Profile-guided Frequency Scaling for Latency-Critical Search Workloads

Daniel Araújo de Medeiros*, Denilson das Mercês Amorim*, Vinicius Petrucci*†
*Universidade Federal da Bahia, Brazil
†University of Pittsburgh, USA

***Abstract*—Dynamic frequency scaling is a technique to reduce power consumption in computer systems. However, this technique poses challenges when adopted in latency-critical applications. 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.**

This work proposes a finer-grained dynamic frequency scaling approach for multi-core processors that leverages information about the computational intensity of certain functions in a latency-critical web search application. First, our approach profiles the running application to identify hot functions for typical workloads. Next, a run-time scheme is devised to adapt the individual core frequency whenever a compute-intensive thread enters or exits a hot function. We implemented and evaluated our proposal in a real multi-core system. We observed energy consumption savings up to 28% when compared to the recent Linux's Ondemand frequency scaling governor, while attaining acceptable levels of tail latency constraints.

***Keywords*-frequency scaling, energy consumption, tail latency, software instrumentation**

I. INTRODUCTION

As modern applications are moving from desktop clients to smaller mobile devices, the processing and storage now needs to rely on powerful servers in the cloud once locally present [1]. 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 2 seconds in returning web search results may impact revenue by over 4% per user [2]. For large cloud companies, this turns out to be a strong negative impact on their business. 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 [3].

Tail latency implies that in cloud systems where each server typically answers quickly, but experiencing high 99-percent latency, all requests will be affected if the request is handled by a single slow server since the results from multiple requests are aggregated by a root server before responding to the user. When scaling the application to use more servers, the tail latency influence is amplified causing the overall system to degrade the user experience [3].

While it is possible to guarantee a high quality of service through the acquisition of better hardware, this may be very costly and may be economically infeasible as the application scales. To improve power efficiency, modern multi-core systems are designed to explore a technique named Dynamic Frequency Scaling (DFS), in which the frequency of operation of a certain core may increase or decrease according to the application's demand. Given that power consumption is linearly proportional to core frequency, a lower frequency will result in less power consumption. Weiser et al. [4] argued that reducing only the clock speed does not reduce the amount of instructions needed to be finished per Joule, once the system must run for longer to perform the same amount of work.

An implementation of DFS is within the Linux's Ondemand governor [5], which attempts to minimize energy consumption by changing the CPU speed according to the actual load based on CPU utilization thresholds. However, traditional techniques like this do not take into account the application behavior or the tail latency constraints of the running application. Several approaches to this problem were already developed in the literature, exploring different concepts and techniques - in particular, scheduling schemes such as statistical prediction [6], feedback-control state machines [8], [15], [20], and reinforcement learning [7], [14] for heterogeneous multi-core systems. These works are coarse-grained in the sense that it maps the entire application onto the best CPU configuration, and usually require external run-time systems that may incur some system overhead. They are also not designed for taking into account particular characteristics found in 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.

This work adopts a new approach for latency-critical Java applications that combines profiling to identify compute-intensive functions with code instrumentation for thread monitoring and adaptation. According to data from Alibaba's datacenter, more than 90% of latency-critical cloud services are written and deployed as Java applications [17]. Our approach profiles the running application to identify hot functions and uses a run-time scheme to adapt the individual core frequency based on the activation of the hot functions. We implemented and evaluated our proposal in a real multi-core system. We observed energy consumption savings up to 28% when compared to the recent Linux's Ondemand

frequency scaling governor (kernel 5.3) while attaining acceptable levels of tail latency constraints. It is worth noticing that Ondemand is well crafted and implemented at the kernel-level.

In Section II, we provide an overview of search workloads and the scaling governors available on Linux systems. In Section III, we present empirical observations to motivate our proposal for a new frequency scaling technique. Section IV presents our solution. Section V shows our experimental results. Related Works are present in Section VI. Finally, conclusions are presented in Section VII.

II. Background

A. Latency-Critical Search Workloads

Brin and Page [9] present a seminal overview of how a web search service is structured. A search engine is composed of three major independent applications: crawling, indexing, and scoring. Crawling is related to the fetching of the contents present in a website. This is usually done by several automated bots (crawlers) which recursively scan a webpage and extract all links from it. Indexing is the act of parsing and storing those crawled pages into an index for later retrieval. Finally, scoring is related to returning the most relevant results, based on previously selected attributes, for the user. Those results are ranked and later returned. We focus on web search since it is one of the most important latency-critical applications. For complex keywords and/or a large number of users, the scoring procedure might take a longer time, affecting user experience.

B. Elasticsearch

Elasticsearch is an open-source search engine built atop of Apache Lucene [10], a Java library that executes the indexing and querying/scoring functions. Like Lucene, Elasticsearch is written in Java and is mainly responsible for serving the search results through its Application Programming Interface (API) while also allowing Lucene to scale among clusters. This means that Elasticsearch acts as a front-end application, responsible for cluster management, thread pool, queues, and monitoring APIs.

As a distributed application, multiple nodes or machines comprise an Elastichsearch cluster, which is the biggest unit in this application. This cluster handles all the non-Lucene tasks, such as query distribution among shards, creation, and replication of shards, and maintenance. Each node contains one or more shards, where each holds a search index. A request query sent by the user goes through all the shards. It also has the size property, which is directly associated with the number of keywords it has.

In our work, we indexed the Wikipedia dataset (see Section V for more information) into Elasticsearch to respond to search queries.

C. Query Generation

FABAN [11] is used as a tool for query generation. Similar to the usage in CloudSuite [12], the load generator follows a Zipfian distribution, where 95% of the queries will have a keyword length between 1 to 8 and the remaining 5% will have its length between 9 to 18. The upper plot from Figure 1 shows a histogram for the requests generated by FABAN in a 30-minute run on the performance governor. The bottom plot shows the cumulative distributive function (CDF) of the service time from the same run. As discussed in Section III, the number of keywords for a query has a great impact on the service time, with higher keyword sizes requiring more overall service time.

