

# Exploring Computational Sprinting in New Domains

A Thesis

Presented in Partial Fulfillment of the Requirements for the Degree  
Master of Science in the Graduate School of The Ohio State  
University

By

Indrajeet Saravanan, B.Tech.

Graduate Program in Computer Science and Engineering

The Ohio State University

2019

Master's Examination Committee:

Christopher Charles Stewart, Advisor

Radu Teodorescu

© Copyright by  
Indrajeet Saravanan  
2019

## Abstract

The dawn of dark silicon and utilization wall are the main issues that current processors face. Moore’s law is virtually dead due to the breakdown of Dennard scaling. An array of novel approaches have been proposed to tackle the above-mentioned issues and computational sprinting is the latest one to be advocated. Computational sprinting is a set of management techniques that selectively speed up the execution of cores for short intervals of time followed up idle periods to achieve improved performance. This is physically feasible due to the inherent thermal capacitance that absorbs the heat generated by a rise in operating frequency or voltage. In our paper, we explore multiple avenues on how and where computational sprinting can be used. Firstly, we apply the core scaling method to sprint web queries which eventually makes page loads faster. We observe a 5.86% and 12.6% decrease in average load time for average-case and best-case scenarios respectively. Likewise, the number of page loads increase by 12.12% (average-case) and 21.88% (best-case). Secondly, we explore the Intel Cache Allocation Technology Tool to enable sprinting for server workloads. Since toggling L3 cache capacity for workloads introduces interference and uncertain consequences, we study the impact of cache stealing from co-located workloads. With base and polluted cache state being 4 MB and 1 MB respectively, we observe an average increase of 41.37 seconds in the runtime of Jacobi workload for every 20% increase in interruption from other workloads. We propose a machine

learning approach for future work. Finally, we study SLOs in practice to analyze the realities and myths surrounding their design and use. We learn that single-digit response goals are challenging, extreme percentiles for complex software is prohibited by black swans, and find the parameters of importance for evaluating infrastructure and complex cloud services.

Dedicated to my family.

## Acknowledgments

I would like to start by thanking my advisor, Dr. Christopher Stewart, for his support, guidance, and the countless moments of encouragement throughout my time at Ohio State. I have been working with Dr. Christopher since the beginning of my first year and could not have hoped for a better mentor. He masterfully guided me through my research – providing direction when needed, while consistently giving me the freedom to explore my own ideas. I would also like to thank Dr. Radu Teodorescu for being a part of my thesis committee. I was inspired to pursue my research and delve further into computer architecture after taking his class.

Furthermore, I want to thank the people who helped with the curation of this thesis: Nathaniel Morris, Jianru Ding, and Ruiqi Cao. Nathaniel implemented a large part of workload profiler and queue simulator. Jianru and Ruiqi did a great deal of work on the study of SLOs.

And finally, I wouldn't be where I am if not for my family. I owe this to each and every one of them. They have always wanted the best for me and I couldn't be more grateful for such a wonderful support system. My parents, Shyamala and Saravanan, and my sister, Haripriya, for their love and encouragement throughout my college career.

## Vita

2017 .....	B.Tech. Computer Science and Engineering, National Institute of Technology, Trichy
2017-2019 .....	Graduate Research Associate, The Ohio State University

## Fields of Study

Major Field: Computer Science and Engineering

# Table of Contents

	<b>Page</b>
Abstract . . . . .	ii
Dedication . . . . .	iv
Acknowledgments . . . . .	v
Vita . . . . .	vi
List of Tables . . . . .	ix
List of Figures . . . . .	x
1. Introduction . . . . .	1
1.1 Computational Sprinting . . . . .	1
1.1.1 Dynamic Voltage Frequency Scaling . . . . .	2
1.1.2 Core Scaling . . . . .	3
2. Computational Sprinting in Browsers . . . . .	5
2.1 Introduction . . . . .	5
2.2 Related Work . . . . .	6
2.3 Sprinting Parameters . . . . .	8
2.4 Design and Implementation . . . . .	9
2.4.1 Google Chrome Extension . . . . .	9
2.4.2 Zipf Distribution . . . . .	11
2.4.3 Algorithm . . . . .	13
2.5 Results . . . . .	15
2.6 Discussions . . . . .	28

3.	Computational Sprinting with Intel Cache Allocation Technology . . . . .	30
3.1	Introduction . . . . .	30
3.2	Design and Implementation . . . . .	33
3.3	Results . . . . .	34
4.	Study on Service-Level Objectives . . . . .	42
4.1	Introduction . . . . .	42
4.2	Analysis and Discussions . . . . .	43
5.	Conclusion and Future Work . . . . .	46

## List of Tables

Table	Page
2.1 Results for computational sprinting on web queries. . . . .	28

## List of Figures

Figure	Page
2.1 Dependency graph captured by traditional approaches (From [24]). . .	7
2.2 Overview of different Navigation Timings involved as per the W3C spec.	10
2.3 Workflow of computational sprinting for web queries. . . . .	12
2.4 Timeout for Sprinting vs Energy Consumed with budget of 500J. . .	16
2.5 Timeout for Sprinting vs Energy Consumed with budget of 3000J. . .	16
2.6 Timeout for Sprinting vs Energy Consumed with budget of 4000J. . .	17
2.7 Timeout for Sprinting vs Energy Consumed with budget of 5000J. . .	17
2.8 Timeout for Sprinting vs Energy Consumed with budget of 6000J. . .	18
2.9 Timeout for Sprinting vs Energy Consumed with budget of 7000J. . .	18
2.10 Timeout for Sprinting vs Energy Consumed with budget of 30000J. . .	19
2.11 Timeout for Sprinting vs Avg. Load Time with budget of 500J. . . .	20
2.12 Timeout for Sprinting vs Avg. Load Time with budget of 3000J. . . .	21
2.13 Timeout for Sprinting vs Avg. Load Time with budget of 4000J. . . .	21
2.14 Timeout for Sprinting vs Avg. Load Time with budget of 5000J. . . .	22
2.15 Timeout for Sprinting vs Avg. Load Time with budget of 6000J. . . .	22

2.16	Timeout for Sprinting vs Avg. Load Time with budget of 7000J. . . .	23
2.17	Timeout for Sprinting vs Avg. Load Time with budget of 30000J. . .	23
2.18	Timeout for Sprinting vs Pages Loaded with budget of 500J. . . . .	24
2.19	Timeout for Sprinting vs Pages Loaded with budget of 3000J. . . . .	25
2.20	Timeout for Sprinting vs Pages Loaded with budget of 4000J. . . . .	25
2.21	Timeout for Sprinting vs Pages Loaded with budget of 5000J. . . . .	26
2.22	Timeout for Sprinting vs Pages Loaded with budget of 6000J. . . . .	26
2.23	Timeout for Sprinting vs Pages Loaded with budget as 7000J. . . . .	27
2.24	Timeout for Sprinting vs Pages Loaded with budget as 30000J. . . . .	27
3.1	With Intel CAT, systems software manages cache lines. . . . .	31
3.2	Implementation of workload profiling in our approach. . . . .	33
3.3	Performance of Jacobi with 20 MB L3 cache. . . . .	35
3.4	Performance of K-means with 20 MB L3 cache. . . . .	35
3.5	Performance of BFS with 20 MB L3 cache. . . . .	36
3.6	Performance of Backprop with 20 MB L3 cache. . . . .	36
3.7	Performance of CFD with 20 MB L3 cache. . . . .	37
3.8	Percentage of Interruption vs Avg. runtime for Jacobi with base cache state as 4 MB and polluted cache state as 1 MB. . . . .	39
3.9	Percentage of Interruption vs Avg. runtime for Jacobi with base cache state as 4 MB and polluted cache state as 2 MB. . . . .	39
3.10	Percentage of Interruption vs Avg. runtime for Jacobi with base cache state as 4 MB and polluted cache state as 3 MB. . . . .	40

4.1	Heat map of delay and percentile goals from industry sources. . . . .	43
-----	---	----

# Chapter 1: Introduction

## 1.1 Computational Sprinting

For the past 40 years, the chip industry followed Moore’s law— the number of transistors on a processor doubled every 24 months. The main limitation that stunted the growth of this trend is due to the breakdown of Dennard scaling [22, 41, 8]. This law states that as the size of transistors decreases, power density remains constant. In other words, for an increase in the count of transistors by a factor of  $\lambda^2$ , the operating voltage decreases by a factor of  $\lambda$  and clock frequency increases by a factor of  $\lambda$ . Due to the chip’s unreliability to operate at low voltages, a halt to the increasing clock frequency was brought [28]. The number of transistors that cannot be powered up due to the limit on Thermal Design Power (TDP) has become a growing issue. This phenomenon is called Dark Silicon [41, 23, 27]. The CTO of ARM, Mr. Mike Muller announced that by the end of 2020, only 9% of the total transistors present in a multi-core processor could be powered up at one time [19].

On the other hand, current applications are highly interactive [48, 30, 6, 35]. They require intense computation for brief intervals which are followed by extended periods of idleness [27]. Examples of such applications include web browsers during page loading [37, 31, 10], web search [29, 44], Internet services [40, 38, 36, 39], edge driven IoT workloads and analytics [2] and route planning.

Computational Sprinting tackles the union of the above-mentioned issues [27]. It is a set of mechanisms which allow response time reduction at the expense of additional power consumption. By boosting frequency and voltage for momentary time spans [23, 13, 15], the idle cores are activated, in turn increasing the performance of the chip. While sprinting, the chip generates heat faster than it can dissipate but the rise in temperature is gradual. The thermal capacitance of the chip absorbs the heat and does not allow a sudden spike in the chip's temperature. After the temperature hits the threshold value, sprinting is aborted by reducing the frequency or powering down cores. The base cores take care of unfinished computation if present. This technique helps in improving the responsiveness of the applications.

Some of the most common means of achieving computational sprinting are Dynamic Voltage Frequency Scaling (Voltage boosting and CPU throttling) and core scaling. We will look at them in detail in the following subsections. This thesis does not cover an widely used alternative: degrading the quality of answers momentarily [17, 1, 25, 2, 46]. These approaches are beyond the scope of this thesis.

