Master's Proposal

Profile-guided heterogeneous-aware scheduling for cloud workloads

Daniel Araújo de Medeiros

August 2018

Programa de Pós-Graduação em Ciência da Computação (PGCOMP)
Universidade Federal da Bahia (UFBA)
Salvador, BA

Thesis Committee:
Vinicius Petrucci, Chair (UFBA)
George Marconi de Araújo Lima (UFBA)
Daniel Mossé (University of Pittsburgh)

Submitted in partial fulfillment of the requirements for the degree of Master in Computer Science.

Abstract

Task scheduling is an essential technique to improve resource efficiency, especially in nowadays heterogeneous architectures capable of executing tasks on big/high-performant and little/power-efficient cores. Tail latency is a critical factor affecting user experience in cloud services such as web search and social networks. The tail latency, typically given by 95th or 99th percentile latency, is heavily influenced by the slowest response time from a poor performing processing core, and its impact is greatly amplified at scale as more heterogeneous servers are hosted in a data center. Prior work on task scheduling for heterogeneous cores are coarse-grained in the sense that it maps the entire application to a mixed set of processing cores (no distinction between application, threads, or functions). It also may depend on a substantial runtime process externally to collect the dynamic task behavior.

In this thesis, we propose a finer-grained scheduling approach that leverages information about function activations for request processing in cloud applications. The approach works by automatically enabling the allocation of a more powerful processing core to an individual thread of the application once it enters in a hot (compute-intensive) function; on the exit of the hot function, the thread can be mapped to a slower core to save power or to leave space for executing a more demanding thread. We envision the approach consisting of (1) a characterization method for identifying hot functions of the application that are closely correlated to the tail latency and (2) an instrumentation methodology to automatically insert adaptation points into the application code. We will implement and evaluate the proposal on a real heterogeneous multi-core system (ARM Juno Board) running established cloud applications, such as web search and data sharing. We will carry out experiments comparing the proposal with state-of-the-art scheduling designs so as to demonstrate the benefits of the approach.

1 Introduction

Modern applications are moving from desktop clients to smaller, mobiles devices; however, the large processing and storage requirements need powerful servers in the cloud [3]. Examples include services such as video streaming, file sharing, and office applications, that were used to run exclusively on desktop clients and now those services run in cloud platforms. For these services, it is critical to deliver a satisfactory metric of quality of service (QoS) because service delays may affect user experience and negatively impact companies' revenue. A study revealed that a delay of 2000 milliseconds in returning web search results may impact revenue in over 4% per user [19]. 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-5% (slowest) distribution of the service's response time [7].

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 [7]. 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 [21]) and software-side (e.g., canary requests [7]) have been developed and deployed in modern cloud systems.

From a computer architecture perspective, there is a drive to explore heterogeneous multi-core systems with different micro-architecture characteristics, which can be individually classified as little/little or big/big cores [12]; the former is slower but power-friendly, while the latter provides high single-core performance. Assuming that CPU power decreases by approximately O(k^2) when CPU frequency decreases by k, the use of many little core system in large-scale homogeneous systems seems reasonable for request-parallel cloud services. However, this is not practical due to Amdahl's law where the difficulty of reducing serialization and communication overhead is proportional to the speedup of using parallel threads [12]. Modern multi-core systems are designed to explore dynamic voltage scaling systems (DVFS), where the frequency of operation may increase or decrease according to the demand, or as an asymmetric multiprocessing system (AMP), which explores two or more distinct micro-architectures — ARM's big.LITTLE technology [9] is an example of this kind of system.

Traditional thread scheduling systems implemented in most common operating systems, such as Linux's Complete Fair Scheduler and Windows's multilevel feedback queue, are unable to make rational use of the heterogeneous processors to meet tail latency with improved energy efficiency for cloud applications. Alternative scheduling approaches have been developed in the literature, making use of features like statistical prediction [13], feedback-control state machines [18] and reinforcement learning [17]. Such approaches, however, are coarse-grained that works by mapping the entire workload onto the heterogeneous cores, and usually require external runtime system that may incur some overhead in the system. 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, that could be explored in a more fine-grained task mapping.

This research aims to address the problem of thread scheduling using heterogeneous cores for improved tail latency and energy efficiency. By exploring characteristics of the software via code instrumentation, we propose a new scheduling algorithm guided by compute-intensive (hot) functions identified in the applications through profiling and instrumentation techniques.

It is worth noting that this is a collaborative effort that involves the following people: Denilson Amorim (undergrad student at UFBA), helping with some of the implementation parts of the project (mainly regarding code instrumentation); Rajiv Nishtala (postdoc researcher at NTNU), helping with the experimental platform and with constructive feedback on the scheduler design. The experiments in this proposal are being carried out on the ARM Juno board (big/little architecture) at the Barcelona Supercomputing Center in a collaborative way with Paul Carpenter (Senior Researcher).

In Section 2, we give an overview of the research context including a brief literature review. Section 3 presents the thesis statement and methodology. Section 4 outlines the thesis organization to be undertaken and Section 5 describes a timetable for the execution of this proposal.

2 Background

2.1 Cloud Workloads

Cloud services including web search and social networks represent the vast majority of user activity on the Internet. The main feature of this kind of workload is the need for displaying combined results from datasets scattered over multiple servers in near real-time. This poses strict requirements on end-user latency [16]. Two types of cloud workloads are investigated in this proposal: web search and data sharing (NoSQL database).

Web Search — Brin and Page (1998) [4] *apud* Coulouris et al. (2012) [6] describe how a web search service works, in particular Google's search engine. The description below is a high-level overview of a typical search application. A search engine is composed by three major applications: crawling, indexing and ranking. Crawling is related to fetching the contents of 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.