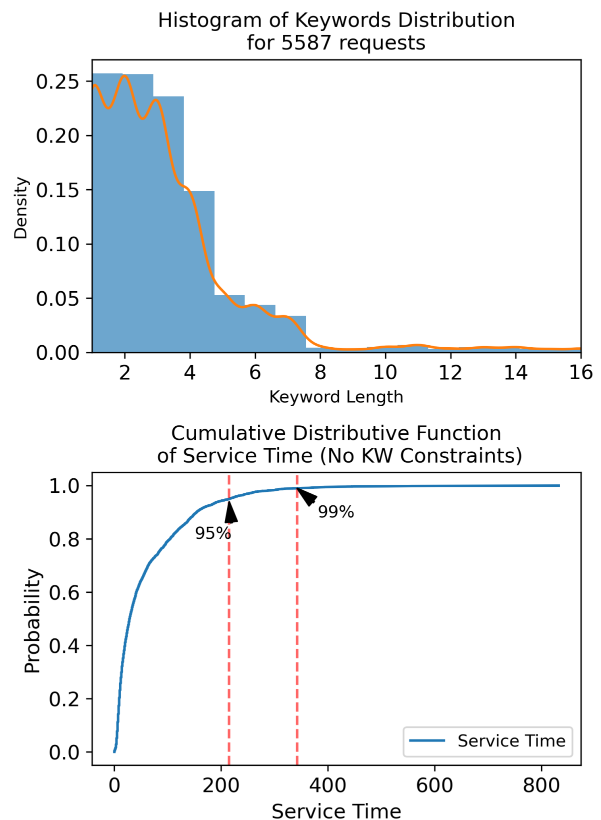

Figure 1. Statistics from a 30-minute Elasticsearch run with FABAN. The upper plot shows the histogram of keyword length, ranging from 1 to 16. The bottom plot shows the CDF of the service times.

In addition to the keyword length, whenever a query reaches a node on the Elasticsearch cluster, the query is promptly distributed among a pool of search threads. Those threads are created exclusively for searching documents within shards and there is at least one thread per shard. Each node will perform parallel processing and act independently on every shard, and each shard will have its own processing time. In this case, the user-perceived tail latency will be determined by the slowest shard. Each search thread scores

its results and returns to the compute node, which returns to the root node of the cluster that sorts the obtained results and replies to the user.

D. Linux CPUFreq

Frequency scaling on Linux is implemented through an infrastructure named CPUFreq, which uses frequencies for identifying operating performance points (known as 'P-States') of processors. Scaling drivers provide scaling governors with information on the available P-states and access platform-specific hardware interfaces to change CPU P-states as requested by scaling governors. The generic Advanced Configuration and Power Interface (ACPI) CPUFreq driver provides at least four scaling policies (governors) by default: `performance`, `powersave`, `Ondemand` [5], and `userspace`. The Ondemand governor has been Linux's default since kernel 3.4.

The `performance` governor sets all cores with the highest available frequency, while the `powersave` governor sets them at the lowest frequency. `Ondemand` measures the CPU usage and when this usage is over a certain threshold, the frequency goes to the maximum and starts decaying gradually after a certain time, while it can go up again if necessary. Finally, `userspace` is a fixed frequency chosen by the user.

E. Java Run-time Instrumentation

Tracing tools like Linux's BPF Compiler Collection (BCC) and DTrace relies on JVM's probes generated on run-time through the use of certain flags (e.g., `-XX:+ExtendedDTraceProbes`). Probes are generated at any thread event (e.g., entry/exit on monitors, waits, etc) without any filtering. As per Java's documentation [19], these probes are associated to severe degradation in performance.

Since Elasticsearch is written in Java as many enterprise cloud applications, we explore instrumenting the Java Virtual Machine (JVM) by designing and implementing a JVM Tool Interface (JVMTI) agent. An agent is capable of tracing functions' states that can help us make frequency scaling decisions. The JVMTI provides an API for libraries, which are written in C/C++ through Java Native Interface to be loaded during the initialization of the JVM, and communicates directly to both the kernel and the VM itself. The JVMTI is defined into a JVM specification and part of the Virtual Machine's own core implementation. Section V-D discusses the overhead introduced by the usage of this agent.

III. Empirical Observations

Before introducing our solution, we present a set of empirical observations used to guide our design decisions. Details regarding software and server configurations used in our experiments are described in detail in Section V. All experiments in this work were repeated at least three times. The standard deviation, although present, can be small and not be easily seen in the plots.

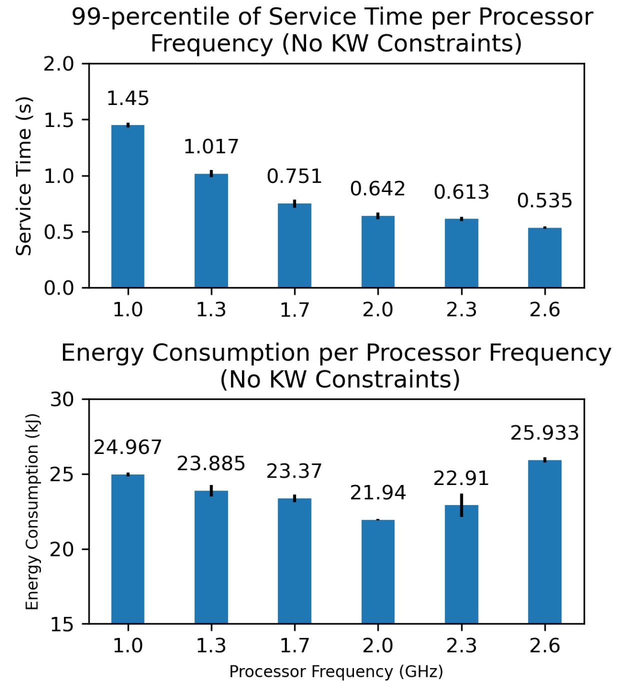

Figure 2. Search requests are executed on multiple frequencies without any constraints in keyword length. The 99-percentile service time for each frequency is shown in the upper plot, while energy consumption is shown in the bottom plot.

