Ondemand monitors the CPU usage in order to choose its operational frequency. A basic question here is: how do they perform against a search workload? Figure 4.2 answers this.

In a nutshell, the powersave governor doesn't meet the deadlines for the high keyword input and even at the low keyword input it is pretty much at the point of the deadline with its own standard deviation. Moreover, powersave consumes more energy than the performance governor itself at the low keyword input - the dichotomy that if you run much slower, you may end up running for more time and this might use more energy, as already pointed out by (Weiser et al. 1994). In the high keyword environment, the tradeoff between service time and energy consumption is more effective for the powersave, hence why it does not consume more energy than performance.

The performance governor performs relatively well but falls behind Ondemand on energy consumption. Since both governors are able to reach the imposed deadline, the Ondemand was chosen as the baseline for rest of this chapter because of its energy efficiency benefits.

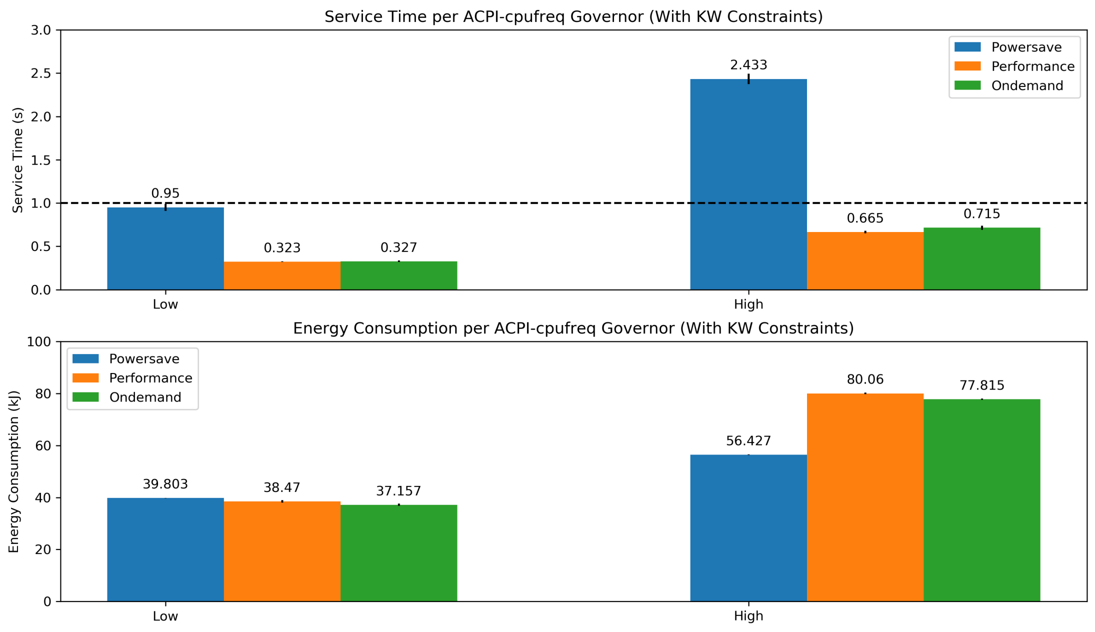

Figure 4.2 Comparison between Linux kernel governors running the search workload

4.3 BASIC EXPERIMENT

The basic experiment is a direct implementation of the algorithm described in Chapter 3. There are two parameters for our frequency governor, known as ”threshold” - when the core frequency needs to move from 2.0 GHz (mid level) to 2.6 GHz (highest level) - and the sampling time. For initial experimental purposes, we set those parameters to 350 ms and 10 ms respectively. These values came from a previous iterations of Hurry-up we did in another environment.

The load environment consisted of 20 minutes runs with low, mixed, and high keyword inputs. The FABAN workload generator was used to create 4 clients with each client issuing, on average, one request per second. The idea is to analyze the service time behavior of each governor policy considering a single request at a time. For measuring the standard deviation, we performed three runs for each keyword length and governor policy (each run consists of 20 minutes of continuous load generation). Note that the ”low” key length is fixed at 4 in order to generate minimum stress at the system, while the ”mix” key length goes between 1 to 16 (following default Faban’s Ziphian distribution). The result is shown in Figure 4.3.