The crawled pages are parsed and each document is converted in a set of word occurrences called hits; these are composed by the word, position in 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 anchors file, which are read by an URLresolver and converted into docIDs. The URLresolver 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 number of users, the query evaluation procedure might take longer times, 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 an answer in hundreds of milliseconds for fluid user experience.

Data Serving — Another established latency-sensitive application is data serving. Apache's Cassandra [14] is a NoSQL database, used by Facebook, Netflix and other companies, that shares some characteristics with a search service application. While Google's Engine is solely focused on attending user requests through searching, Cassandra is about writing data into massive no-relational databases and later reading it. In the example case, it's used for Facebook's inbox which requires both enormous write and read throughput due to the large number of users and data. For this purpose, Cassandra is built through distributed systems techniques such as partitioning, replication, membership, failure handling and scaling. For this specific proposal document, Data Serving will not be throughly analyzed. This will be done in the final thesis document. However, it's important to give a general overview of how Cassandra's read/write requests works, as described by Lakshman and Malik (2009).

The data model in Cassandra is composed of a table through a distributed multidimensional map indexed by a key, which value is a string without any particular size restriction. Cassandra's API is covered by the insert (write), get (read) and delete methods. When any of the insert or get methods for a key is called, the request is routed to any node in the cluster where the replicas for this specific key are determined. For insert requests, the request is routed to the replicas and awaits for a certain quorum of replicas to acknowledge the completion of writes. For get requests, based on client-required consistency guarantees, the system chooses between routing the requests to the closest replica or routing the requests to all replicas, while awaits for a quorum of responses as well.

Cassandra is also a latency-sensitive application. It stores the most frequent keys in either cache or key caches, which are fast enough for quick data lookup. However, most of non-hot keys are stored in disk through a structure known as SSTable, which is slower both because of its size (the search scope is bigger) and the input/output speed. For requests inside any SSTable, it might violate the imposed deadline - hence, the use of big cores in this situation to accelerate search requests inside it should be necessary for the tail latency guarantees.

2.2 Motivation: Web Search

For web search we will explore the Elasticsearch benchmark, a search service application built atop of Apache Lucene [2]. Our proposal is motivated by the following experiments. The first experiment, shown in Figure 1, compares the tail latency between big and little cores when executing 200 thousands search requests. Elasticsearch was indexed with 10 gigabytes of Wikipedia text, publicly available by Wikimedia Foundation. In the server side, we used a 64-bit Juno board (big/little architecture) running Debian Linux as the operating system. An external machine, located in the same LAN Network, was used to send 200 thousand requests of keyword length between 1 and 14 based using a negative exponential distribution (similar to prior load generators like Faban [1]). Each request was serviced one at a time and the time taken in the server side was recorded. Data was statistically treated "as is", without any removal of possible outliers.

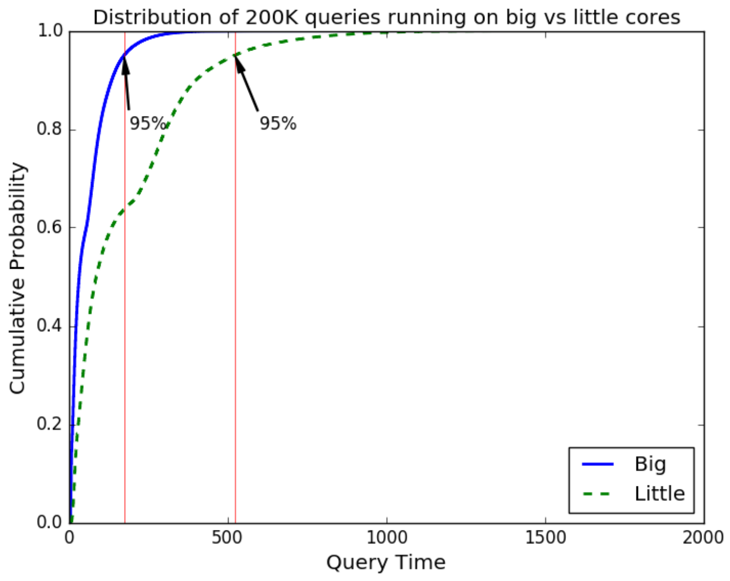

Figure 1: Tail latency comparison between big and little cores (200K requests)

We can see from the results shown in Figure 1 that 95% of all requests on big cores were returned within 170 milliseconds while the little cores took over 500 milliseconds. This is particularly important because it shows the difference in service time response between these core types. Thus, little cores are more prone to the effects of tail latency, as described in Section 1, even though they may be more power efficient.

The second experiment, shown by Figures 2 to 4, illustrates the influence of the keyword length on the service time. We notice that the search application is susceptible to heavy and light requests. In this experiment, 50K requests were sent to the web search using only the big cores (controlled using Linux's taskset for CPU affinity).

Figures 2 and 3 are summarized by Figure 4. When checking the 95-percentile service time for each keyword, we can see a direct correlation between the keyword length and the service time, growing almost linearly (see the trend from 1 to 4 keywords and from 11 to 14 keywords).