Observation 1: *Given multiple search requests of different keyword sizes, 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 means that the overall service time for a web search can be influenced by the CPU operating frequencies. We can show 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 filtering on the keyword length. The results are shown in Figure 2 (upper plot). We can see that the service time of the requests running at 2.6 GHz (the maximum available frequency) is the fastest and gets slower as the core frequency decreases. In particular, the difference in service time between the frequencies of 1.0 GHz and 2.6 GHz is nearly three-fold.

In this and the following experiments, we group queries with 3-4 keywords and name them "light queries" (or "low keyword count") and for the ones ranging from 12 to 18 onward, we name them as "heavy queries" (or "high keyword count"). The distribution of light vs heavy queries was chosen based on the number of queries generated by the load generator (see Section II-C).

Observation 2: *Given two different types of requests considered "light" (3-4 keywords) and "heavy" (12-18*

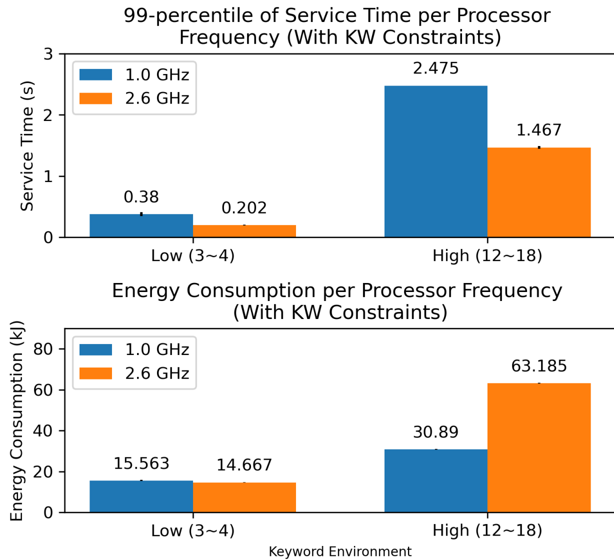

Figure 3. Elasticsearch running with low vs high keyword inputs. The upper plot shows the 99-percentile service time when the CPU is set to 1.0 GHz vs 2.6 GHz. The lower plot shows the energy consumption on each processor frequency.

keywords), the "heavy" requests will take longer to finish.

This experiment is to show 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 and 2.6 GHz, the minimum and maximum available in the processor. Figure 3 (upper plot) 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.

Each query in Elasticsearch 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 common sense that higher frequencies consume more energy, we wanted to see if there is any meaningful difference in energy consumption between light and heavy requests. The results can be seen in Figure 3 (bottom plot).

In fact, heavy requests consume more energy than their light counterpart at the same frequency. Heavy requests make the energy consumption over fourfold higher at the 2.6 GHz frequency. 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, since it is doing the same work but the system must run longer.

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

We show this by running only light requests while varying the available frequencies. Figure 2 (bottom plot) shows that 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 while also providing similar service time results when compared to nearby operating points (1.7 and 2.3 GHz); hence, in this case, the frequency 2.0 GHz has the best energy consumption.

IV. Frequency Scaling Solution

Our solution works by identifying that individual requests can demand distinct CPU processing time, thus the frequency can be adjusted to match the request's demand. The key idea is to dynamically track the execution stage of the most compute-intensive threads (i.e., *search threads* in Elasticsearch) and to adjust the associated CPU frequency based on the stage information. The execution stage is characterized by two information sources: (1) whether or not the thread is executing a particular hot function, and (2) for how long it has been running in the hot function.

The main elements of our approach are detailed as follows.

A. Designing the JVMTI Agent

An overview of our solution is presented in Figure 4. We design a JVMTI agent, called "Hurry-up", which consists of two parts. First, there is an event generator for capturing function entry and exit calls. Second, there is the frequency governor that consumes those events and decides the CPU frequency to run the thread associated with those particular events. The Hurry-up Agent runs as a separate process from Elasticsearch at the JVM.

Whenever a user-generated request arrives at Elasticsearch, the request is split among the existing shards. Elasticsearch uses parallel threads to process that particular request across all the shards. When a search thread enters or exits the hot function, an event is sent to the JVMTI Agent (Event Generator) whenever it reaches either the entry or exit point. After all the shards' results are scored and sorted, the request returns to the user. The *Event Generator* is responsible for intercepting the hot-function entry or exit events and pushing them into a global queue.

During the search process, our frequency scaling solution works by reacting to hot function entry/exit events that are intercepted at run-time. A flowchart of the frequency scheduler logic is shown in Figure 5. Recall that every thread entry/exit to/from the hot function generates an event that is pushed into the Event Queue that is consumed by the scheduler. Based on those events, the *Scheduler* is responsible for increasing or decreasing the CPU frequency running each thread based on the executing stage of these search threads. Each event has information about what core each thread is running at, a timestamp for detecting if this

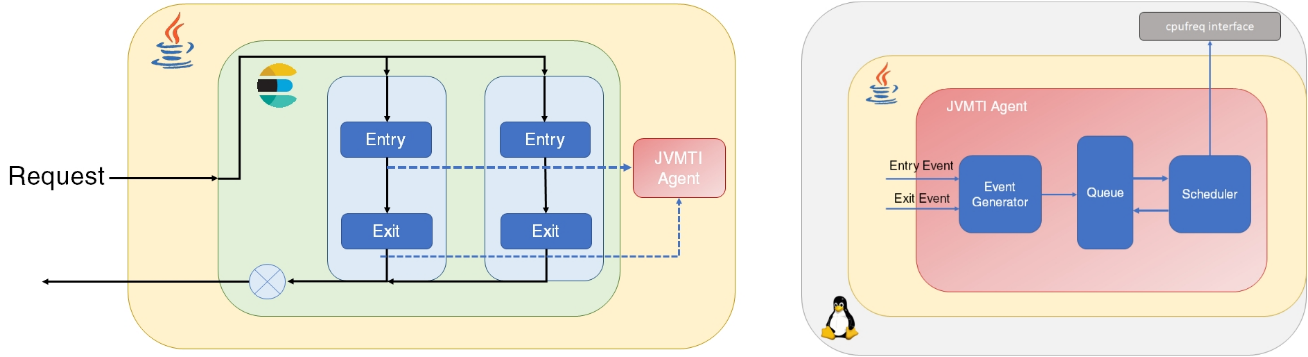

Figure 4. Overview of our solution. Every thread entry and exit event on a hot function is intercepted at run-time by the JVMTI agent (shown on the left). In the JVMTI agent, shown on the right-hand side figure, there is an *Event Generator*, which pushes the entry and exit data to the *Event Queue*; the *Frequency Scheduler* consumes the events from this queue and performs the CPU frequency changes using the `cpufreq` Linux interface