### **1.1.1 Dynamic Voltage Frequency Scaling**

Dynamic Voltage Frequency Scaling (DVFS) is the mechanism in which the voltage or frequency of the processor could be increased to expend more power and increase performance or could be decreased to conserve power and decrease performance.

Voltage boosting refers to the process of modifying the supply voltage on the fly. You can either undervolt or overvolt a processor. CPU throttling modifies the number of instructions processed per cycle by the processor and hence, directly correlates to

the performance of the machine. You can over-clock or under-clock a processor based on your needs.

Intel’s Turbo Boost technology achieves higher performance by running the active cores of a processor at a frequency that is greater than the base operating frequency [9]. The maximum frequency that Intel Turbo Boost can allow is constrained based on the number of active cores, estimated power consumption, processor temperature, and estimated current consumption. PUPiL, a hybrid power capping system focuses on timeliness and efficiency and beats RAPL (Running Average Power Limits), Intel’s software approach by 18% in mean performance [47]. Hsu et. al proposed Adrenaline to address the shortage of techniques to improve tail latency for online data-intensive services (OLDI) [15]. Adrenaline makes use of Short Stop, a circuit design that enables fine-grained Voltage-Frequency scaling [26].

### **1.1.2 Core Scaling**

Core Scaling is the technique of reducing latency by powering up more cores to work on a specific task. This is a means by which we derive parallelism from the hardware.

In most cases, core scaling and DVFS are combined to provide an efficient means for sprinting. Haque et. al proposed Few-to-Many incremental parallelization, which utilizes hardware and software parallelism to address tail latency by dynamically applying resources based on the progress of each request [13].

The remainder of this paper is organized as follows. In Chapter 2, we learn the effects of using computational sprinting for speeding web queries. In Chapter 3, we employ Intel Cache Allocation Technology to manipulate the last-level cache as a

technique to realize computational sprinting. Chapter 4 studies the Service Level Objectives employed in practice. Finally, Chapter 5 draws conclusions.

## Chapter 2: Computational Sprinting in Browsers

### 2.1 Introduction

Mobile users expect webpages to load instantly. A recent survey reveals that 2 seconds is the average wait time for a webpage to load for more than 50% of internet users. Failure to meet such high standards could result in user dissatisfaction and a huge loss in revenue for service providers eventually. To put this in perspective, Amazon could suffer a loss of \$1.6 billion in sales per year for a mere delay of one second in page load time [16]. The Google Search engine previously used page load time as a key factor in ranking for desktop searches. Starting July 2018, Google released "Page Update" which focuses on mobile page speed to be a core ranking factor in search results [43]. In spite of the significance of page load time, the collective effort of the research community has fallen short to match user demands. There are various factors that might affect the page load time of webpages. Substandard design, network issues, server failure, etc. are a few common examples. The reality is that most of these issues act in unison, and hence it is not trivial to precisely locate or understand the complexity behind slow page loads. Based on past studies, the following three reasons attribute to the majority of delay observed in load time [45].

1. **Size of web pages:** Unnecessary code that is transmitted that will only be used based on user activity.

2. **Disparity in computation and data transfer:** Due to inefficient scheduling and ordering in loading objects of a webpage, data transfer and processing don't take place concurrently.
3. **Complex dependencies in page resources:** Convolved dependencies between objects prolong the time taken to reach the final state of a webpage.

To tackle all the above issues, we propose our approach that works on the user's device assuming best-effort services in other junctions. We employ computational sprinting for specific web requests to abide by a predefined service level objective. By tracking parameters such as arrival rate, queuing delay, energy budget, etc. and an option to choose from multiple sprinting policies, sprinting is achieved. We collect data by loading the most visited 100 pages from Alexa Top 500 Pages [42].

All of our results are collected on a Chrome browser as it is considered to be the de-facto standard for browsers and serves as a platform for 62.41% of internet traffic [34]. To keep track of page load time of each request, we designed a Chrome Extension that employs the Navigation Timing API to record page metrics. Each web request is associated with a process ID and by using OS tools, isolated-sprinting for discrete pages is achieved. We find that by sprinting web queries, a 5.86% and 12.6% decrease in load time is observed for average-case and best-case respectively. Correspondingly, the energy consumption increases by 1.33% and 3.33% for average-case and best-case respectively.

## 2.2 Related Work

There have been numerous undertakings to understand the factors that affect page load time. These efforts have been implemented on the various layers present in the

life-cycle of a page load. Silo reduces the number of HTTP requests to load the page by using JavaScript and DOM Storage [20]. This approach modifies the structure and content of web pages to enable fine-grained caching. SPDY and QUIC are protocols deployed in the network level to improve the performance of page load latencies [4, 33]. These methods help in minimizing the network costs but do not address the substantial number of objects being fetched. At the computation level, solutions such as ZOOMM and Adrenaline focus on parallelizing the requests to objects in web pages [7, 18]. This approach fails to have a considerable impact because of the dependencies present in webpages.

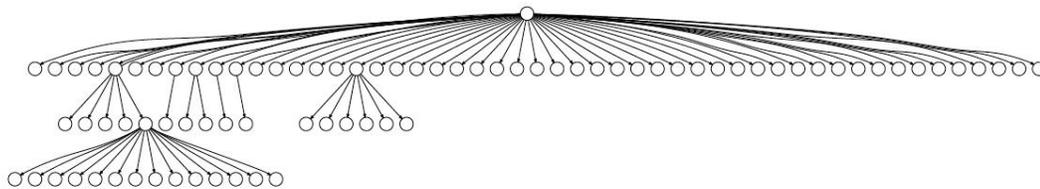


Figure 2.1: Dependency graph captured by traditional approaches (From [24]).

A website can be structured in the form of a dependency graph as shown in Figure 2.1. The graph reveals dependency between modules and also edges that could be processed in parallel. Browsers are provided with a partial dependency graph which leads to constrained loading of objects fearing of violating dependencies. Netravali et. al proposed Scout, a fine-grained dependency tracker that reveals finer critical paths and Polaris, a dynamic client-side scheduler that takes advantage of Scout to reduce page load time [24].

## 2.3 Sprinting Parameters

To discuss sprinting policies, we need to learn the terminology of input parameters.

1. **Timeout:** Decision of whether to sprint or not is made based on the value of timeout. If the total execution time of a query crosses the timeout, then sprinting is triggered.
2. **Arrival Distribution:** This parameter governs the rate at which queries come into the system. Pareto, Exponential, and Gamma distributions could be used for the same.
3. **Sprint Budget:** This signifies the amount of resources that are available that could be used for sprinting. This can be represented in seconds or joules.
4. **Service Rate:** It is defined as the inverse of mean processing time for query executions which aren't sprinted. The most common unit in practice is queries per hour. In statistics,  $\mu$  is used to represent the service rate.
5. **Arrival Rate:** It is defined as the mean number of arrivals per unit time. It is generally expressed as queries per second and represented with  $\lambda$  in statistics. It is shown as a percentage of service rate. Inter-arrival time is the time between subsequent arrival of queries.
6. **Queuing delay:** This is the mean amount of time queries spend in a queue waiting for a slot to start execution. Computational sprinting creates an inter-dependency between queuing time and processing time.

7. **Processing Time:** This is the total amount of time spent on processing a query with computational resources. It includes both the time spent on processing the query with and without sprinting.
8. **Response Time:** This is the summation of processing time and queuing time. Based on the scenario, looking at the mean response time could be useful along with its variance.

## 2.4 Design and Implementation

In this section, we will look at the modules that provide the ability to sprint page loads based on policies.

### 2.4.1 Google Chrome Extension

Google Chrome is the leading web browser in the market amongst other alternatives such as Mozilla Firefox, Microsoft Edge, etc. To make smart split decisions on whether to sprint or not, keeping track of the load time is essential. Google Chrome allows us to add functionality to the web browser by means of extensions. Extensions could be downloaded from the Chrome Web Store or can be developed and deployed by the user.

The design and structure of the extension is as follows:

```
root
├── background.js
├── content.js
├── icon.png
└── manifest.json
```

It is mandatory for every Chrome extension to have a manifest.json file. It provides crucial details such as version, permissions, and URLs on which content scripts need to run. The content.js file is a content script. A content script is a JavaScript file that runs in the context of web pages. In other words, this file can interact with the pages that are loaded in the browser. For the content script to interact with other components of the extension, it uses Google's message passing API. To enable listening for other actions happening in the browser, we require the background.js file. It's a background script that interacts with the content script. The icon image file is displayed on the extensions section on the browser.

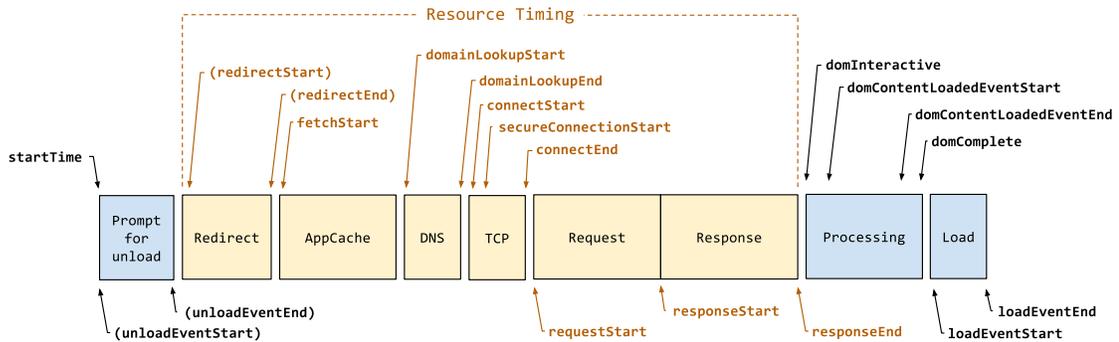


Figure 2.2: Overview of different Navigation Timings involved as per the W3C spec.

The Navigation Timing API provides data that can be used to measure the performance of websites. It has various interfaces such as Performance, PerformanceNavigationTiming, PerformanceNavigation, and PerformanceTiming. For our purpose, we have used the PerformanceTiming interface to gather critical values like domainLookupStart, connectStart, etc.