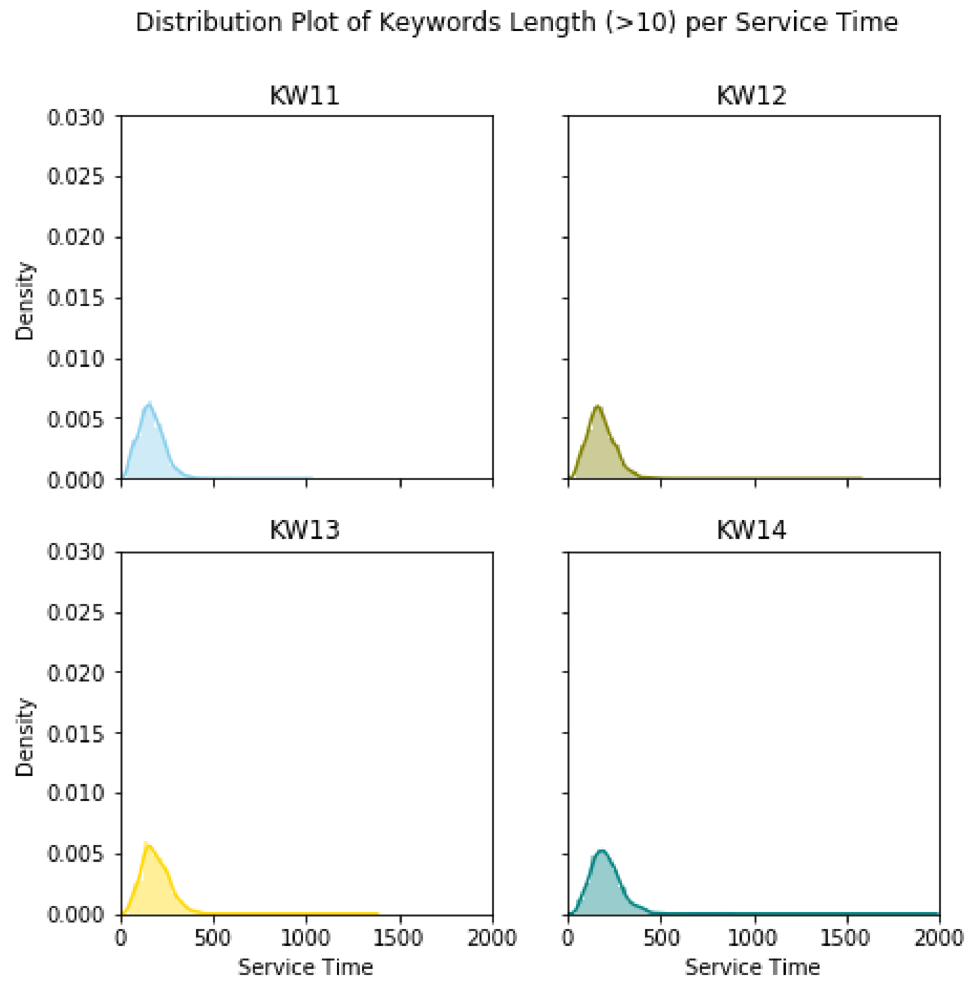

Figure 2: Distribution plot of the service time taken per keyword length (>10) on big core.

However, the irrational use of big cores may have some effects on the monetary health of a corporation. Shehabi (2016) [20] estimates the total energy used by data centers at United States of America as 73 billion Watt, where servers alone consumes 40 to 80% of this total, and Haque (2017) [10] says that by approximating the cost of a kiloWatt-hour for values between 4 and 21 cents, the total account would be located between 1 and 6 billion dollars.

The third experiment tries to evaluate the energy consumption of cores in Juno board for different keyword sizes. As it was shown by experiment 2, there's an intimate relationship between the keyword size and the service time, so the hypothesis is that requests with longer keyword length that were scheduled to big or little cores took more time to be processed and thus had bigger energy consumption. Figures 5 to 7 shows the obtained results.

The energy measurement was done through the reading of the Juno's energy registers for A57 (big) and A53 (little) clusters. A custom version of Elasticsearch, compiled with some Java Native Interface (JNI) code that linked