thread is over a particular time threshold, and status (e.g. either an entry or leave event).

The scheduler can be tuned by changing the following parameters: a `threshold` time to classify a thread as critical and to speed it up, and a `sleep` time to block waiting for new events to come in the queue that helps minimizing the overhead of calling the scheduler code too often. In Section V-F, we show the results of a sensitivity analysis to understand the impact of tweaking these parameters.

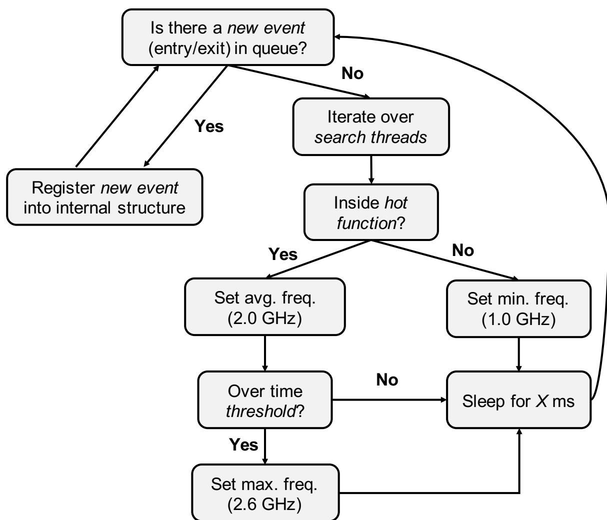

Figure 5. Flowchart describing the frequency scheduler logic.

B. Identifying hot functions

To track the execution stage, our approach needs to identify the application's hot function. This is done via profiling during deployment time while running the application with previous representative input traces.

We generate a Flame Graph of Elasticsearch after running it using the Linux `perf` tool (see Figure 6). Based on this plot, we found that the function

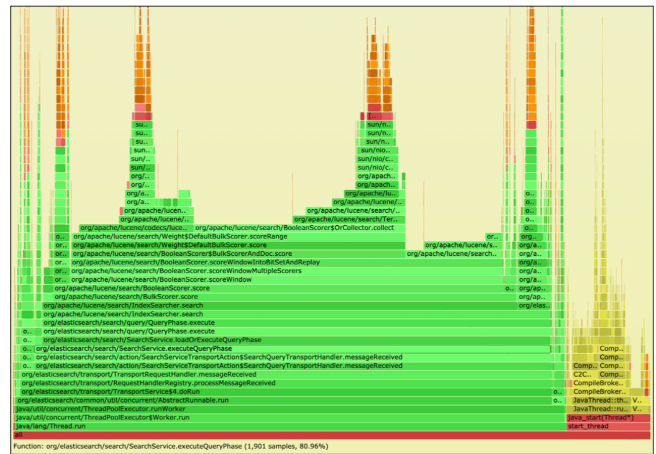

Figure 6. Flame Graph of Elasticsearch is used to accurately identify a hot function by sampling the function call stacks. The *x-axis* shows the stack frames and the *y-axis* shows the depth of each stack frame. Each stacked box in the figure represents a function call, where the width of a particular box represents the frequency at which that function was found in the sampled stack frames.

`org/elasticsearch/search/SearchService.executeQueryPhase` dominated about 81% of the search process. We selected this particular function from the call stack since it is the top function in the stack from the search Java package. Elasticsearch may have other hot functions not relevant to our work, mainly used for other functionalities such as indexing, and cluster management.

V. Evaluation

In this section, we describe in detail how our solution was evaluated.

A. Experimental Setup

We conducted our experiments on a "baremetal" server instance, provided by the Chameleon Cloud service [13]. The server consists of an Intel Xeon Gold

6126 (Skylake architecture), with 196GB of DRAM and a 220GB SSD disk. We run Ubuntu 19.10 (Linux kernel 5.3) and Elasticsearch version 6.5.4. For the search index database, we used the Wikipedia dump version `enwiki-latest-pages-articles` downloaded in June, 2020.

The CPU is DFS-capable, allowing the operating frequency to be changed on a per-core basis. The operating frequency can range between 1.0 and 2.6 GHz for each individual core. Moreover, the CPU has 24 physical cores equally divided between two sockets. Intel's Hyperthreading technology was turned off, so we only consider the physical core count.

To isolate the network effects in the shared experimental platform, we configured socket 0 to run the server-side applications (Elasticsearch) and socket 1 to execute the load generator (FABAN). The energy measurement was done through Intel's Running Average Power Limit (RAPL) interface. We measured the energy consumption counter at the beginning of the experiment and at the end; the difference between those two values is considered the energy consumed for that particular experiment.

After its initialization, Elasticsearch spans over 70 Java threads in total. Each thread has a specific function, such as searching, indexing, system interaction and cluster management. We configured Elasticsearch to spawn 12 search threads (same number of cores on the server-side CPU socket) so that there is one thread per core; this was done to avoid possible core-sharing interference between search threads.

The client (FABAN [11]) runs on a separate CPU socket and is responsible to generate query requests with randomized keywords, ranging between 1 to 18 keywords as explained in Section II-B. The client is also instrumented to report the user-perceived response time distribution.