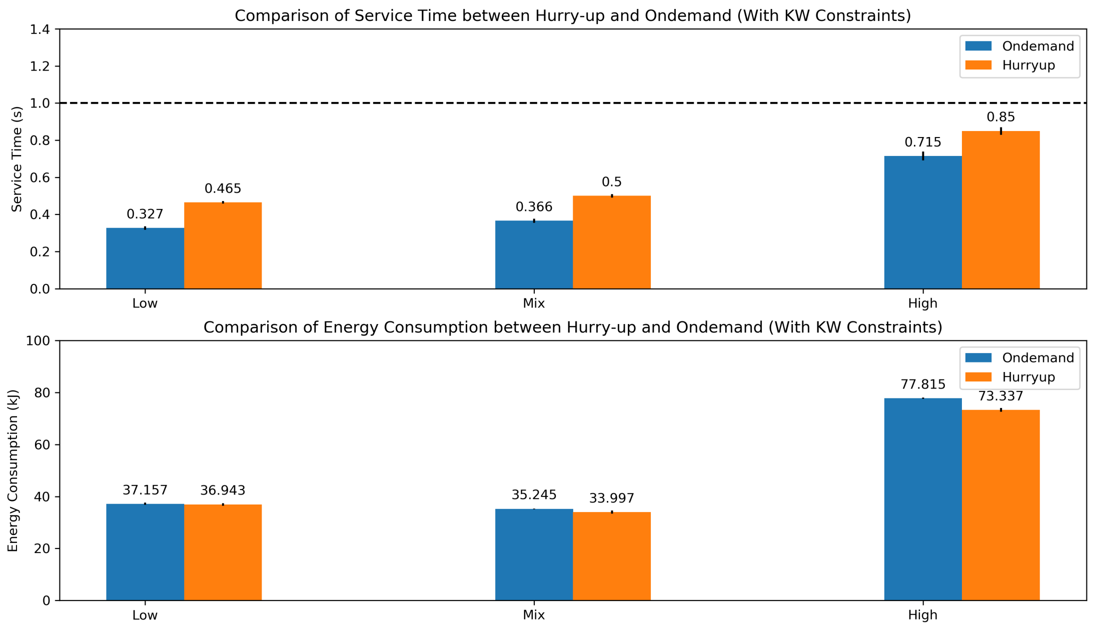

Figure 4.3 Initial implementation of Hurry-up vs the Ondemand in-kernel governor.

While both of them are able to reach the imposed deadline, Hurry-up has energy-saving that amounts up to 6% in comparison to the Ondemand governor - mainly on high-load keywords. In order to analyze why this effect happens, checking image 4.4 is necessary. First, both Ondemand and Hurry-up concentrate most of their idle time at 1.0 GHz in order to minimize energy consumption when inoperative - this happens for all keywords and has its effect accentuated at the low and mixed keywords section.

When the CPU load is too high, either given a request with a large keyword length or with a higher CPU need, the Ondemand governor starts concentrating its core frequencies at 2.6 GHz, which is not optimal in most of the cases as not all requests would require this speed. The longer it runs or the higher the load, the difference between Hurry-up and Ondemand becomes more prominent, as it will be seen later in this chapter.

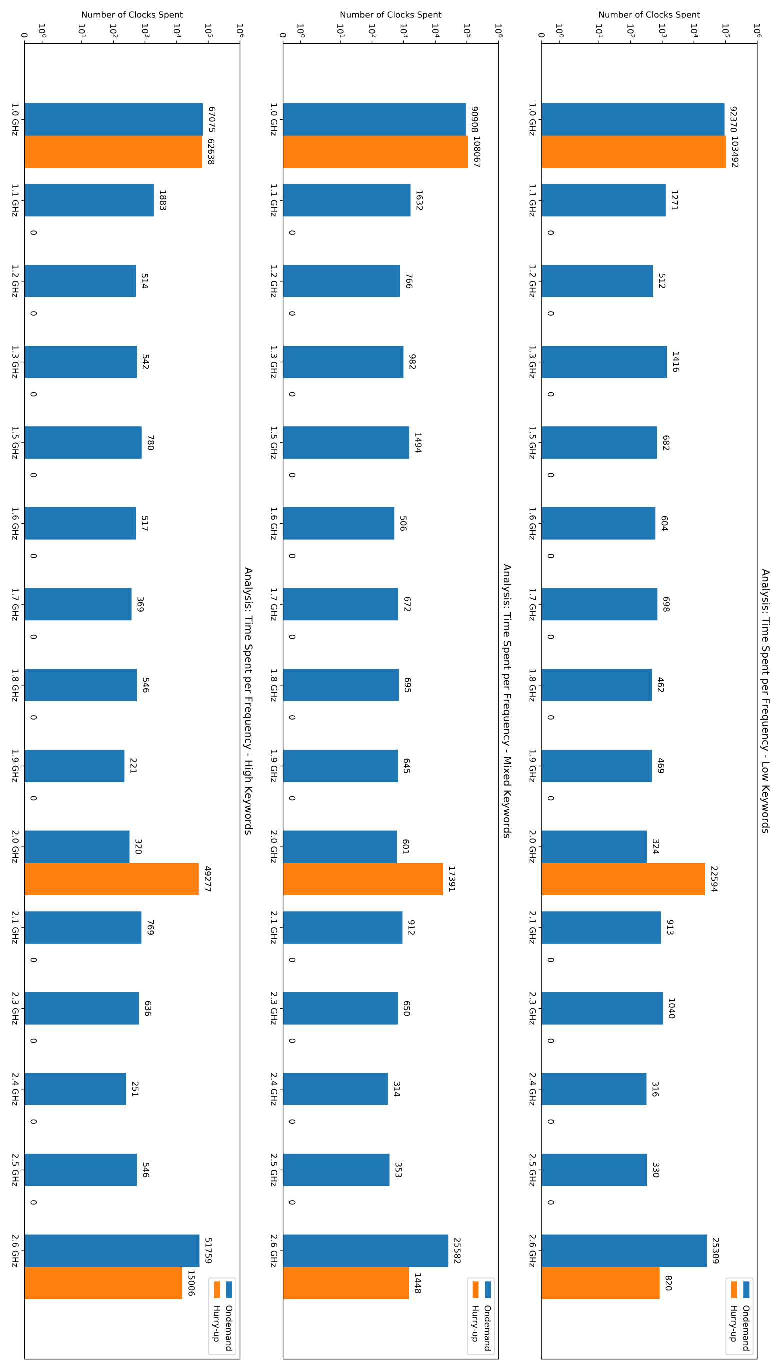

Figure 4.4 Comparison of frequency steps between Hurry-up and the Ondemand governor.