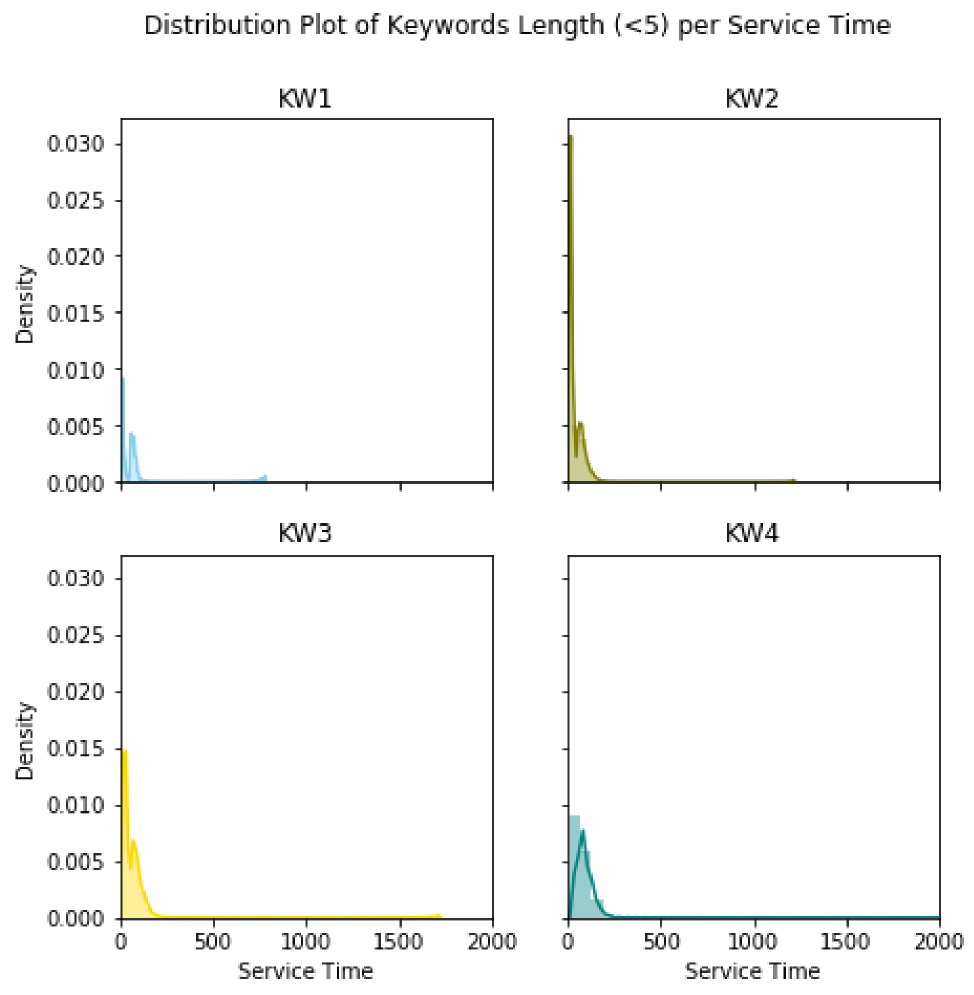

Figure 3: Distribution plot of the service time taken per keyword length (<5) on big core.

the energy monitor provided by ARM, was created. Also, some sandbox security mechanisms of the Java Virtual Machine had to be disabled through a specific flag when running this new version of Elasticsearch.

Every time a certain chosen search function - which is responsible for about 80% of Elasticsearch's execution time - was called, the application would check the energy register and save it into a log file for entry and exit of the function. A number of 40 thousand requests were sent to Elasticsearch, generating 160 thousand entries for each of the studied case. The scenario described as "Big (14KW)" in figures 5, 6, 7 means that all the requests were directed to the A57 cluster through Linux's *taskset* command and the requests contained keywords of length between 8 and 14. For the 5KW case, the length was between 1 and 6 keywords. The same applies for the the little (A53) cluster.

For figures 5 and 6, as the output of the energy monitor was the accumulated energy of the board, a preprocessing of the data was done and the difference between the accumulated energy at the exit of the function and at the entry was calculated. Possible outliers, such as negative energy values or very discrepant values were removed before the generation of the boxplots. For figure 7, the total energy is simply the difference between the initial measured energy and the final one for each cluster.

As expected, the case where longer keywords ran had bigger energy consumption. Not only this, but running on big core spends more energy than on little on all cases. It is important to mention that although the scenarios for BIG (5KW) and LITTLE (5KW), mainly of figure 6, look virtually the same for statistical purposes, it may be more efficient to run requests with smaller keyword length on the little cluster than on big cluster because you could replace N big cores with N*K little cores, where K is typically determined by a factor of the power budgets given to

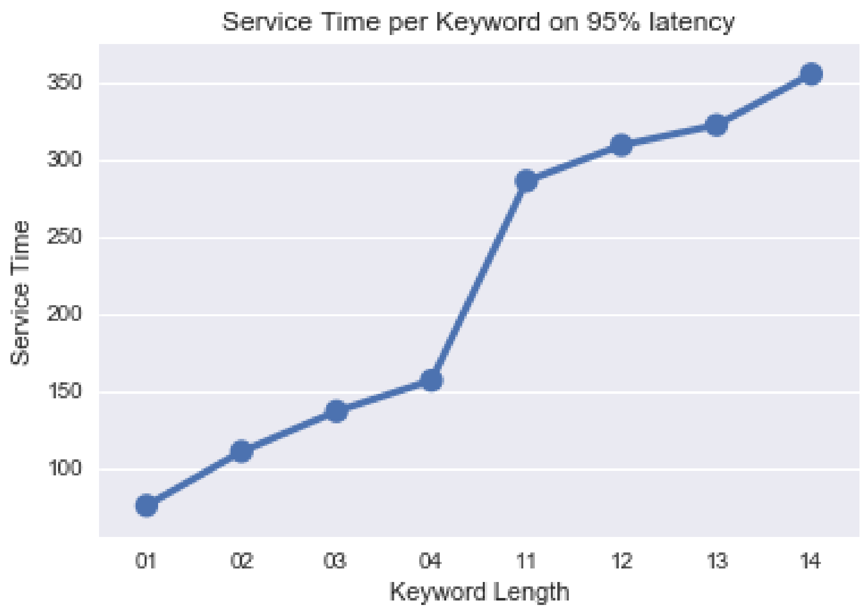

Figure 4: Scatter plot of the service time taken per keyword length on big core.