B. Establishing the Baseline

As explained in Section II-D, the `Performance` governor sets the cores at the highest frequency, the `Powersave` governor sets to the lowest available frequency, and the `Ondemand` sets the frequency dynamically depending on the CPU utilization.

Figure 7 shows an experiment comparing the Linux governors running Elasticsearch. For this experiment, we set the QoS target to 99-percentile at 1 second to accommodate the worst-case scenario with high length keywords. The `powersave` governor violates the deadline when Elasticsearch receives queries with high keyword size and almost violates the deadline with the low keyword length; considering the standard deviation, the deadline is exactly at 1 second. While `powersave` reduces the energy consumption for high keyword lengths, it uses more energy than `performance` or `Ondemand` for low keyword lengths.

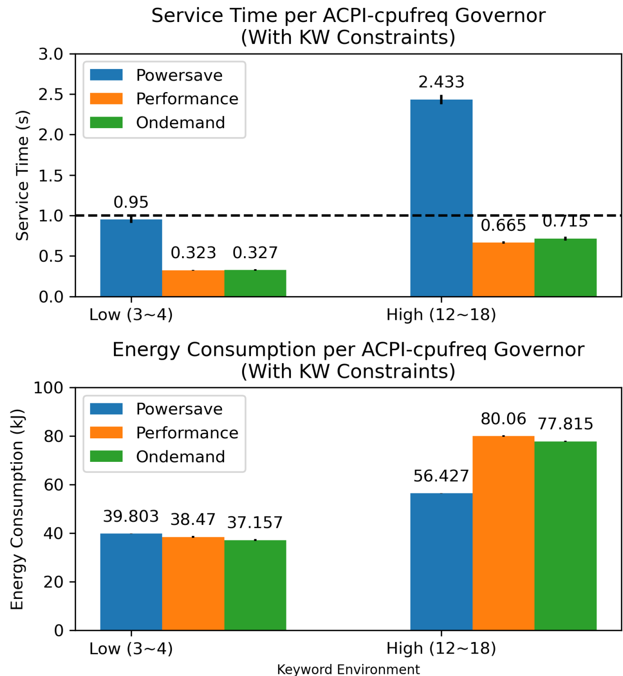

Figure 7. Linux Governors: 99-percentile latency (upper plot) and energy consumption (bottom plot). The latency target used for service time is 1 second.

As expected, the `Performance` governor performs well on meeting the latency requirement (1 second) but falls behind `Ondemand` on energy consumption. Since both governors can meet the imposed deadline, the `Ondemand` was chosen as the baseline for the rest of this section because of its energy efficiency gains compared to `Performance`.

C. Comparing with Ondemand

The following experiment is a direct implementation of the flowchart shown in Figure 5 described in Section IV-A. We would like to see how our Hurry-up implementation performed against `Ondemand`. There are two parameters for our Hurry-Up governor, one known as "time threshold" to determine when the core frequency needs to move from average frequency (2.0 GHz) to maximum frequency (2.6 GHz), and "sampling time" used for collecting the call stacks and generating the hot-function entry/exit events. For this experiment, we intuitively set those parameters to $350ms$ and $10ms$ respectively, which are low enough to trigger most adaptations.

The load generation process consisted of 20 minutes runs of continuous search queries 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 goal was to analyze the service time behavior of each governor policy considering a single request at a time.

We performed three runs (of 20 minutes each) for each combination of keyword length and governor policy and reported the standard deviation. Note that the "low" keyword length is fixed at the 3-4 range in order to generate at least a minimum load in the system, while the "mix" setting goes between 1 to 18 keywords (see Figure 1 for more details).

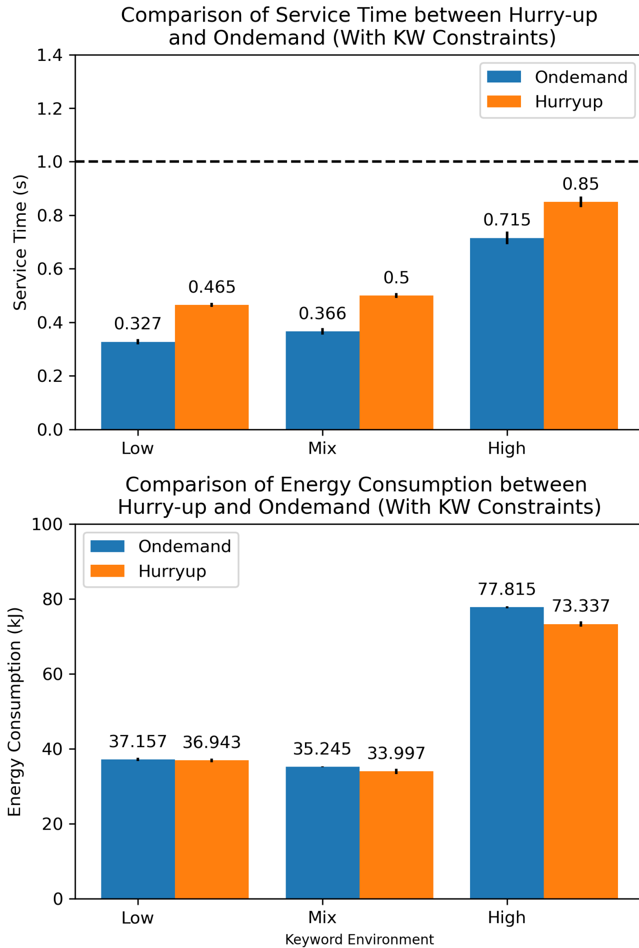

Figure 8. An experiment comparing Ondemand against Hurry-up on Service Time (upper plot) and Energy Consumption (bottom plot).