4.4 TWO STATES VS THREE STATES POLICY

A first comparison point with the proposed "Three States Policy" would be comparing against a "Two States" one, where the frequency simply goes up (to highest level) whenever the thread access the hot function, and there's not a threshold.

We evaluated the Two States policy using 2.0 GHz and 2.6 GHz as maximum available frequency. We ran each policy for 10 minutes against low and high key length inputs. The results can be seen in Figure 4.5. It is worth noticing that the standard deviation, although pretty small, is present in the image.

Although the Two States Policy with 2.0 GHz shows the greater decrease in energy consumption, its service time violates the imposed deadline of 1 second in about 10% of the service time, rendering this particular policy not useful for practical settings. The 2.6 GHz meets the deadline in all cases but has its energy consumption 7% higher than the Three States policy.

Since the energy consumption of the Three States policy is the best among the policies that don't violate the deadline, it is the most suitable choice for this problem.

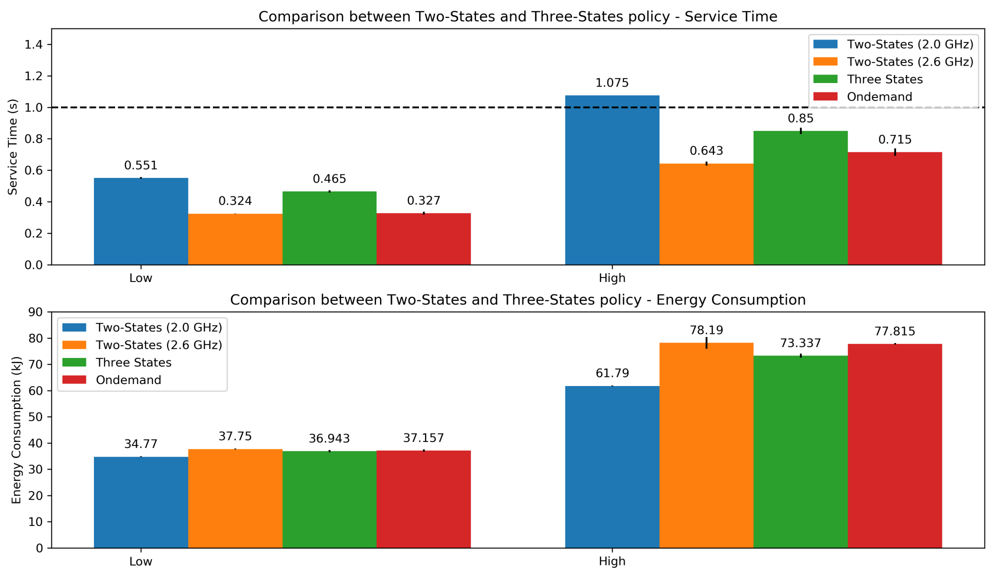

Figure 4.5 Comparison of Two-States vs Three-States Policy.

Although we did not empirically test this fact, we believe that, with a large spectrum of available frequencies and with more request classes, a granularity with 4-states or beyond may produce better results.

4.5 OVERHEAD ANALYSIS

Another type of analysis is the actual overhead given by the governor code embedded into the Elasticsearch code. For measuring this, we used the same Hurry-up code without any frequency change (which means that the scheduler will run at a fixed frequency) and compared against a fixed frequency run without the scheduler. Figure 4.6 illustrates the overhead level.