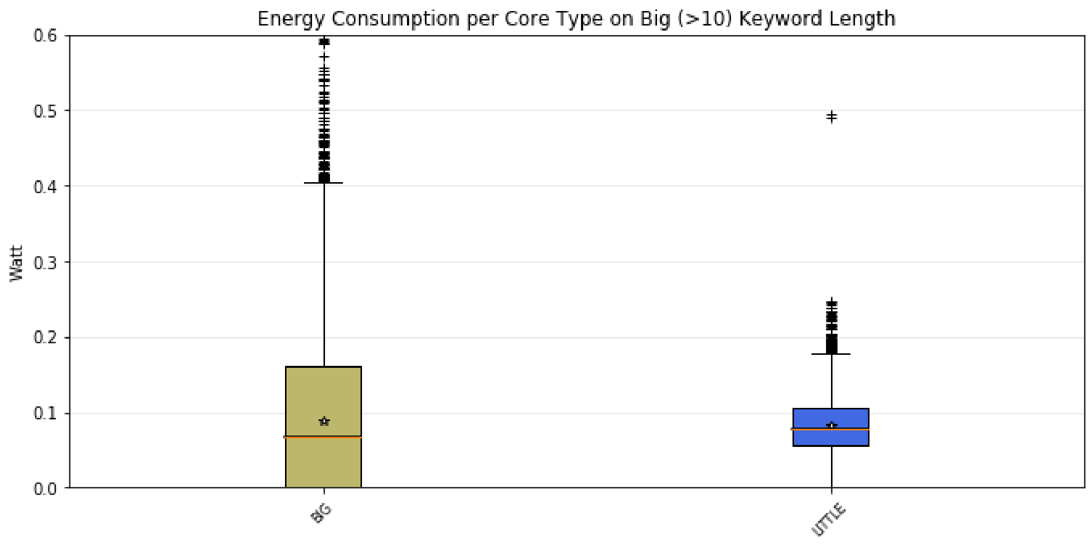

Figure 5: Energy consumption (mW) per keyword length and core type.

the core types at design stage [11].

By allocating requests that may take longer execution times to big cores, the assumption is that the energy consumption will keep to the minimum necessary while guaranteeing the quality of service of the application. However, although the keyword length request is associated to the service time, it is hard to predict the keyword length or even if the request will take a long time.

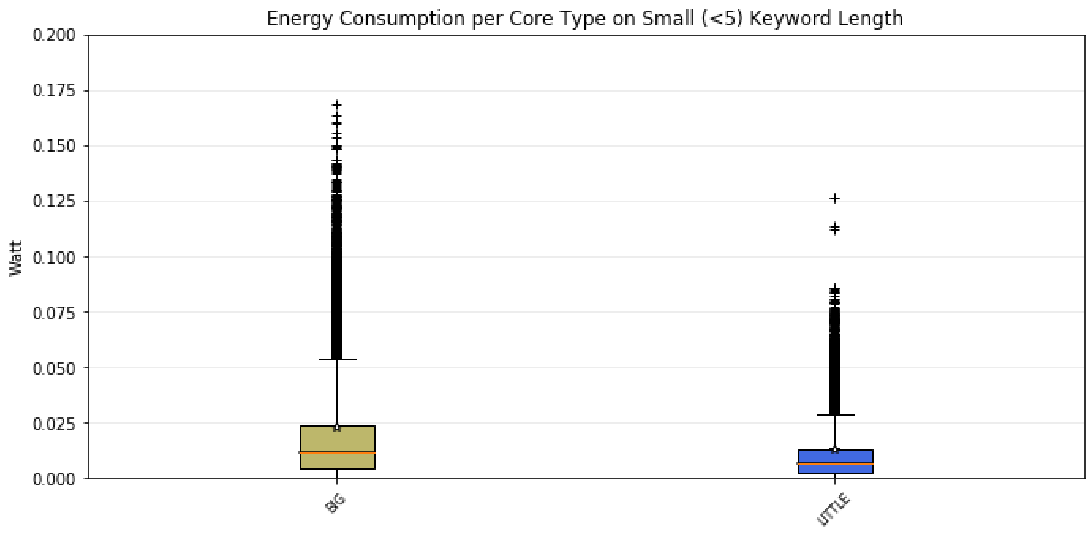

Figure 6: Energy consumption (mW) per keyword length and core type.

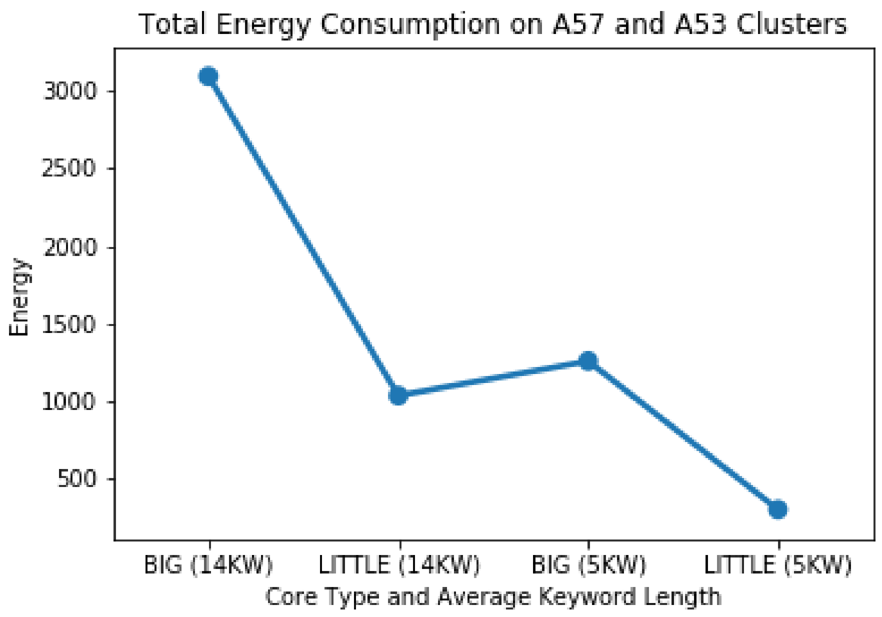

Figure 7: Total energy consumption (mW) per keyword length and core type.

2.3 Literature Review

Several approaches to this problem were already developed in the literature, exploring different concepts and techniques. Pegasus[15], Rubik[13], OctopusMan[18], Hipster[17] and Adaptative Slow2Fast[10] are the ones most closely related to our proposal and will be discussed here.

Both Pegasus and Rubik 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 fastest as possible. For this, a feature present in Intel-only processors - Running Average

Power Limit (RAPL) - was utilized, and it allows the user to set up a energy limit where the CPU never should exceed.