The result of Hurry-up when compared with `Ondemand` is shown in Figure 8. While both approaches satisfy the deadline, our Hurry-up policy has energy-saving that amounts up to 6% in comparison to the `Ondemand` governor, specially on high-load keywords. This is because both `Ondemand` and Hurry-up concentrate most of their idle time at 1.0 GHz to minimize energy consumption when not performing actual search ranking. Figure 9 shows the distribution of time spent on relevant frequencies for each policy. Hurry-up uses the average (2.0 GHz) frequency for most of the search execution and promotes only the heavy queries to run at the maximum frequency (2.6 GHz); these queries are 5% of the Zipfian distribution (Section II-C).

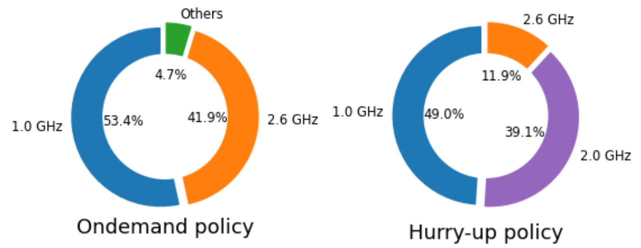

Figure 9. Distribution of time spent on a particular CPU frequency when using Ondemand and the initial Hurry-up implementation. The label "others" refers to all other frequencies between 1.2 and 2.4 GHz.

Whenever there is a spike in CPU usage, e.g., by sequentially processing requests with different keyword lengths, `Ondemand` sets the CPU frequency to maximum (2.6 GHz), which is not the most efficient in the average cases since typically not all requests would require this speed. As will be seen later in Section V-G, the difference between Hurry-up and Ondemand becomes more prominent with a higher load.

D. Overhead Analysis

We evaluate the overhead of our Hurry-Up implementation with a CPU scaling governor embedded into the JVM/Elasticsearch. To perform the analysis, we used the original code base but without any frequency scaling interaction and compared it against Hurry-up with a fixed frequency setting (through the `userspace` governor) without calling the scheduler logic.

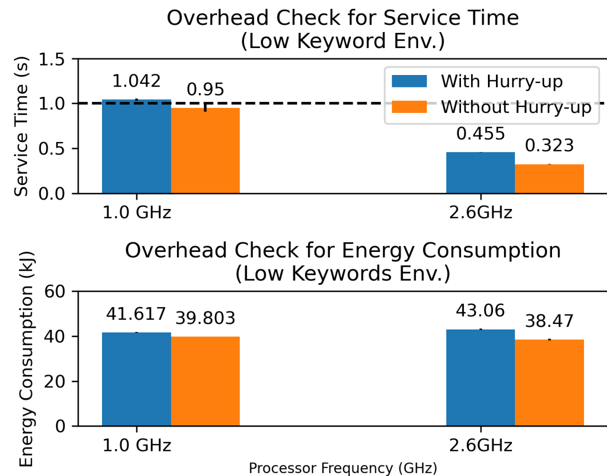

Figure 10. Overhead Comparison of the Implemented Agent on (upper) Service Time and (bottom) Energy Consumption.

Figure 10 illustrates the measured overhead for each case. 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 energy overhead from the scheduler is 5% for 1.0 GHz and 11% for the other case. We use low-keyword queries to minimize potential additional stress that might be caused by high-keyword queries, thus amplifying the signal-to-noise ratio. In sum, there's a moderate to severe overhead with the agent introduction, but this overhead is mitigated by the benefits our solution produces.

This overhead can be explained by the JVMTI agent interrupting the search thread for a very brief moment to collect call stack data to capture hot-function entry or exit events. Also, the agent itself generates one more thread that performs bookkeeping operations, such as manipulating the event queue. In Section V-F, we show the effects of adjusting the sampling time parameter.

E. Analysis of C-states

The CPU has different modes of operations collectively called C-States. The Skylake Architecture has four C-States, namely C0, C1, C1E, and C6, where C0 is the fully active and C6 is the lowest energy state (including 0 Voltage). Analyzing the processor's C-States is necessary to understand the behavior of both Ondemand governor and our Hurry-up Implementation.

The C-States can be controlled through the */sys/fs* interface and it has a per-core per-state granularity. First, we compare how our implementation and the Ondemand governor performs in service time and energy consumption when only the C0 state is available and when all states are available. Table I shows the results for the experiment of varying the C-states of the CPU.

Table I
SERVICE TIME AND ENERGY CONSUMPTION PER C-STATE (NO KW CONSTRAINT)

Service Time (ms)	Ond. Low Env.	Ond. High Env.	Hurry-up Low Env.	Hurry-up High Env.
Only C0	0.418	0.837	0.458	0.9
All States	0.357	0.689	0.459	0.788
Energy Consump. (kJ)				
Only C0	30.915	40.763	29.27	36.660
All States	20.525	44.310	18.705	35.635

While we could not make a definitive correlation between the C-states and the service time, we noticed that the energy consumption for "Only-C0" mode 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. Table II shows the time spent (in clock-tick units) per state when running with all states enabled.

For low keywords, both Hurry-up policy and Ondemand governor spent much longer time at the C6 state, meaning that most of the idle time was at the lowest energy level.

Table II
NUMBER OF CALLS PER C-STATE

	Ond. Low Env.	Ond. High Env.	Hurry-up Low Env.	Hurry-up High Env.
C0	45.034	22.279	37.836	43.234
C1	119.163	52.560	31.213	38.914
C1E	141.273	137.206	206.654	416.103
C6	573 x 10^6	417 x 10^6	547 x 10^6	354 x 10^6

The difference in consumption can be explained by Hurry-up executing low queries at an average level of 2.0 GHz, while Ondemand operates at 2.6 GHz. In the long run, a higher difference in consumption may be seen and even more favorable for the Hurry-up implementation.

For higher keywords, Hurry-up spends more time on activity (C0 State), but at a lower frequency. As shown in more detail in Section III, this lower frequency translates to a lower energy consumption even at the cost of a small delay on service time.

F. Sensitivity Analysis

Our implementation of the Hurry-up policy allows us to choose two parameters: the "sleep" time, which is responsible for the interval time of the scheduler, and the "threshold", the time when we consider a request as a heavy one and decide that we need to change frequencies.