We use Figure 2.2 to compute important metrics such as Total First Byte time, Latency, DNS/Domain Lookup time, Server connect Time, Server Response Time, Page Load time, Transfer/Page Download Time, DOM Interactive Time, DOM Content Load Time, DOM Processing to Interactive, DOM Interactive to Complete, and Onload.

We employed XAMPP stack for collecting all the above metrics locally [11]. XAMPP is a web server solution stack that contains Apache HTTP Server, MariaDB database, PHP and Perl. XMLHttpRequest (XHR) is used to transfer data from the web browser to the SQL Web server. This server listens for GET requests. Once the page is done loading, the data is passed by adding data to URL variables and launching the GET request. All the above functionality is programmed in PHP.

## 2.4.2 Zipf Distribution

Zipf's law states that given some corpus of natural language utterances, the frequency of any word is proportional to its rank in the frequency table. A frequency table is a table that displays the frequency of various outcomes in a sample.

Formally, let:

- $N$  be the number of elements;
- $k$  be their rank;
- $s$  be the value of the exponent characterizing the distribution.

Zipf's law then predicts that out of a population of  $N$  elements, the normalized frequency of elements of rank  $k$ ,  $f(k;s,N)$ , is:

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^N (1/n^s)} \quad (2.1)$$

From [21], we can learn that the trace of websites visited for an average user follows the Zipf distribution.

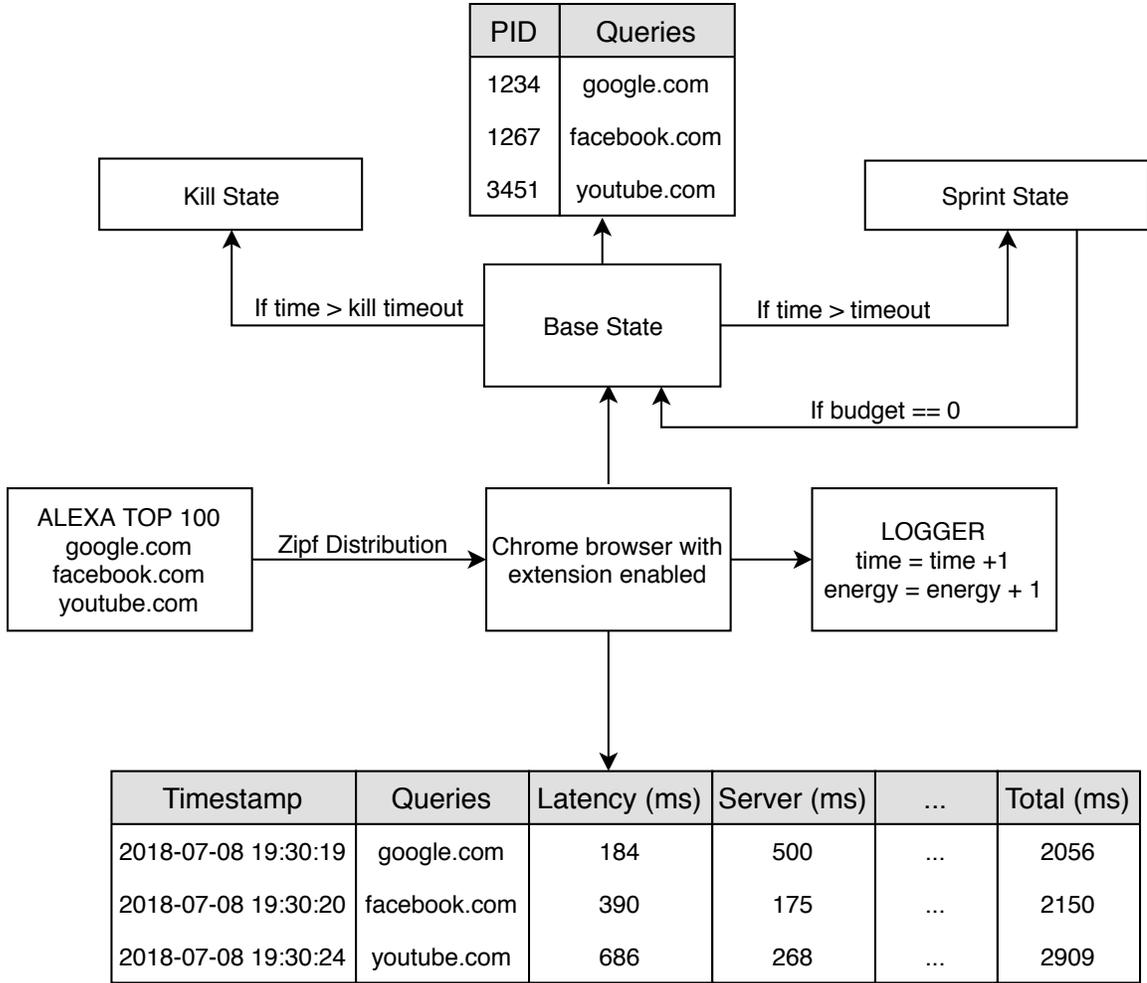


Figure 2.3: Workflow of computational sprinting for web queries.

### 2.4.3 Algorithm

As shown in Figure 2.3, each query is associated with a parent process ID (PID) as soon as its execution starts. Using this PID, we can keep track of all the child PIDs and the amount of time spent on processing this query. If the current load time is greater than the specified timeout, we move the query to sprint state. This is achieved using `taskset`, a Unix command capable of modifying the core affinity of processes. We use a global budget for sprinting. Every time, a query is processed in sprint state, the energy consumed is deducted from the global budget. Eventually, when the budget equals zero, any queries that are in sprint state are migrated back to the base state. We use the `perf stat` Unix command to monitor energy consumption.

The following code snippet communicates the important methods in the program.

```
# Input Parameters
# path_webpages
# energy_budget
# sprint_timeout
# kill_timeout
# inter_arrival_scale
# base_cores
# sprint_cores

time_logger = []
energy_logger = []
sprinted_pids = []

def Logger():
    if (flag == 1):
        return
    threading.Timer(1.0, Logger).start()
    time = stat_cmd.run().get_time()
    energy = stat_cmd.run().get_energy()
    time_logger.append(time)
    energy_logger.append(energy)
```

```

Logger()

def kill():
    if(flag == 1):
        return
    threading.Timer(0.1, kill).start()
    pids = get_pids_cmd.run()
    etimes = get_etimes_cmd.run()
    if len(pids) != 0 and len(etimes) != 0:
        pids_list = pids.generateList()
        etimes_list = etimes.generateList()
        for id, etime in enumerate(etimes_list):
            if etime > kill_timeout:
                kill_cmd.run()

kill()

def sprint():
    if(flag == 1):
        return
    threading.Timer(0.1, sprint).start()
    pids = get_pids_cmd.run()
    etimes = get_etimes_cmd.run()
    if len(pids) != 0 and len(etimes) != 0:
        pids_list = pids.generateList()
        etimes_list = etimes.generateList()
        for id, etime in enumerate(etimes_list):
            if etime > sprint_timeout:
                if sum(energy_logger) < energy_budget
                    if pids_list[id] not in sprinted_pids:
                        taskset_cmd.run()
                        sprinted_pids.append(pids_list[id])

sprint()

id = 0
with open(webpages, 'r') as queries:
    for query in queries:
        taskset_chrome_cmd(base_cores, query).run()
        time.sleep(exponential_distribution[id])
        id = id+1

flag = 1

```

## 2.5 Results

The following list specifies the range of values used for experiments.

- **Sprint Budget:** 500, 3000, 4000, 5000, 6000, 7000, 30,000 (joules)
- **Inter-arrival time:** 2.6, 4.4, 6.6, 7.9, 8.4 (seconds)
- **Timeout for Sprinting:** 0, 4, 8, 16, 32 (seconds)
- **No. of Pages loaded:** 100
- **No. Of Base Cores:** 1
- **No. Of Sprint Cores:** 4

These tests were run on an Intel(R) Core (TM) i5-4210U CPU @ 1.70 GHz processor with a total of 8 cores. Intel Turbo Boost was disabled by setting the governor as userspace. This allows the user to set the operating frequency of the processor.

The following plots help us see the correlation on how the energy consumed varies based on timeout for sprinting. We have grouped our results with inter-arrival time between queries.

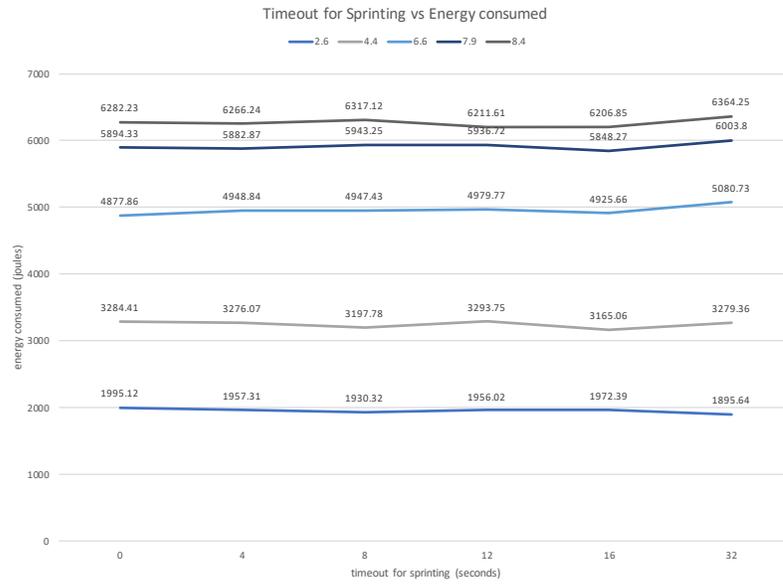


Figure 2.4: Timeout for Sprinting vs Energy Consumed with budget of 500J.