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

Nevertheless, important to say that this approach has two problems: first, it relies on a feature present in Intel-only processors, which may not be interesting for those who work with others types of processor architecture. Second, this works only for DVFS systems, not working on heterogeneous systems.

Rubik is also a DVFS approach, but without utilizing that processor-specific feature. The kernel of the idea is a statistical model that utilizes a dynamically collected service time distribution in order to handle the uncertainty from computational needs of individual requests. This allows the distribution predict 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.

OctopusMan was designed to meet the desired quality of service while having the lowest energy consumption. The algorithm is constituted by a Quality of Service Monitor and a Mapper. The first is responsible to collect performance data of the heterogeneous service through profiling techniques and by analyzing application-level (e.g.: latency for web search requests) metrics and operating systems metrics (e.g.: CPU utilization, cycles). The latter takes decisions based on those metrics and adapts the system for energy efficiency. The mapper has two designs: the first one is based on a proportional-integral-derivative (PID) system and the second works by a state machine. The available configurations of a processing core for the controlled systems are represented by the possible states and each element of the little or big set is mapped to another state in the transitioning system. The former also occurs from a state to another when some trigger conditions - specified by quality of service rules - are met.

Hipster utilizes an hybrid approach of Reinforcement Learning (RL), which has 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 where the algorithm is rewarded by a point if it's correct and punished if error. Regarding the implementation, Hipster has two variants: HipsterIn, for optimized energy efficiency of latency-critical only tasks, and HipsterCo, which allows to run latency-critical along with others jobs for better server resource management.

Like OctopusMan, Hipster uses a 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 the exploitation phase, Hipster maps the tasks to the core based on the reward mecanism (HipsterIn or HipsterCo) and updates the lookup table. If the QoS guarantee falls bellow the target, Hipster goes automatically to the learning phase.

The **Adaptative Slow to Fast** (AS2F) algorithm aims to explore the heterogeneity on a finer-grained manner in both DVFS and AMP 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 has a feedback-based control that computes the threshold migration time based on the measured tail latency, the target one and the system load. The online phase consists on thread mapping based on thresholds and task progress.

3 Proposal

We observe that prior scheduling approaches do not explore any specificities of the application itself, only those characteristics obtained by the runtime/operating system (e.g., load usage), and they map the entire application across heterogeneous core configurations. In contrast, our hypothesis is that:

Thesis Statement: *The use of a finer-grained, code-instrumented scheduling policy, can further improve the performance and energy efficiency of cloud applications.*

The proposed scheme in this work will be designed with following characteristics:

- **Application code structure:** The proposed algorithm should be able to use characteristics of the application in a finer-grained way, such as exploring function activations.
- **Controlled overhead:** Assuming that there's a batch of tasks to be scheduled and with a very small deadline, the possible introduction of a large overhead by the scheduler may significantly impact the task execution.
- **Programming-language independent:** Such algorithm or methodology should be generic enough to be not restrict onto a single programming environment (either operating systems or virtual machines).
- **Useful for multiple architectures:** The proposed algorithm must be able to model itself to n types of processors, be it DVFS or AMP, as long as it is possible to measure performance for each one of them.

We believe that the methodology and algorithmic ideas developed by this thesis, and discussed in next section, will enable more service providers to utilize instrumentation techniques to improve energy efficiency while guaranteeing the desired quality of service. All the code and scripts developed in this thesis will be publicly available under Apache 2.0 License.

3.1 Methodology

3.1.1 Experimental Infrastructure

Our algorithm will be executed on a 64-bit ARM Juno board, which consists of two A57 high-performant cores and four A53 power-efficient cores. This board is currently located at Barcelona Supercomputing Center under the supervision of Paul Carpenter (Senior Researcher). The analyzed dataset will be obtained from cloud applications, such as Elasticsearch/Lucene and Cassandra, while they are executing on the Juno board. In some cases, certain application may be swapped for an ”easier to collect data” benchmark, such as CloudSuite [8].

3.1.2 Profiling and Instrumentation

In order to accomplish the development of a function-guided scheduling algorithm, techniques of profiling and instrumentation are used. Profiling is a form of dynamic analysis, which means that it happens at runtime, and allows the mensuration of many aspects of an application (e.g: number of function calls, number of access to memory and CPU cycles). Hence, profiling allows us to verify which functions of a certain application are the bottleneck in terms of computational load. This is important as the function with highest load will likely take more time executing when a heavy request arrives (e.g.: longer keyword length, in case of Elasticsearch). This particular analysis is carried out through profilers (e.g., YourKit Profiler, for Java applications) or OS tools (Linux's perf).

The activity of instrumenting an application allow us to alter its behavior - through the insertion of code (”code placement”) during its runtime - without recompiling it, and this process can be done even for black-box applications (e.g.: binary blob).

For applications written in Java, ASM [5] is the framework used in this work. The desired code placement is implemented is through the use of ASM's Visitor class. Because the complexity of the instrumented code can grow, there is a need for an efficient communication infrastructure. Thus, rather than inserting the adaptation code for thread migrations into the application itself, we instrument the application to send a signal to another piece of code (”the control logic”) that can carry out the actual thread scheduling. Figure 8 gives an overview of how the communication infrastructure with the scheduler works for managing a cloud application.

By instrumenting the cloud application, whenever a new request arrives, the request is designated to a previously created thread by the application. A fixed thread pool is created at deployment stage. A thread starts executing until it calls the desired instrumented function. When this happens, the thread is intercepted and sends a signal to a ring-buffer, which stores data such as the thread id and the timestamp. Another signal is sent when the thread leaves that function. Every certain amount of time, the scheduler checks if there's any event to be consumed at the ring-buffer and store those events in a new place while the ring-buffer moves its pointer, and then allocate the events (through calls to the operating system) according to the desired policy.