Tables III and IV show the results of this experiment on service time and energy consumption. To isolate the effects of varying the threshold, we fixed the sleep time at 10 ms and varied the threshold for the values of 150 ms, 300 ms, 450 ms, and 600 ms. We notice that energy consumption tends to decrease as the threshold increases, and the service time follows the opposite trend.

Table III
SENSITIVITY ANALYSIS: SERVICE TIME

Threshold (ms)	150 ms	300 ms	450 ms	600 ms	
Low Env.	0.373	0.438	0.525	0.453	
High Env.	0.705	0.8	0.9	0.9	
Sleep (ms)	5 ms	10 ms	20 ms	50 ms	100 ms
Low Env.	0.455	0.438	0.408	0.518	0.518
High Env.	0.85	0.8	0.825	0.825	0.85

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 value increases, the time a thread stays at 2.0 GHz also increases - thus a decrease in energy consumption is already expected. The chosen parameter, in this case, was 450 ms. Although a 600 ms threshold might appear to work better, 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 sleep parameter, we fixed the threshold at 300 milliseconds and tried values of 5, 10, 20, 50, and 100 ms. Overall, there was a small service time overhead when using

very small or very large values. In the end, we chose 50 ms as the default parameter.

Table IV
SENSITIVITY ANALYSIS: ENERGY CONSUMPTION

Threshold (kJ)	150 ms	300 ms	450 ms	600 ms	
Low Env.	20.85	19.41	18.48	19.32	
High Env.	39.3	37.07	34.63	32.34	
Sleep (kJ)	5 ms	10 ms	20 ms	50 ms	100 ms
Low Env.	19.66	19.41	19.65	18.1	18.09
High Env.	36.46	37.07	36.44	35.69	35.8

G. Results with tuned parameters at higher load

Based on the sensitivity analysis, we tuned our implementation with the parameters of $450ms$ for the time threshold and $50ms$ for the sleep time. We kept the load at a minimum to understand the impact on the service time. Results can be seen in Figure 11. While both schemes meet the deadline, Hurry-up has an energy consumption of about 17% less than the one by Ondemand at the high keyword length environment. The Zipfian distribution, used by FABAN and CloudSuite, favors very low keyword lengths (e.g., 1 and 2 keywords happens more than 3 and 4), hence why the "Mix" has a lower service time than the "Low" (3-4 keywords in size). For this experiment, the difference between Ondemand and Hurry-up in energy consumption favors Hurry-up in about 2%.

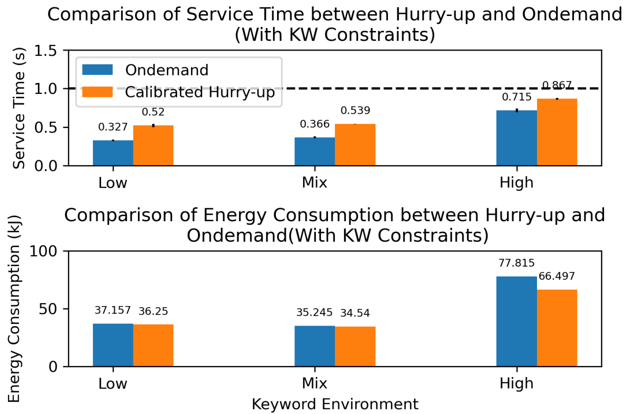

Figure 11. Comparison of the results at *higher load*, showing service time (upper plot) and energy consumption (bottom plot) for each frequency scaling scheme (Ondemand and Hurry-up).

To scale the application to respond to more clients and queries, we increased the request-per-second rate three-fold. At such a high load, both Ondemand and Hurry-Up violate the target deadline when all queries have high keyword length, hence why they are not shown here.

Figure 13 shows the obtained results for a higher load. For low and mix keyword lengths, when both schemes meet the specified deadline, the difference in energy usage between

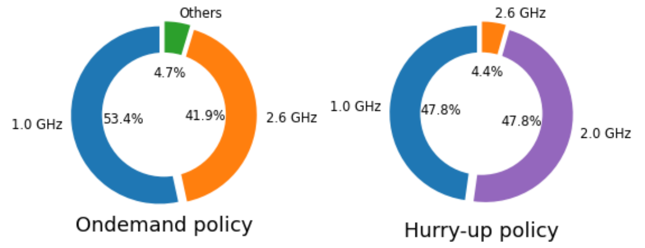

Figure 12. Distribution of time spent on a particular CPU frequency when using Ondemand and the calibrated Hurry-up implementation. The label "others" refers to all other frequencies between 1.2 and 2.4 GHz.

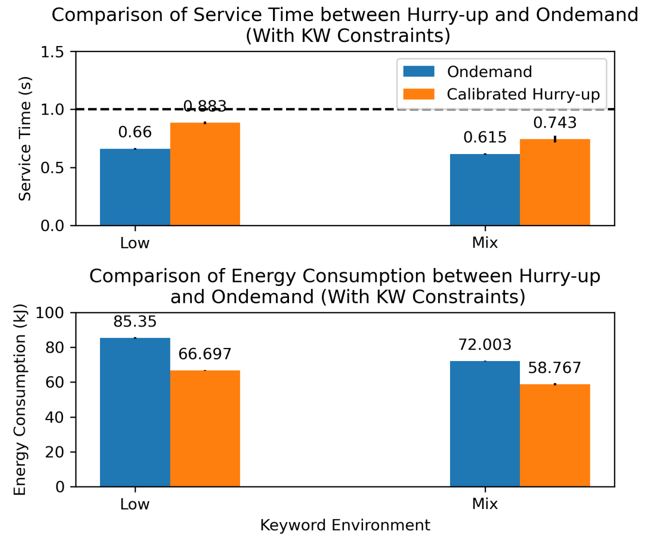

Figure 13. Comparison of the results for a 3x base load. (Upper) Service time and (bottom) energy consumption for each governor.