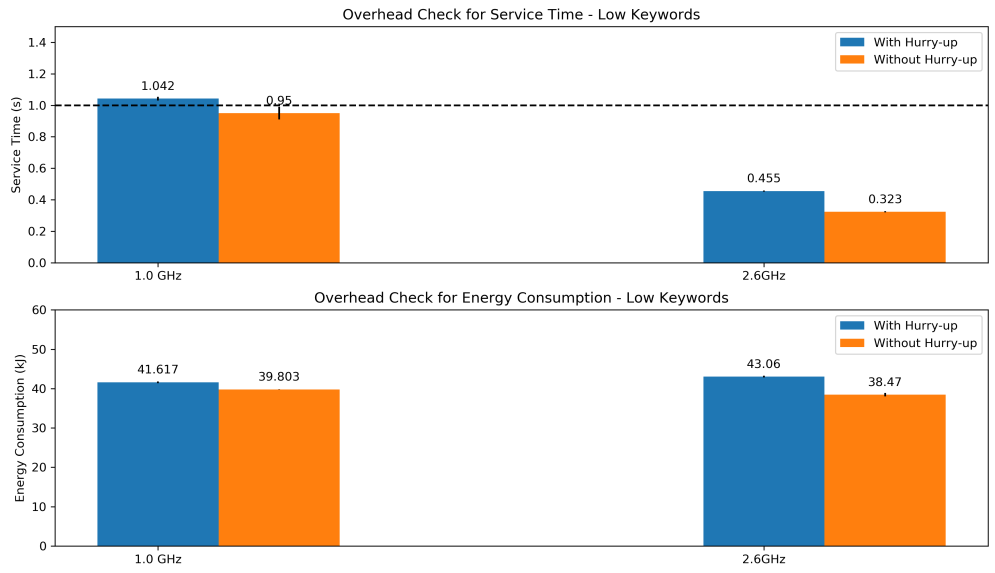

Figure 4.6 Overhead Analysis of the Hurry-up Scheduler

The idea behind running only low queries consisted of reducing eventual stress that might be caused by high-load queries, thus amplifying the signal-to-noise ratio. In sum, there's a moderate to severe overhead with the agent introduction. For service time, the overhead averages to about 0.1 seconds in absolute term, but amounts to about 10% at 1.0 GHz and 40% at 2.6 GHz. This is less problematic on the energy consumption side, where the energetic overhead from the scheduler is 5% for 1.0 GHz and 11% for the other case.

Our hypothesis for such overhead is because the agent interrupts the search thread for a brief moment in order to collect data regarding the entry or exit events from the hot function, and also the fact that the agent itself generates one more thread that performs operations every certain time. The latter effects can be easily seen at Section 4.7, where simply increasing the awakeness parameter means a lower energetic consumption.

4.6 C-STATES ANALYSIS

Analyzing the processor's C-States is necessary to understand the behavior of both Ondemand governor and the Hurry-up scheduler. Refer to Table 2.2 for an explanation about the four available C-States at the Intel Xeon Gold 6182 processors.

The C-States can be controlled through the */sys/fs* interface and it has a per-core per-state granularity. First, we compare how Hurry-up and the Ondemand governor performs in service time and energetic consumption when only the C0 state is available and when all states are available. For this particular experiment, each run lasted 10 minutes in the same environment as the previous ones. Figure 4.7 illustrates the results for this.

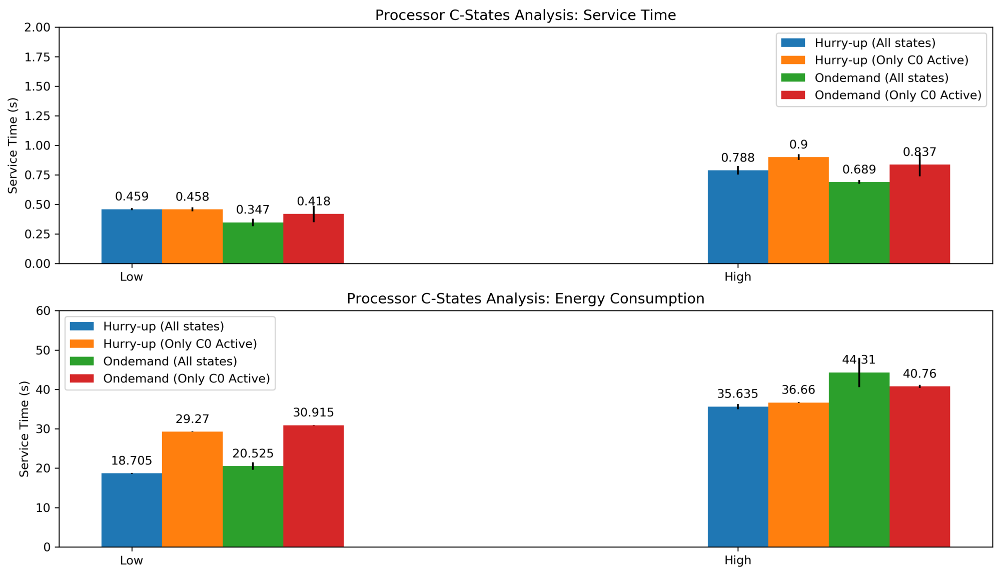

Figure 4.7 C-States Analysis of the Service Time and Energetic Consumption

While we can't make a definitive correlation between the C-states and the service time, it is easily seen that the energetic consumption for Only-C0 is higher than when executing in 'All States' mode. This is because the consumption when the processor is at sleep state is lower than running at the lowest available frequency of 1.0 GHz. Figure 4.8 shows the time spent (in clock-tick units) per state when running with all states enabled.

For low keywords, it is clear that both Hurry-up and Ondemand governors spend a lot of time at the C6 state, meaning that they're most of the idle time at the lowest energetic level. The difference in consumption is theorized by the fact that Hurry-up executes low queries at the optimal level of 2.0 GHz, while Ondemand uses 2.6 GHz. On a very long run, a higher difference in consumption may be seen.

For higher keywords, it is seen that Hurry-up spends more time on activity (C0 State), but at a lower frequency. This lower frequency is optimal, as seen in Chapter 3, and guarantees a lower energy consumption even at the cost of delayed service time.