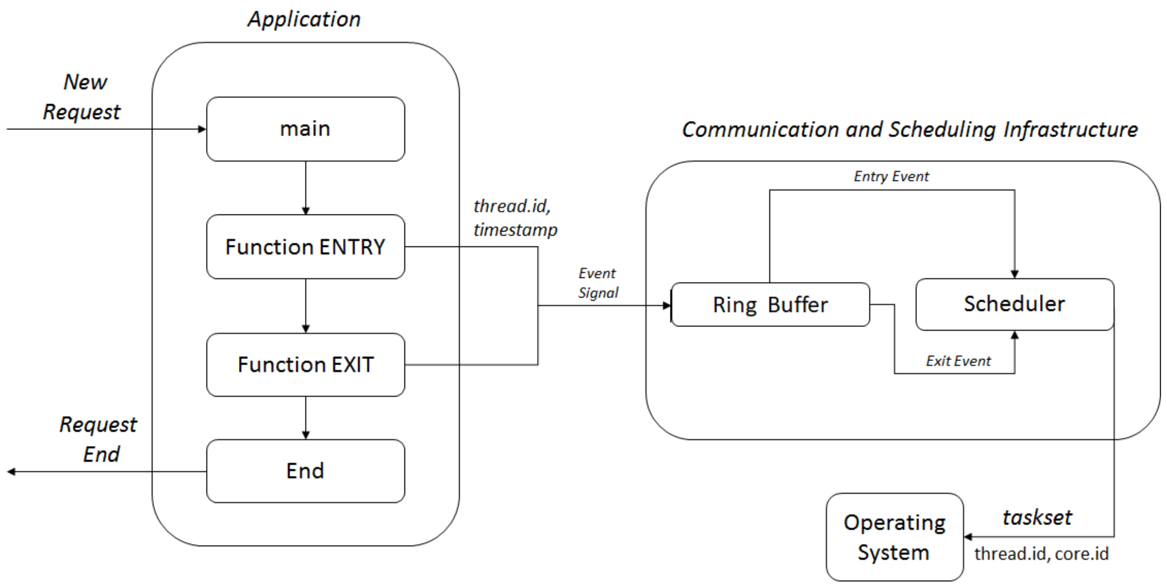

Figure 8: High-level overview of the communication infrastructure.

3.1.3 Scheduler Design

Leveraging the communications infrastructure previously described, we introduce the Hurry-up scheduler, a function-guided algorithm. We illustrate the concepts of the algorithm using the Elasticsearch benchmark. For this particular benchmark, we identified the following hot function *org.elasticsearch.Search.SearchService.executeQueryPhase()* to be monitored. This function, identified via profiling, was found to be responsible for about 80% of thread execution time on Elasticsearch.

A few assumptions were made in deriving the algorithm design:

- **Heavy vs Light requests**: There is notable difference in computing demands in the user requests (heavy and light ones).
- **Hot functions**: The application has particular functions experiencing high computing demands (hot functions), accounting for the increase of the tail latency.

We define the following terms used in the algorithm description:

- **Threshold**: To minimize unnecessary up and down migrations, a threshold should be defined when monitoring hot function execution. This is used to better distinguish between heavy and light requests.
- **Status**: When multiple requests compete for a limited number of cores, there is a need to manage in the task mapping decisions.

The proposal scheme is described in Algorithm 1. We define a n-sized matrix, where n is the number of cores. The matrix is used to store each thread identifier (generated by the JVM/OS), the thread status, and the thread's timestamp as the time the thread has entered in the hot function. A threshold variable is specified to help determine the criticality of a thread.

The thread status works as a priority system for thread management. They are: $NotInFunction$, meaning that the thread left the hot function, $BelowThreshold$ to characterize a thread running in the function, but is still below the criticality threshold, and $AboveThreshold$ to capture the thread running the hot function with time elapsed above the threshold. We index these status as 0, 1 and 2, in which 2 is the highest priority.

In order to avoid unnecessary core migrations, the threshold is used to more accurately capture threads considered ”heavy”; that is, longer execution times that may violate the latency deadline. Empirically, the threshold may be given by the mean of execution of certain amount of requests, multiplied by a factor (say, one or two standard deviations).

Algorithm 1 Hurry-up Scheduler Execution

```
1:  Initialize a $n$-core-sized matrix $M$
2:  Initialize table $T$ to store thread identifier, thread timestamp and thread status
3:  Set $n$ as the thread threshold
4:  loop scheduler every t miliseconds
5:      while there is a new EVENT in ring buffer do:                                  ▷ Atomic operation
6:          if event is ENTRY EVENT then
7:              put thread identifier into first empty space in matrix           ▷ Round-robin allocation
8:              set thread status as BelowThreshold
9:          if event is EXIT EVENT then
10:             set thread status as NotInFunction
11:     actualTimestamp ← system time
12:     for each element in matrix do                                   ▷ Check which requests are over threshold
13:         if actualTimestamp - thread timestamp $>$ threshold then
14:             set thread status as AboveThreshold
15:     for each LITTLE cores do                                 ▷ Check if any of little cores are over threshold
16:         if thread status == AboveThreshold then                          ▷ Actual request is over threshold
17:             for each BIG cores do                                 ▷ Check if there's an empty space on big
18:                 if any of threads has status == NotInFunction or BelowThreshold then          ▷ It's available
19:                     swap thread identifier with the found position
20:                 if all BIG status are AboveThreshold then               ▷ Not available, prioritize the oldest one
21:                     swap the oldest thread to BIG core
22:     set CPU affinities ← core identifier, thread identifier
```

Every t milliseconds, the scheduler will execute. First, it checks for new events in ring buffer and allocate in any free space at the matrix, in a round-robin fashion. As the thread just reached the hot function, its status is set to $BelowThreshold$. For implementation purposes, the ring buffer sends the same thread identifier whenever the thread leaves the hot function, so a duplicate id just sets the actual thread status to zero.