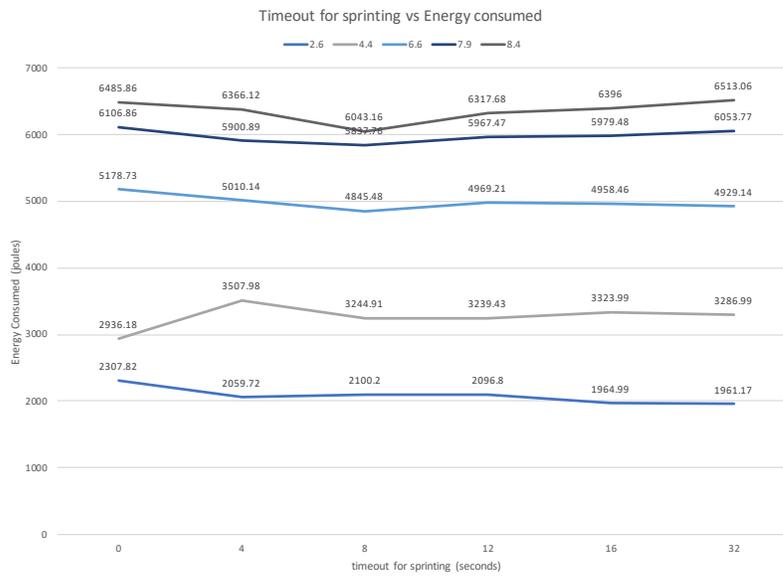


Figure 2.5: Timeout for Sprinting vs Energy Consumed with budget of 3000J.

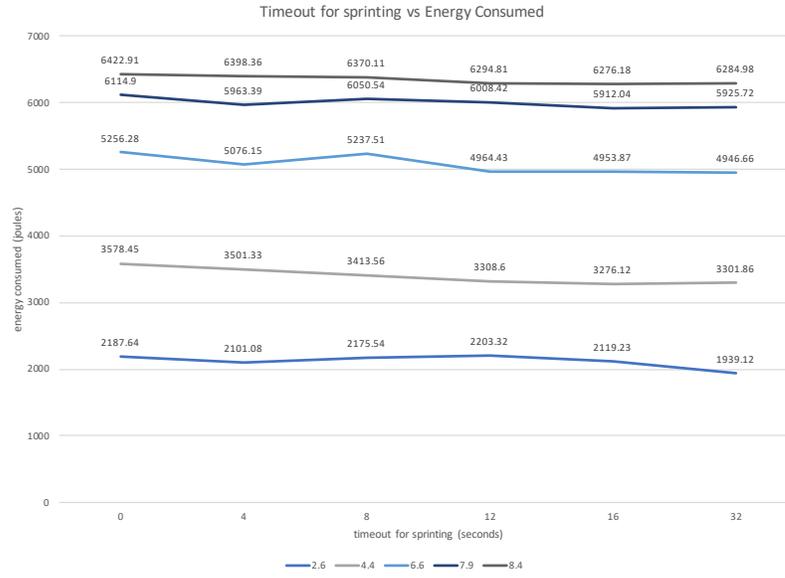


Figure 2.6: Timeout for Sprinting vs Energy Consumed with budget of 4000J.

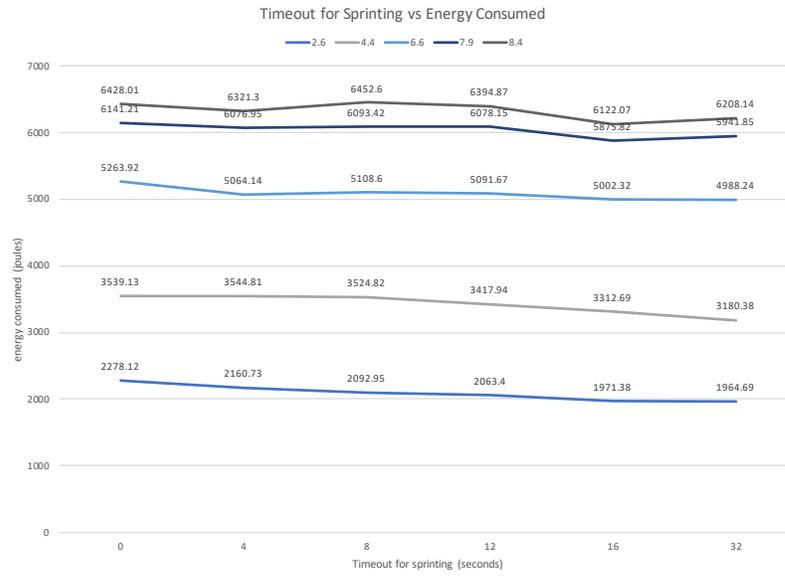


Figure 2.7: Timeout for Sprinting vs Energy Consumed with budget of 5000J.

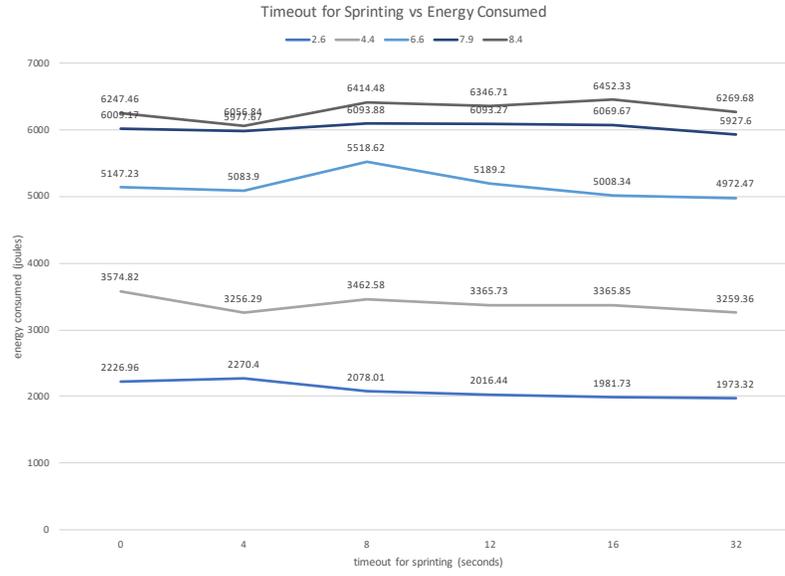


Figure 2.8: Timeout for Sprinting vs Energy Consumed with budget of 6000J.

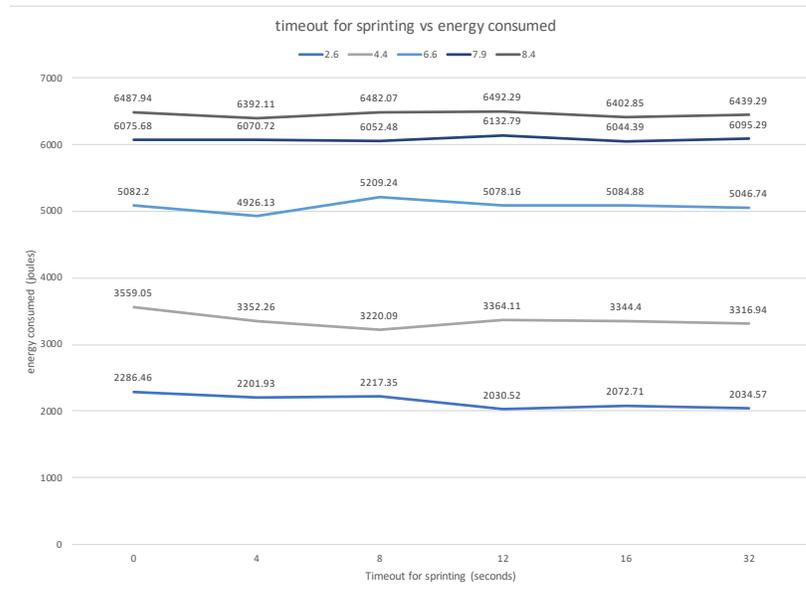


Figure 2.9: Timeout for Sprinting vs Energy Consumed with budget of 7000J.

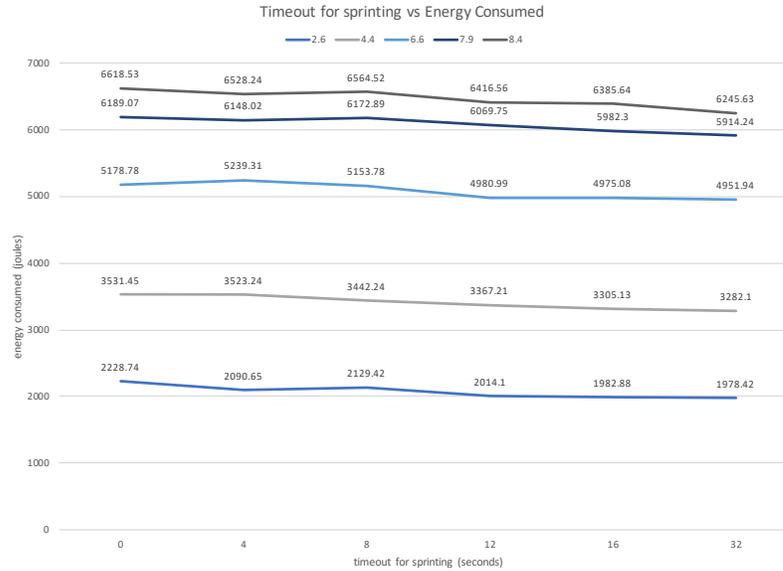


Figure 2.10: Timeout for Sprinting vs Energy Consumed with budget of 30000J.

The first observation in all the above plots is the trend of increasing energy consumption for fixed energy budget and fixed timeout as inter-arrival time between queries increases. This only makes sense since increased inter-arrival time means the processor spends more time processing queries and hence the consumption of energy also increases.

We can also see in the above figures on how for a fixed energy budget and fixed inter-arrival time, the energy consumption varies. As the timeout for sprinting increases, the amount of energy consumed decreases relatively. There are two ways we can reason for the above observation.

1. Since more pages are done loading within the timeout, the energy spent on sprinting for loading pages will be lesser.

- The amount of time spent on sprint state will be lesser as timeout increases, hence the energy consumption accordingly reduces.

For instance, for all inter-arrival times in Figure 2.10, we can see a steady decrease in the energy consumed as timeout increases. In this specific case, an energy budget of 30,000J virtually represents unlimited sprinting. With timeout value as zero, every query will be executed in sprint state till its completion. With the increase in timeout, energy increases.