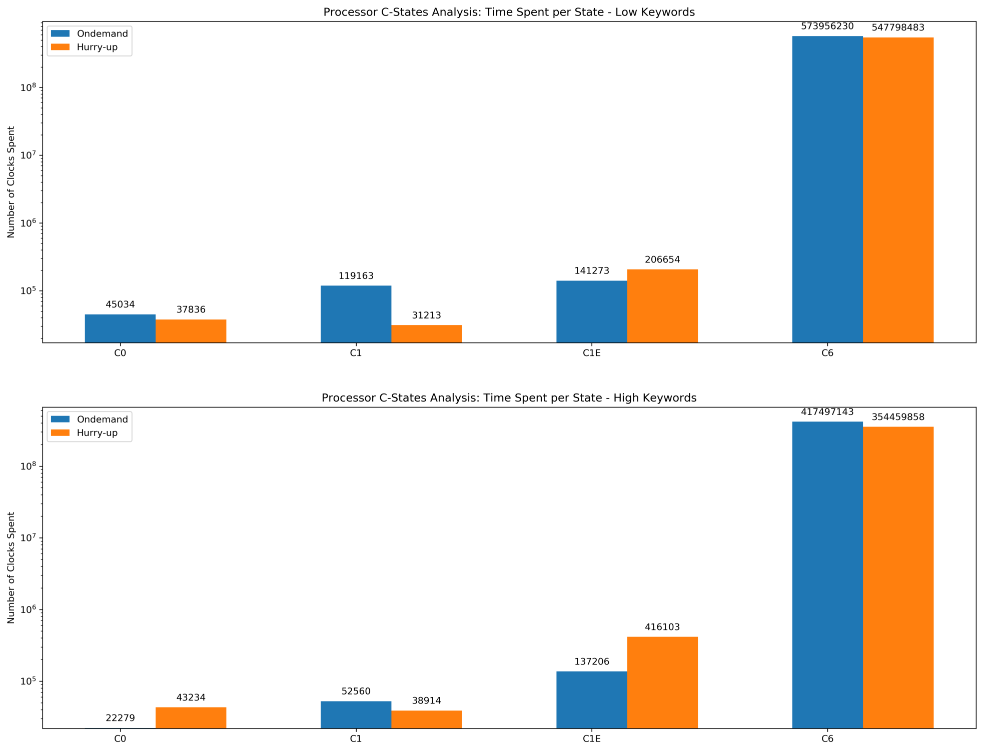

Figure 4.8 Calls for each C-State per Keyword length

4.7 SENSITIVITY ANALYSIS

The Hurry-up Scheduler allows us to choose two parameters: the awakeness - which is responsible for the sampling time of the scheduler - and the threshold - the time when we consider the request as a heavy request and we need to change frequencies.

For testing the threshold, we fixed the awakeness time at 10 ms and varied the threshold for the values of 150 ms, 300 ms, 450 ms, and 600 ms. Figure 4.9 shows the results for this experiment. In a nutshell, energy consumption tends to decrease as the threshold increases - and the service time goes on the opposite way. While we don't care about the threshold for low queries - since most of the queries won't even need going to the highest frequency -, the same doesn't apply to higher ones.

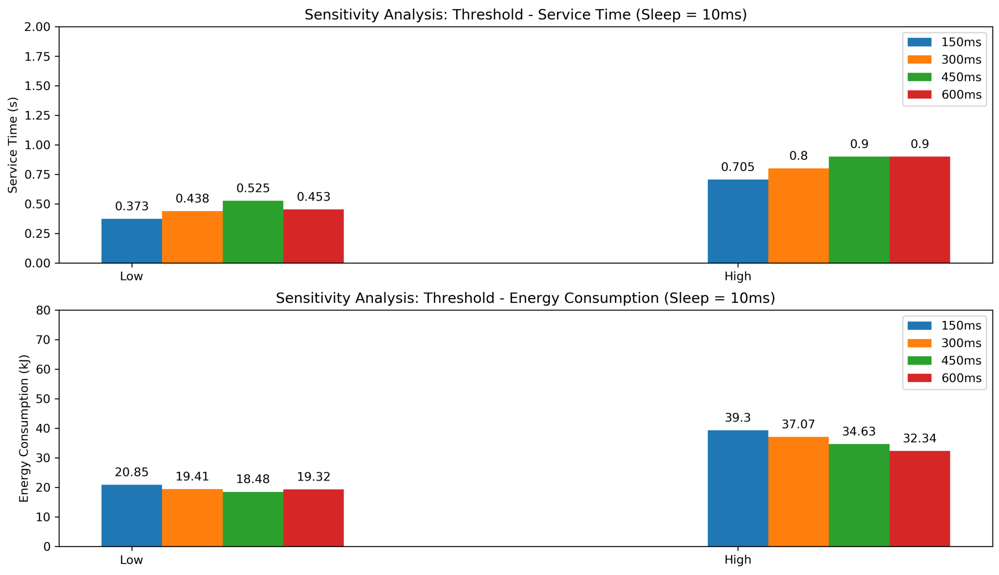

Figure 4.9 Sensitivity Analysis of the Threshold at Hurry-up

Higher queries mean that the threshold will be an important parameter on how much time they will run at either 2.0 or 2.6 GHz. As the threshold goes up, the higher the time at 2.0 - thus a decrease in energy consumption as it's already expected. The chosen parameter in this case was 450 ms. Although the 600 ms threshold might appear to be the best, we believe that it can be an issue when the load consists of only very-high queries - thus, the service time would be a lot more affected.