The actual time of the system is registered for comparison purposes. Finally, all threads are checked and if they are over the threshold, their status are set to "$AboveThreshold$". It's important to say that in the case of Juno board, the matrix ranges from id 0 to 5. Ids 0 and 1 represent the big core while the little cores are represented by ids 2 to 5. The final step of the algorithm ("set affinities") is just setting the thread identifiers stored on matrix to either big or little cores (line 22).

The intuition of the algorithm is that all threads running on little core will be evaluated and compared against the ones at big cores. At runtime, there are two criteria: first, if any thread on little core has status 2 ($AboveThreshold$) while at least one thread on the big core has status equal to 1 ($BelowThreshold$) or less ($NotInFunction$), there'll be a swap. However, if all big cores are occupied, the priority of execution on big goes to the oldest one.

Some experimental previous testing, although not quantified by this research, has shown that the overhead introduced by needlessly migrating threads from big core to little core and vice-versa is capable of slowing the service time when compared with its not-scheduled counterpart, so it's very important to keep the migrations to as few as possible.

Finally, it's important to say that the (i) the way hurry-up is currently developed only supports cases where the number of threads are equal to the number of cores; however, we believe that expanding support to different cases may be simple, and (ii) a first version of the scheduling algorithm is currently implemented and tested. A new set of experiments with this very initial implementation will be carried out.

4 Tentative Outline of the Thesis

Next, I briefly describe the tentative structure of the thesis. The thesis is mainly divided into three parts: the related work, the methodology and the experimentation.

Part 1 In the first part of the thesis (Sections 1-2), I outline the main concepts regarding quality of service and processor architecture and deeply review all the state-of-art scheduling algorithms, such as Pegasus, Rubik, Octopus-Man, Hipster and Adaptative Slow2Fast, also highlighting their pros and cons.
Part 2 In the second part of the thesis (Section 3), I present the proposed algorithm, its implications and scientific importance and also a methodology that fully attends the Thesis Statement.
Part 3 In the final part of the thesis (Sections 4-5), I implement the proposed algorithm and collect results. I also analyze those results and compare with the existent scheduling algorithms.

This new approach is to be implemented into Elasticsearch and Cassandra, under Java platform, and also tested under simulated high-load cases. The techniques for implementation includes profiling and instrumentation. Finally, obtained results are compared with the ones existent in literature.

5 Timeline to Completion

September - November: Evaluate and improve the scheduling algorithm. Collect results. Analysis of preliminary results. Perform possible methodology adjusts in the thesis proposal.
December - April: Evaluate the designed algorithm for other applications or benchmarks (Cassandra, CloudSuite). Collect final results. Write thesis.
May 7, 2019: Submit thesis to committee members for feedback.
June 3, 2019: Thesis defense.
June 24, 2019: Final thesis submission.

This is a tentative time table for completion of the thesis. Adjusts may be done according to the progress of the research.

References

[1] Creating and running a http workload using fhb and a workload configuration file. Available at: http://faban.org/1.2/docs/tutorials/fhbconfig.html.

[2] *Apache Lucene 4*, Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval, 2012.

[3] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, second edition, 2013.

[4] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998. ISSN 0169-7552. doi: 10.1016/S0169-7552(98)00110-X. URL `http://dx.doi.org/10.1016/S0169-7552(98)00110-X`.

[5] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[6] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Designs*. Pearson, fifth edition, 2012.

[7] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[8] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[9] Peter Greenhalgh. Big.little processing with arm cortex - a15 and cortex-a7. White paper, ARM, September 2011.

[10] Md Haque, Yuxiong He, Sameh Elnikety, Thu Nguyen, Ricardo Bianchini, and Kathryn McKinley. Exploiting heterogeneity for tail latency and energy efficiency. *MICRO-50*, pages 625–638, October 2017.

[11] James C. Hoe. 18-447-s18-l01-s1, 2018.

[12] Urs Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 2010.

[13] Harshad Kasture, Davide Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, December 2015.

[14] Avinash Lakshman and Prashant and Malik. Cassandra - a decentralized structured storage system. In *Workshop on Large-Scale Distributed Systems and Middleware*, 2009.

[15] David Lo, Liqun Cheng, Rama Govindaraju, Luis Andre Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *IEEE/ACM 41st International Symposium on Computer Architecture (ISCA)*. IEEE, June 2014.

[16] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th ACM International Symposium on Computer Architecture*, 2011.

[17] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. Hipster: Hybrid task manager for latency-critical cloud workloads. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, February 2017.

[18] Vinicius Petrucci, Michael Laurenzano, Doherty Doherty, Yunqi Zhang, Daniel Mossé, Jason Mars, and Lingjia Tang. Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers. In *High Performance Computer Architecture (HPCA)*. IEEE, February 2015.

[19] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search, June 2009. Research Presentation.

[20] Arman Shehabi, Sarah Smith, Dale Sartor, Rich Brown, Magnus Herrlin, Jonathan Koomey, Nathaniel Horner, Eric Masanet, Ines Azevedo, and William Lintner. United states data center energy usage report. Technical report, Lawrence Berkeley National Laboratory, June 2016.

[21] Tse-Yu Yeh, Deborah Marr, and Yale Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Proceedings of the 7th ACMInternational Conference on Supercomputing*, pages 67–76, July 1993.