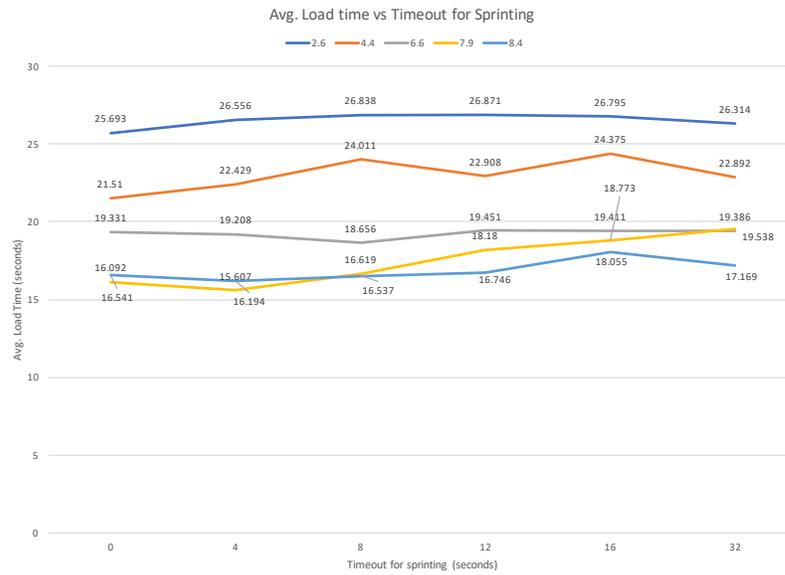


Figure 2.11: Timeout for Sprinting vs Avg. Load Time with budget of 500J.

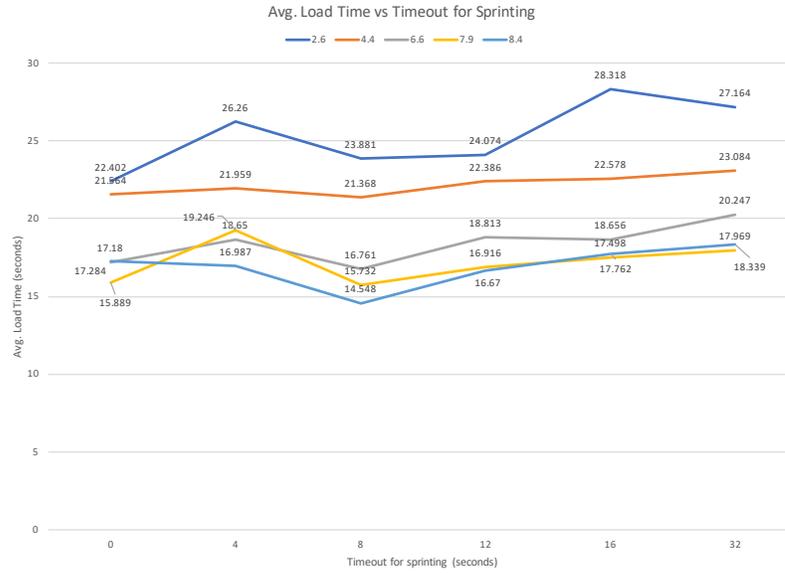


Figure 2.12: Timeout for Sprinting vs Avg. Load Time with budget of 3000J.

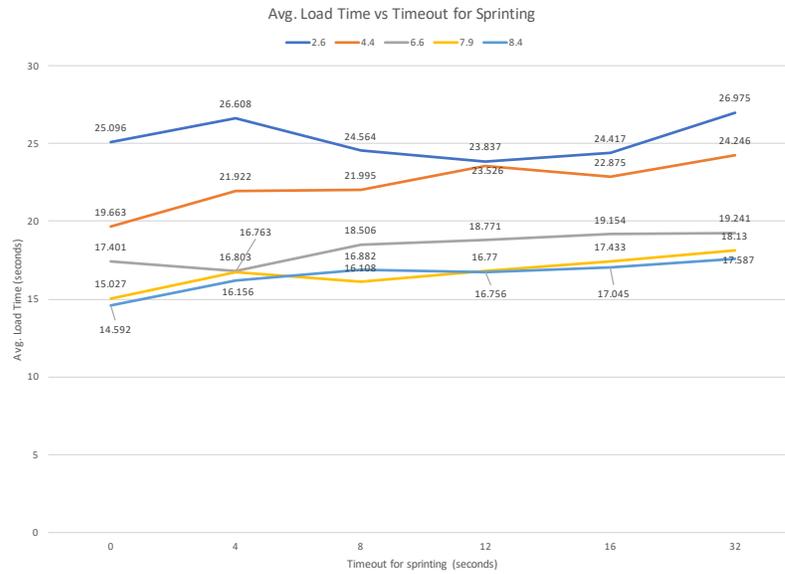


Figure 2.13: Timeout for Sprinting vs Avg. Load Time with budget of 4000J.

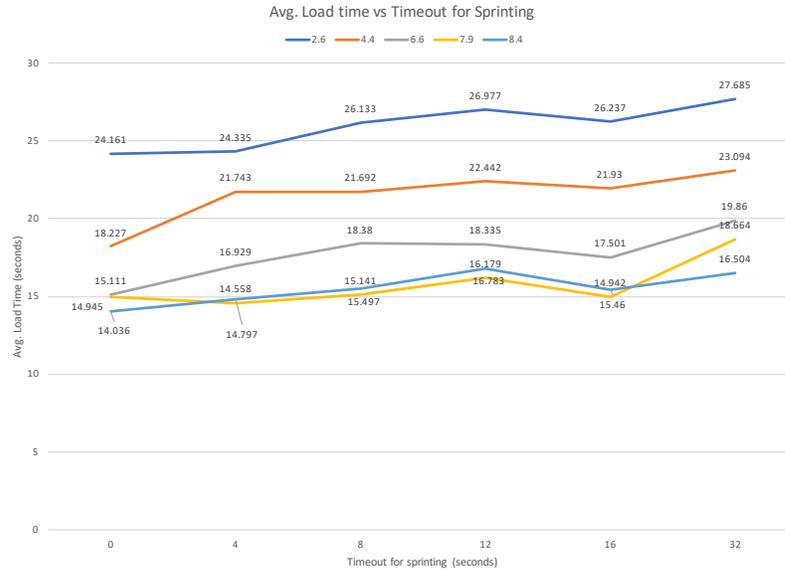


Figure 2.14: Timeout for Sprinting vs Avg. Load Time with budget of 5000J.

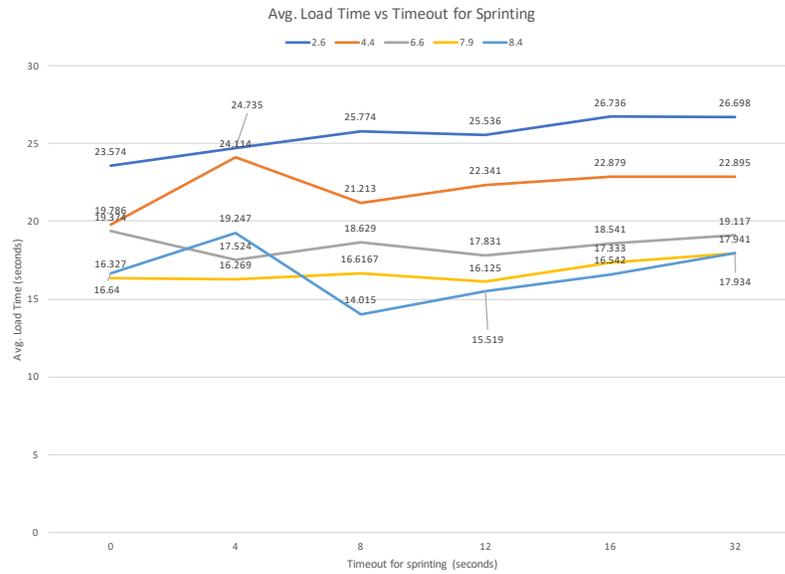


Figure 2.15: Timeout for Sprinting vs Avg. Load Time with budget of 6000J.

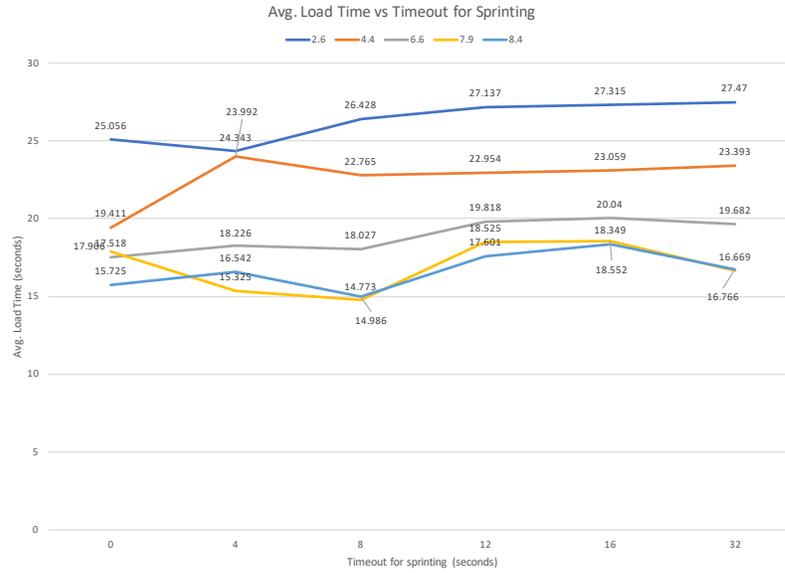


Figure 2.16: Timeout for Sprinting vs Avg. Load Time with budget of 7000J.