For the awakeness parameter, we fixed the threshold at 300 milliseconds and tried values of 5, 10, 20, 50, and 100 ms. Results can be seen in Figure 4.10. Overall, there's a small service time overhead when using very small or very large values. In the end, we chose 50 ms for the optimal awakeness parameter.

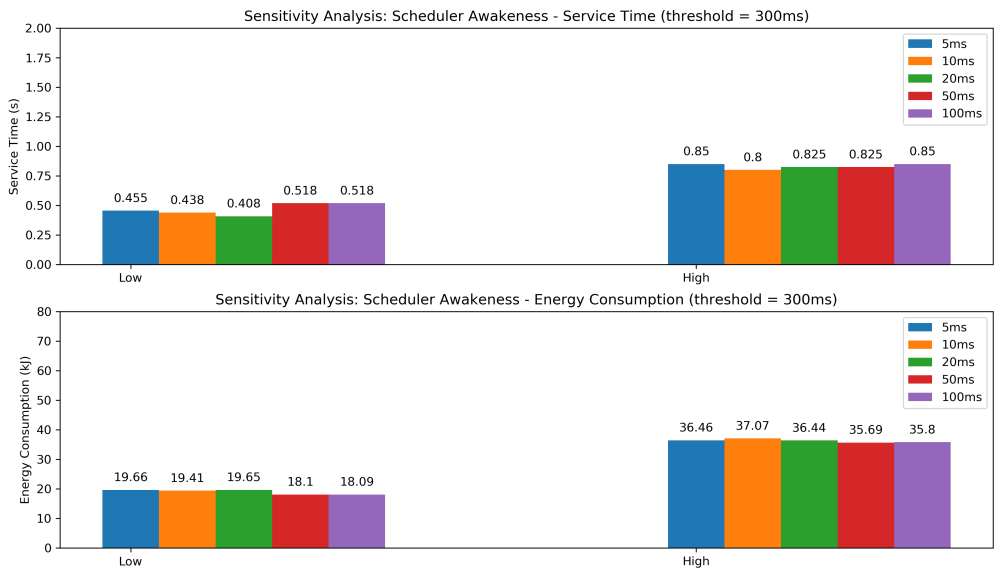

Figure 4.10 Sensitivity Analysis of the Threshold at Hurry-up

4.8 FINAL RESULTS

With the sensitivity analysis done, we re-calibrated the Hurry-up Scheduler with the 450 ms for threshold and 50 ms for sleep time. We gave 20-minutes runs for each experiment, with the same characteristics as described in Chapter 4.3. Results can be seen in Figure 4.11. While both heuristics meet the deadline, Hurry-up has the power consumption about 17% less than the one by Ondemand at the high key length. The Zipfian distribution, used by FABAN and CloudSuite, favors a low key length, between 1 to 6, hence why the "Mix" has a lower service time than the "Low" (which is fixed at 4).

While small, the difference between Ondemand and Hurry-up in energy consumption favors Hurry-up in about 2%. To check if this rate will maintain, we increased the request-per-second rate in three-fold for both schedulers. Note that, at the high key length, both schedulers lose their deadlines, hence why they're not shown here. Figure 4.12 illustrates the obtained results.

Both key lengths, while meeting their deadlines, had its difference in energy consumption amplified. In particular, the "Mix" version had its difference increased to 22% while the "Low" key length, 28%! This happens because, due to the CPU stress, Ondemand stays most of the time at the highest available frequency - 2.6 GHz - and while it's the best for service time, it's not the optimal for energy consumption. Since Hurry-up is able to answer all requests within the deadline at the optimal configuration, it saves a lot more energy.

As a note, for the "Mix" key lengths, Hurry-up is still capable to answer within the deadline with a four-fold base load, although most of the answers will fall just within