Ondemand and Hurry-up was noticeably more pronounced. In particular, the energy savings of Hurry-up increased to 22% for the mix load input and 28% for the low keyword length. Due to the CPU at a higher load, Ondemand stays most of the time at the highest available frequency (2.6 GHz), which hinders the energy savings. Hurry-up can respond to all requests within the deadline adjusting the CPU frequency more carefully between average to maximum, thus it can save more energy. Figure 12 shows the time spent per frequency illustrating this point.

Based on the results presented, our major findings are as follows. In search workloads, we observed variability in service times depending on the CPU frequency chosen and, thus, an energy usage difference between light and heavy requests. The best frequency for running lighter and mid requests was found to be a midpoint (2.0 GHz in our case). As for heavy requests, since they are prominent to lose the imposed deadline, it is indeed necessary to run them at the highest available frequency. Regarding the C-states

capability, when the CPU is not able to return to the C6 state for idling, the lowest energy consumption will be at the lowest available frequency.

VI. Related Work

Recent literature shows a myriad of related works on frequency scaling of applications with dynamic usage of computational resources. While this work is original in its methodology using profile-driven CPU scaling on search workloads, the ones that have some degree of similarity to this paper are Pegasus [18], Rubik [6] and Hipster [7].

Both Pegasus and Rubik rely on dynamic-variable frequency-scaling processors intending to optimize power consumption. Pegasus' paper introduces the iso-latency policy, which monitors point-to-point task latency and modifies energy configurations of all servers so they reach the deadline accordingly. The Running Average Power Limit (RAPL) technology, available only in Intel processors, is used for this purpose and allows user-level definitions of a power limit threshold for the CPU.

Pegasus explores RAPL to implement its iso-latency policy. In essence, whenever data reveals that there's room for latency, Pegasus lowers the allowed power level; else, when the latency is nearing the deadline, Pegasus increases the allowed power level. The main issue with Pegasus' approach is the dependence on Intel-only technology, which is not effective for other system architectures (e.g., ARM, PowerPC) or other chip manufacturers (e.g., AMD). Rubik uses a similar data analysis approach but without using RAPL.

The central idea in Rubik is the development of a statistical model that uses service time data collected on execution time to overcome the computational uncertainties of each request. This allows the model to predict which is the lowest frequency that will not violate the deadline - and each time a new request arrives, a new prediction is made and frequency is changed accordingly.

Hipster's approach uses Reinforcement Learning to dynamically schedule tasks in heterogeneous cores at the same time that chooses optimizes DVFS parameters. The problem is solved through the Markov Decision Problem where the algorithms get a point if it's right and loses a point if the answer is wrong (i.e., the request was replied to after the deadline). Hipster instantiates a QoS monitor which collects all the related data, and this allows the decision mapping of a thread to a set of processors which can follow either a resource management policy or a power efficiency policy.

VII. Conclusion

This work has shown that it is possible to optimize the energy consumption using DFS while maintaining a similar level of QoS compared to existing DFS techniques. We analyzed how search requests behave and which frequency is most suitable for each request class. We developed an approach based on the observation that each request class needs a different frequency setting and implemented a proof-of-concept scheme in the Elasticsearch/Lucene. Our scheme was compared against the `Ondemand` governor, which is the mainstream solution on Linux, and obtained energy-savings up to 28%.

Acknowledgments

Results presented in this paper were obtained using the Chameleon testbed [13] supported by the National Science Foundation (USA). The work was also supported by FAPESB (JCB 008/2015) and the Brazilian federal government under CNPq grant (No. 430188/2018-8).

Source code availability

The source code of our solution with instructions for reproducibility is open and available at https://github.com/raijenki/elastic-hurryup/.

References

[1] L. A. Barroso, J. Clidaras and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2nd Edition, Morgan & Claypool Publishers. 2013.
[2] E. Schurman, J. Brutlag. *The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search* (Presentation Notes). June, 2009.
[3] J. Dean and L. A. Barroso. *The Tail at Scale*. Commun. ACM. 2013.
[4] M. Weiser, B. Welch, A. Delmers and S. Shenker. *Scheduling for Reduced CPU Energy*. OSDI. 1994.
[5] V. Pallipadi and A. Starikovskiy. *The Ondemand Governor: Past, Present and Future* (Tech Report). Intel Open Source Technology Center. 2006.
[6] H. Kasture, D. Bartolini, N. Beckmann and D. Sanchez. *Rubik: Fast Analytical Power Management for Latency-Critical Systems*. Micro. 2015.
[7] R. Nishtala, P. Carpenter, V. Petrucci and X. Martorell. *Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads*. HPCA. 2017.
[8] V. Petrucci, M. Laurenzano, D. Doherty, Y. Zhang, D. Mossé, J. Mars and L. Tang. *Octopus-Man: QoS-Driven Task Management for Heterogeneous Multicores in Warehouse-Scale Computers*. HPCA. 2015.
[9] S. Brin and L. Page. *The Anatomy of a Large-scale Hypertextual Web Search Engine*. Computer Network ISDN System. 1998.
[10] A. Bialecki. *Apache Lucene 4*. SIGIR. 2012.
[11] *Faban Harness and Benchmark Framework*. Available at http://java.net/projects/faban/.
[12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, et al. *Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware*. ASPLOS. 2012.
[13] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, et al. *Lessons Learned from the Chameleon Testbed*. USENIX ATC. 2020.
[14] R. Nishtala, V. Petrucci, P. Carpenter and M. Sjalander, *Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services*. HPCA. 2020.
[15] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. *Heracles: improving resource efficiency at scale*. ISCA. 2015.
[16] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, et al. *Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake*. MICRO. 2017.
[17] J. Guo et al. *Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces*. IWQoS. 2019.
[18] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. *Towards Energy Proportionality for Large-Scale Latency-Critical Workloads*. ISCA. 2020.
[19] T. Lidholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification: Java SE 8 Edition*. Oracle Corporation. 2015.
[20] M. Haque, Y. He, S. Elnikety, T. Nguyen, R. Bianchini, and K. McKinley. *Exploiting heterogeneity for tail latency and energy efficiency*. MICRO. 2017.