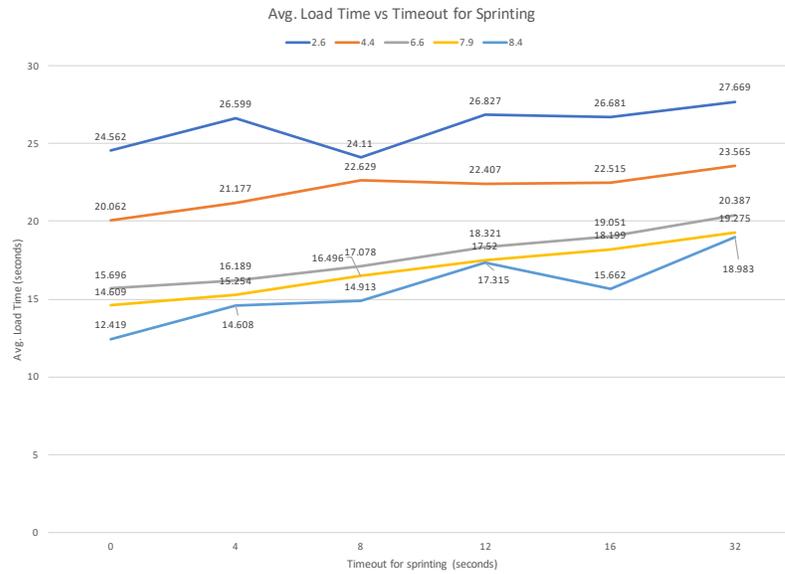


Figure 2.17: Timeout for Sprinting vs Avg. Load Time with budget of 30000J.

The above plots depict the relation between average load time and timeout for sprinting with varying energy budget. The main observation is when the inter-arrival time and energy budget is fixed. In Figure 2.17, the average load time when timeout for sprinting is 0 is 12.49 seconds. As timeout increases, the average load time increases. However, anomalies are sometimes observed, for instance at timeout for sprinting at 12 seconds, we see a spike in average load time.

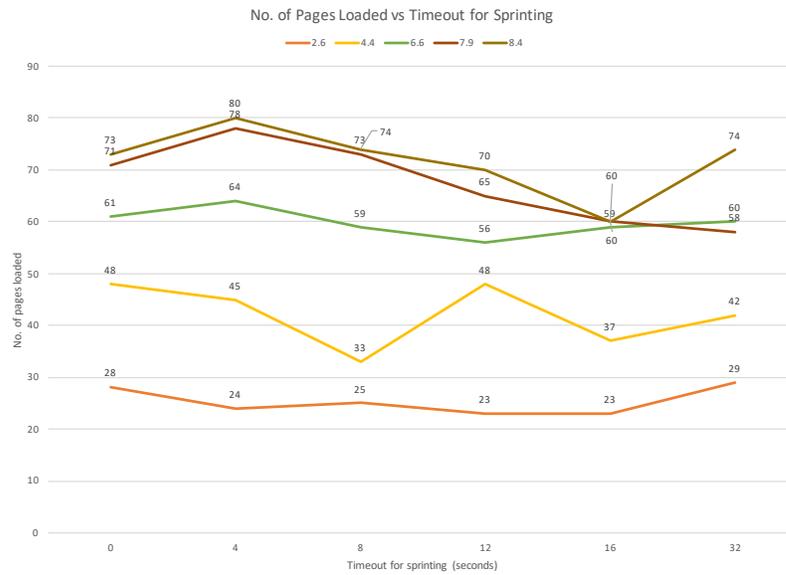


Figure 2.18: Timeout for Sprinting vs Pages Loaded with budget of 500J.

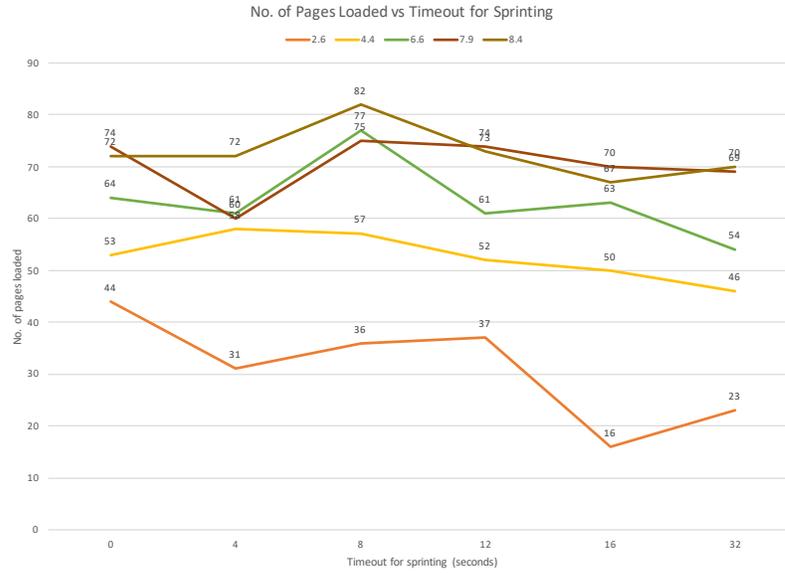


Figure 2.19: Timeout for Sprinting vs Pages Loaded with budget of 3000J.

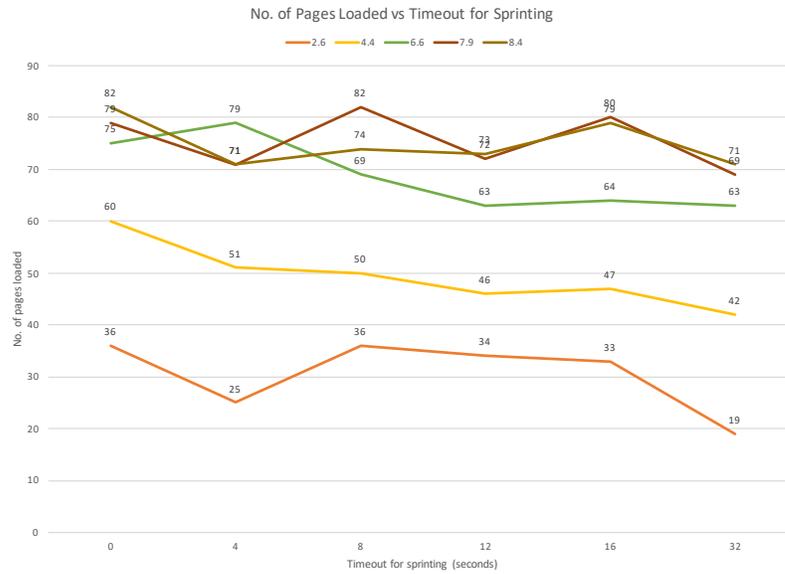


Figure 2.20: Timeout for Sprinting vs Pages Loaded with budget of 4000J.

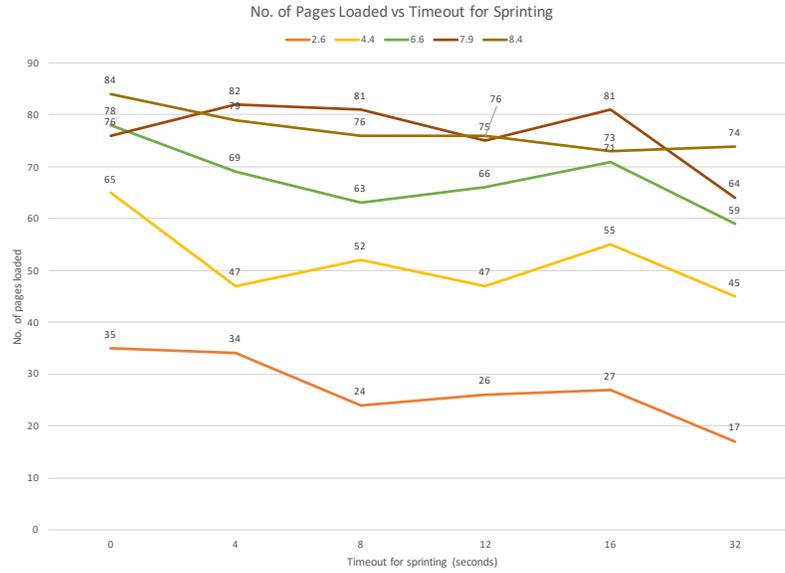


Figure 2.21: Timeout for Sprinting vs Pages Loaded with budget of 5000J.

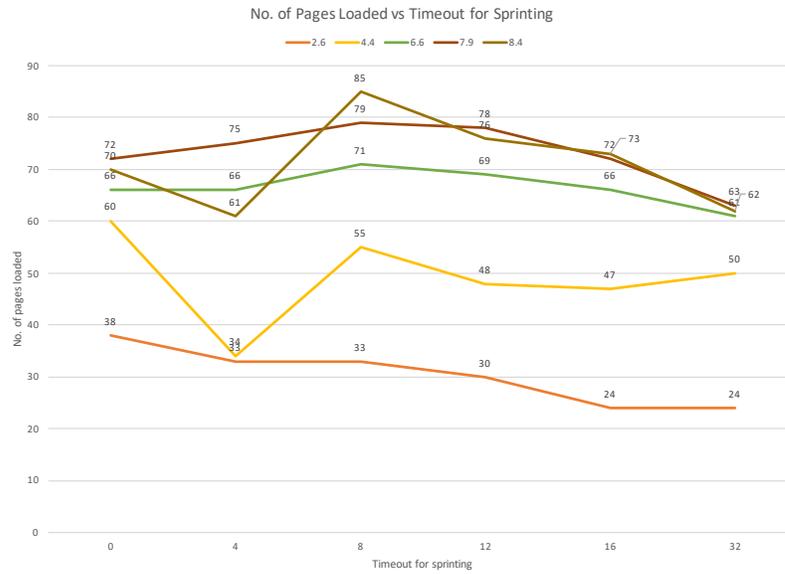


Figure 2.22: Timeout for Sprinting vs Pages Loaded with budget of 6000J.

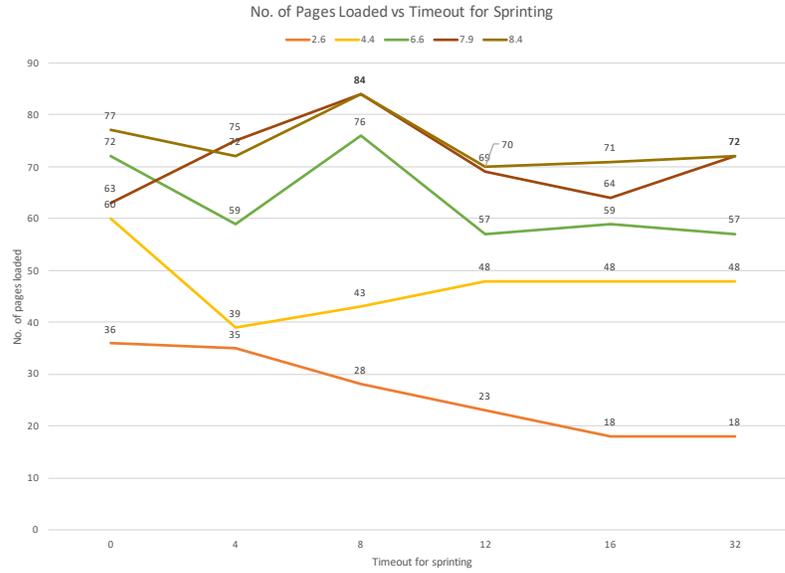


Figure 2.23: Timeout for Sprinting vs Pages Loaded with budget as 7000J.