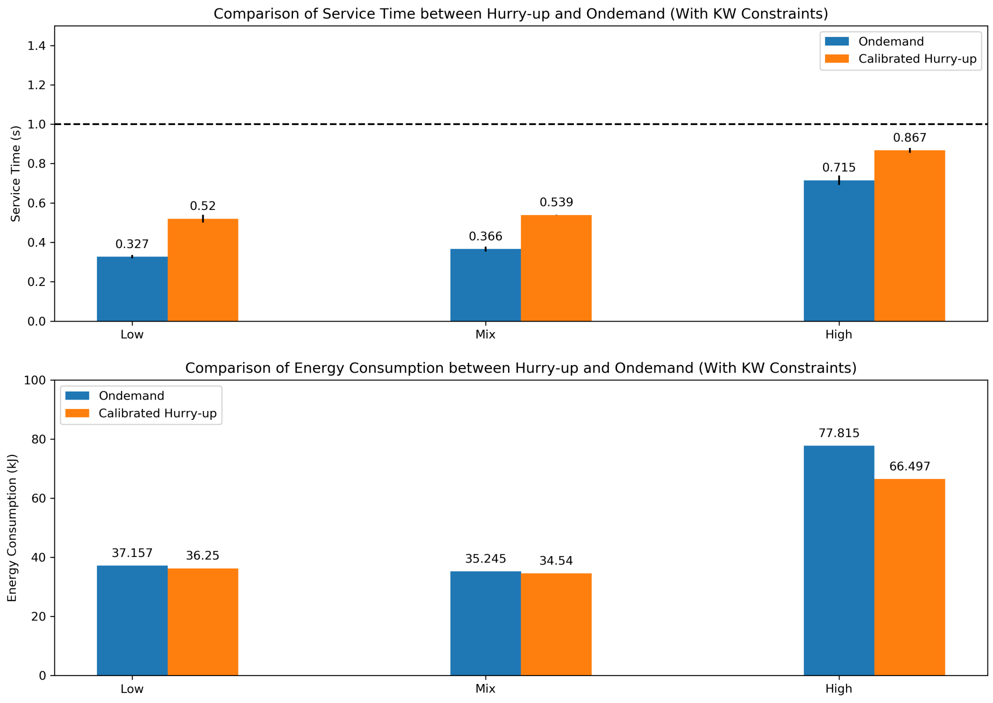

Figure 4.11 Comparison of Hurry-up and Ondemand schedulers at Base Load

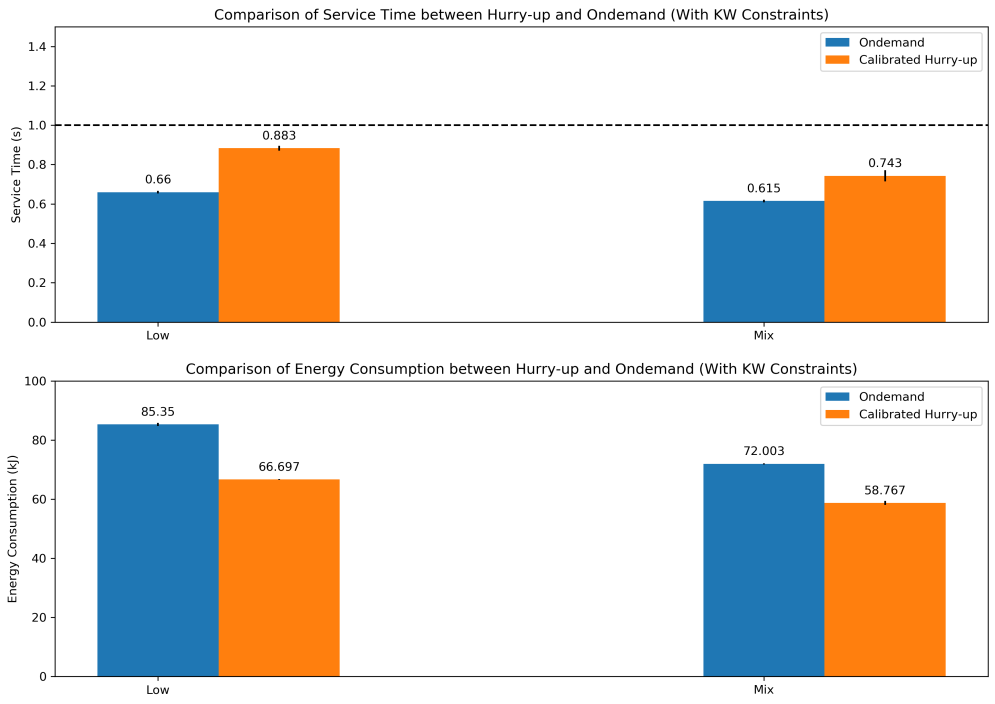

Figure 4.12 Comparison of Hurry-up and Ondemand schedulers at 3x Base Load

the imposed deadline. Above that load, modifications on the parameters described by Chapter 4.7 will be necessary since there will be the need to improve service time as well. Some suggestions to improve the scheduler can be seen at Chapter 5.

Chapter 5

CONCLUSIONS

The present work has shown that it is possible to optimize the energy consumption using DVFS while maintaining a similar level of Quality of Service (QoS) compared to existing techniques. We analyzed how search requests behave and which frequency is optimal for each request class. We developed an approach based on the assumption that each request class deserved its own optimal frequency and implemented it for Elasticsearch/Lucene. This scheduler was compared against the Ondemand governor, which is default on Linux, and obtained energy-savings up to 30%.

A summary of our findings:

- In search workloads, we observe varying service times and, thus, an energy usage difference between light and heavy requests.
- The optimal frequency for running lighter and mid requests was found to be a mid point (2.0 GHz in an Intel processor), as it can provide the best Millions of Instructions per Joule (MIPJ).
- As for heavy requests, since they're prominent to lose their deadline, it's necessary to run them at the highest available frequency - 2.6 GHz.
- If the processor is not able to return to the C6 state for idling, the lowest energy consumption will be at 1.0 GHz.
- Implementing the Three States Policy, based on the empirical observations from above, for search workloads, our approach can save up to 30% on energy consumption against the Ondemand governor on Linux.