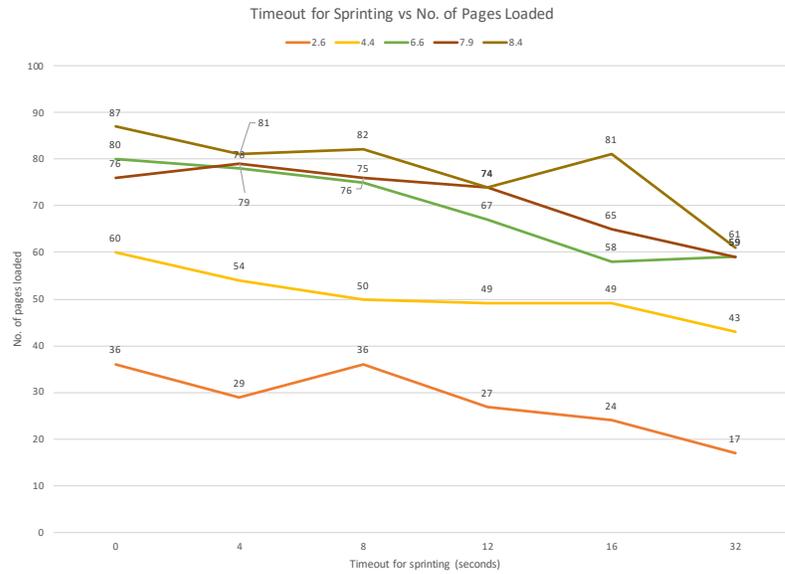


Figure 2.24: Timeout for Sprinting vs Pages Loaded with budget as 30000J.

The above charts plot the number of pages that successfully loaded against timeout for sprinting with varying energy budgets. With inter-arrival time increasing, the number of successful page loads also increase. This is because of the decreasing amount of termination of stale queries.

An important observation in Figure 2.20 is that with inter arrival time as 2.6 seconds, 30 pages are loaded on average with sprinting at different timeouts. However, with 32 as the timeout value, the case where there is no sprinting possible, we have only 19 pages loaded. This trend can be observed in other figures as well like Figure 2.21, Figure 2.22, and Figure 2.24. We can conclude that sprinting does indeed help in loading extra web pages that would otherwise not load given the conditions.

## 2.6 Discussions

We divide our sprinting policy into three categories: 1) No Sprinting 2) Average Sprinting 3) Always Sprinting. They can be correlated to worst-case, average-case and best-case scenarios respectively. Table 2.1 below reveals the improvement in employing computational sprinting for webpage loads. The average inter-arrival time is 5.98 seconds.

Category	Energy Consumed	Avg. Load Time	Avg. Page Loads
Worst-Case	4507.35J	21.21s	51.82
Average-Case	4567.24J	19.97s	58.11
Best-Case	4657.24J	18.58s	63.17

Table 2.1: Results for computational sprinting on web queries.

By using computational sprinting, we can decrease the average load time by 5.86% (average-case) and 12.6% (best-case). The number of page loads increase by 12.12% (average-case) and 21.88% (best-case). Accordingly, the energy consumption increases by 1.33% (average-case) and 3.33% (best-case).

While we can see trends in our results, the number of anomalies are equally high. This implies that this technique works in speeding up web queries but we cannot completely rely on them. One of the main reasons for this could be the delay in moving from base state to sprint state for a specific query. The delay and improvement gained from sprinting cancels out each other and we are left with no net improvement.

In the future, we can group a bunch of queries and sprint them all together at the same time which would reduce the overhead of changing states. But the downside would be the depletion of sprint budget rapidly. All experiments were carried out on Chrome browser on laptops. Since mobile browsers have surpassed the internet usage, we can look at manipulating the cores of mobile devices.

## Chapter 3: Computational Sprinting with Intel Cache Allocation Technology

### 3.1 Introduction

In 2016, Intel introduced the cache allocation technology (CAT) in the Xeon 2600 V4 architecture [14]. Figure 3.1 compares the traditional caching approach to CAT. CAT functions by storing tags for both virtual memory and process ID. Cache hits must match the process ID. CAT lets workloads share last-level cache and exclusively reserve cache lines. Workloads designed to execute on shared servers can now have guaranteed SRAM cache. Integrated SRAM supplies data 12X faster than DRAM. CAT substantially speed up workloads that exhibit cache locality.

SRAM is created using only electronic switches and transistors. It is the only memory that can be synchronously integrated with processor clocks. SRAM represents the majority of transistors on modern processors. Since 2012, SRAM storage on Xeon processors has grown by 40%. L3 caches were present in only 25% of Xeon processors in 2012, but are now present in 40%. When manufacturing processes support 5nm transistors, SRAM storage on the chip will increase further. The Xeon 2800 has a cache size of 60 MB. This server recently broke the world record for TPC-H QphH@Size (Composite Query per Hour Performance Metric). TPC-H is a decision support benchmark which is adopted hugely in industry [5].

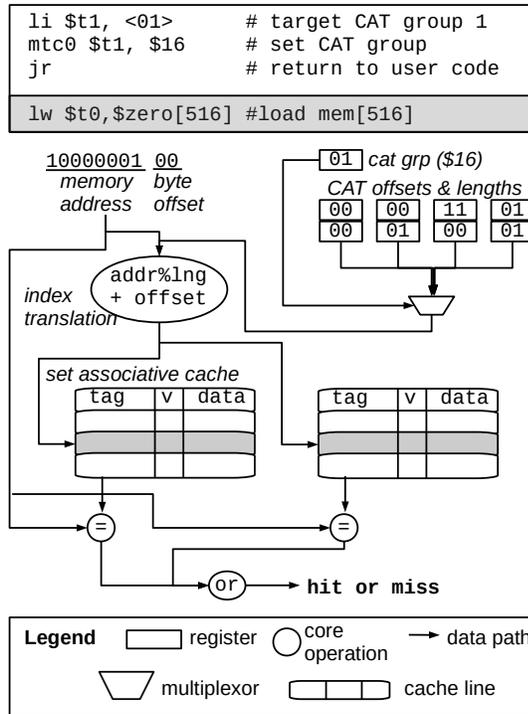


Figure 3.1: With Intel CAT, systems software manages cache lines.

Few workloads need exclusive access to large amounts of cache. For example, small workloads such as 1 GB TPC-H databases perform best with L2 cache of 10 MB. It is important to note that 80% of databases worldwide are less than 1 GB. However, competing workloads cause jitter and can degrade throughput by 15%. One solution is to over-provision the cache, giving each workload access to more cache than needed. Over-provisioning is (1) hard to control and (2) wasteful. On a Xeon 2800, over-provisioning a database that uses 10 MB by 25% degrades parallelism by 25%. CAT reduces jitter. Workloads sensitive to cache interference can now run on shared servers without over-provisioning. Significantly reducing the cost to execute these workloads in the cloud.

The main problems with state of the art approaches are listed below.

- Over-provisioning is too costly
- Cache interference
- Cloud resources execute this workload inefficiently
- Economies of scale as the database grows

Multi-core machines can run processes sharing the software to manage on-chip cache lines (SRAM). Guest workloads can now exclusively reserve cache lines and exploit integrated SRAM (100X faster than main memory) to run efficiently on shared CPU. However, SRAM has low memory density. SRAM should be reserved parsimoniously. Static management policies over-provision the SRAM, fragment cache between workloads, trigger unneeded scale-out and, degrade whole system performance. In contrast, computational sprinting is an innately parsimonious approach. A workload is initially allocated few resources, but if a service level objective (SLO) violation becomes likely, additional resources are allocated and the workload is sped up, i.e., a sprint. Sprinting with CPU resources, e.g., voltage and core scaling, can reduce SLO violations by 95%. Cache allocation technology supports a new sprinting mechanism: at-risk workloads could temporarily access more cache lines. This tool has the ability to modify cache lines in the range of 400-700 milliseconds. This would be adequate for workloads that take 2-3 seconds to complete execution. However, the downside is that cache interference affects speedup and predictability. Hence, our study focuses on studying the feasibility of utilizing cache allocation technology for scheduling workloads on servers.

### 3.2 Design and Implementation

Intel’s CAT tool enables control over last-level cache by introducing *Class of Service* (CLOS) which represents a group of processes (or) threads (or) applications. A process can be added or removed from a CLOS. Every CLOS has a resource capacity bitmask (CBM) that indicates the amount of cache available for use. However, there are two constraints in creating a CLOS. The allocated ways must be contiguous and each CLOS should have at least two ways. These ways are mapped logically to processors. Assume a process was assigned to CLOS A. Due to subduced performance, it is moved to CLOS B which has more cache lines assigned to it. This doesn’t imply that the cache hits for this process only come from CLOS B, they can be serviced by cache lines of CLOS A as well.