5.1 SUGGESTIONS FOR FUTURE WORKS

There are three suggestions - all of them are independent from each other - to improve this work.

1. **Advanced Profiling:** There is a need to establish a more methodical way - either through metrics or an algorithm - to define the hot function. This work used only one very-well defined hot function because the main purpose of Elasticsearch is to search - hence it's obvious why the hot-function would be search-related. Some applications may behave differently or have different purposes and have two or more non-correlated hot functions, and finding them automatically will make the instrumentation process easier.

2. **Multi-hot functions:** While we used only the Search hot function for Elasticsearch in this work, there might be others - such as a hot function for Indexing or even managing multiple shards across a distributed system. For an application to perform even better, those non-main hot functions might have to be taken in account as well.

3. **Multi-states:** The present work uses only two types of loads: low/mid and high. In some cases, there may be a necessity to define more types of loads, such as ultra-low or even ultra-high - for this, more states can be useful to optimize the best MIPJ metric for each situation.

4. **Adaptive Threshold and Sampling Period:** The patterns of keyword length alongside the rate of queries per second that arrives at the cluster may not always be constant. In this case, both the threshold and sampling parameters may be useful for controlling how the scheduler reacts. A good example is that for very high loads and a very high rate of queries per second, it may be in scheduler's interest to minimize the threshold value while using higher values of sampling period as there will be the need to run constantly at higher frequencies to lower the total queue time.

REFERENCES

Barroso, Clidaras and Hölzle 2013 BARROSO, L. A.; CLIDARAS, J.; HöLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Second. [S.l.]: Morgan & Claypool Publishers, 2013. (Synthesis Lectures on Computer Architecture).

Brin and Page 1998 BRIN, S.; PAGE, L. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 30, n. 1-7, p. 107–117, abr. 1998. ISSN 0169-7552. Available from Internet: ⟨http://dx.doi.org/10.1016/S0169-7552(98)00110-X⟩.

Chen, Christina and Martínez 2019 CHEN, S.; CHRISTINA, D.; MARTÍNEZ, J. Parties: Qos-aware resource partitioning formultiple interactive services. In: ASSOCIATION FOR COMPUTING MACHINERY. *2019 Architectural Support for Programming Languages and Operating Systems.* [S.l.], 2019.

Coulouris et al. 2012 COULOURIS, G. et al. *Distributed Systems: Concepts and Designs.* Fifth. [S.l.]: Pearson, 2012.

Dean and Barroso 2013 DEAN, J.; BARROSO, L. A. The tail at scale. *Communications of the ACM*, v. 56, n. 2, p. 74–80, feb. 2013.

Ferdman et al. 2012 FERDMAN, M. et al. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In: *17th International Conference on Architectural Support for Programming Languages and Operating Systems.* [S.l.: s.n.], 2012.

Guliani and Swift 2019 GULIANI, A.; SWIFT, M. Per-application power delivery. In: *Fourteenth EuroSys Conference.* [S.l.: s.n.], 2019.

Haque et al. 2017 HAQUE, M. et al. Exploiting heterogeneity for tail latency and energy efficiency. *MICRO-50*, p. 625–638, oct. 2017.

Kasture et al. 2015 KASTURE, H. et al. Rubik: Fast analytical power management for latency-critical systems. In: *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* [S.l.]: IEEE, 2015.

Lo et al. 2014 LO, D. et al. Towards energy proportionality for large-scale latency-critical workloads. In: *IEEE/ACM 41st International Symposium on Computer Architecture (ISCA).* [S.l.]: IEEE, 2014.

Lo et al. 2015 LO, D. et al. Heracles: Improving resource efficiency at scale. In: ASSOCIATION OF COMPUTING MACHINERY. *42nd Annual International Symposium on Computer Architecture*. [S.l.], 2015.

Mastrangelo and Hauswirth 2014 MASTRANGELO, L.; HAUSWIRTH, M. Jnif: Java native instrumentation framework. In: *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools*. New York, NY, USA: Association for Computing Machinery, 2014.

Nishtala et al. 2017 NISHTALA, R. et al. Hipster: Hybrid task manager for latency-critical cloud workloads. In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. [S.l.: s.n.], 2017.

Pallipadi and Starikovskiy 2006 PALLIPADI, V.; STARIKOVSKIY, A. *The Ondemand Governor: Past, Present and Future*. [S.l.], 2006.

Petrucci et al. 2015 PETRUCCI, V. et al. Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers. In: *High Performance Computer Architecture (HPCA)*. [S.l.]: IEEE, 2015.

Schurman and Brutlag 2009 SCHURMAN, E.; BRUTLAG, J. *The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search*. 2009. Research Presentation.

Weiser et al. 1994 WEISER, M. et al. Scheduling for reduced cpu energy. In: *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. [S.l.: s.n.], 1994.

Yeh, Marr and Patt 1993 YEH, T.-Y.; MARR, D.; PATT, Y. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In: *Proceedings of the 7th ACMInternational Conference on Supercomputing*. [S.l.: s.n.], 1993. p. 67–76.