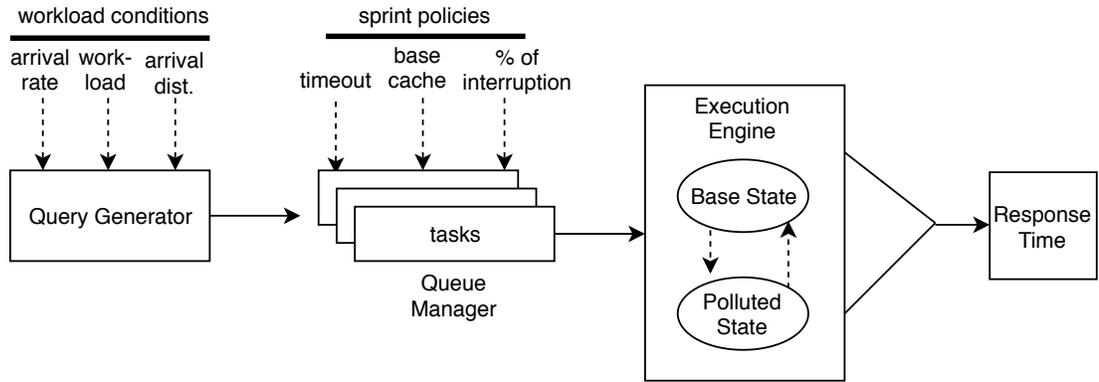


Figure 3.2: Implementation of workload profiling in our approach.

We use Rodinia, a benchmark for heterogeneous computing for our experiments [32]. From Figure 3.2, we can see that the query generator uses arrival rate, workload type, and arrival distribution to produce HTTP query requests. All the requests enter

a FIFO queue. The query at the head of queue starts execution once a slot opens in the execution engine and the query is popped from the queue. Every query has a timestamp variable which helps in calculating the time spent in queue and execution. Every request has a base cache that is set. The cache capacity of each query is toggled between a base state and polluted state based on a Poisson distribution. The query manager makes decisions based on timeout in our case. The response time and other metrics of cache usage are recorded locally using perf stat Unix tool.

### 3.3 Results

The following experiments are to profile the impact of L3 cache on the workload's performance. We use the results from this experiment to decide on how many cache lines would be ideal to use for base mode and sprint mode.

The processor specs used for experiments are as follows: Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz with a total of 16 cores. The shared L3 cache size is 20 MB.

We can observe from Figure 3.3 that there is a stark decrease in the execution time of the workload as the number of cache ways increase. There is a 71.95% decrease in runtime between using 1 MB and 2 MB of L3 cache. Hence, Jacobi would perform a lot better if it's assigned 2 MB rather than 1 MB. The best response time is achieved at 4 MB of cache with a 75.3% decrease and over-provisioning has no effect. Actually, it causes extra latency in some cases. We can also see that the percentage of last-level cache load misses in cache hits reduces as the size of cache assigned increases.

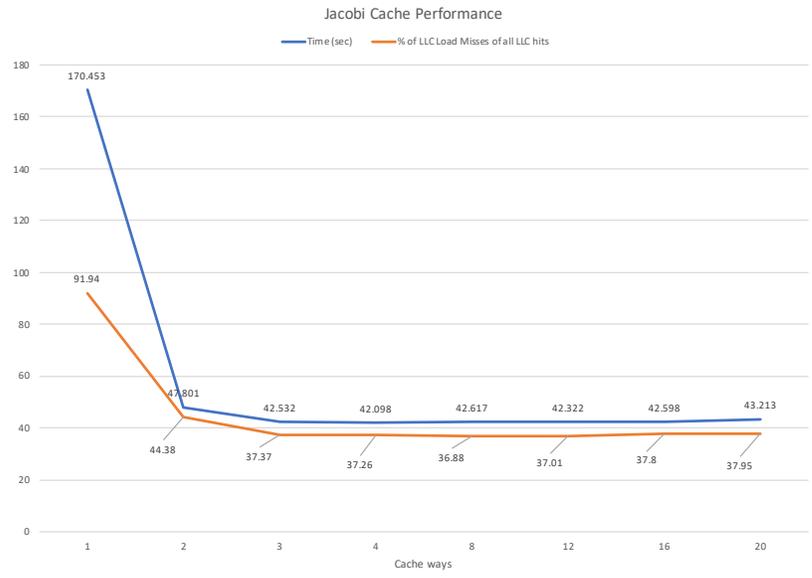


Figure 3.3: Performance of Jacobi with 20 MB L3 cache.

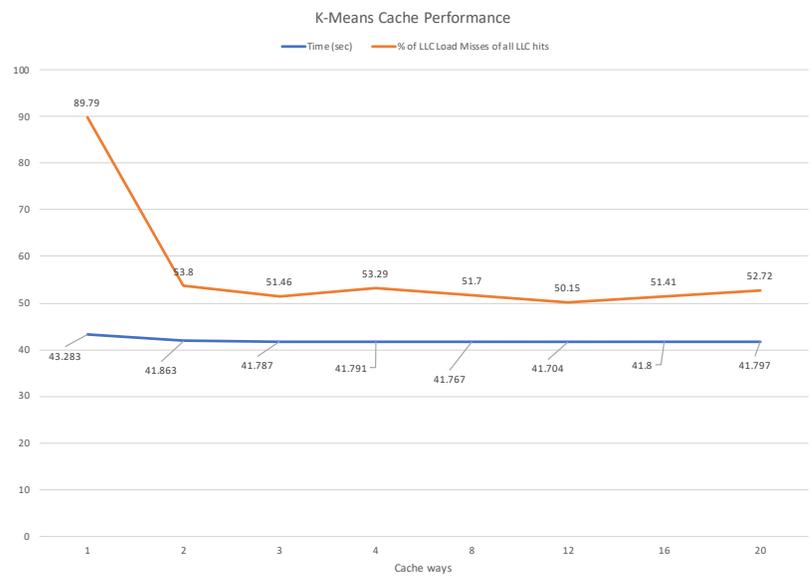


Figure 3.4: Performance of K-means with 20 MB L3 cache.

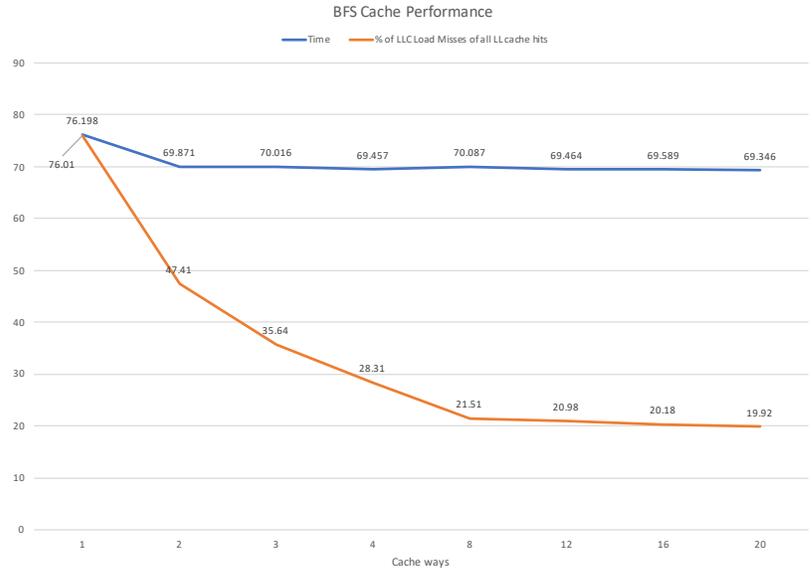


Figure 3.5: Performance of BFS with 20 MB L3 cache.

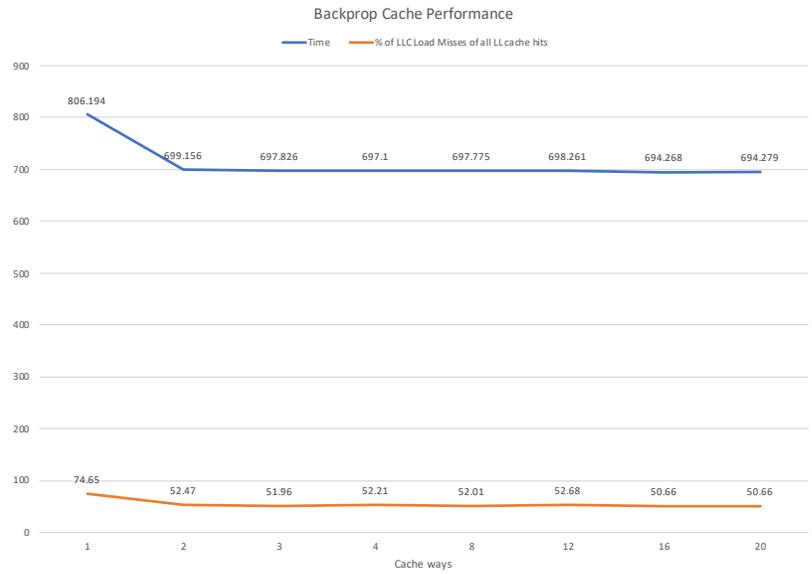


Figure 3.6: Performance of Backprop with 20 MB L3 cache.

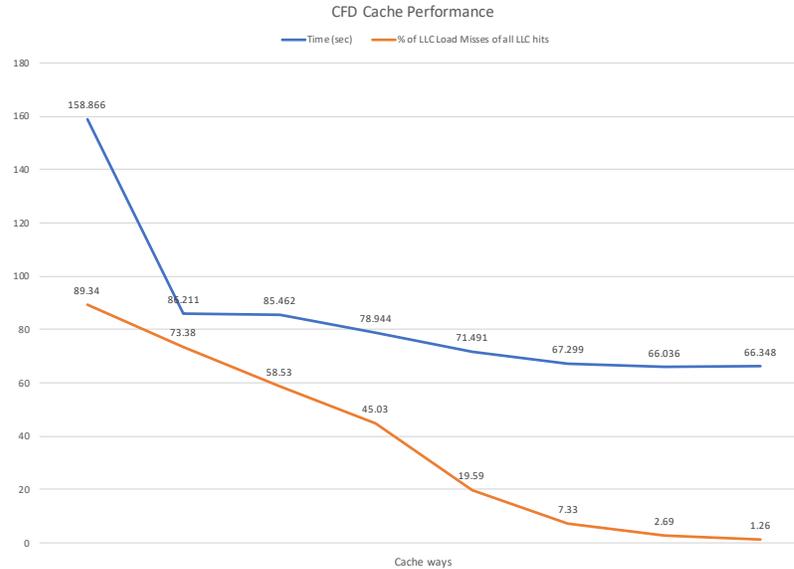


Figure 3.7: Performance of CFD with 20 MB L3 cache.

While we can see the same trend in Figure 3.5, Figure 3.6, and Figure 3.7, the magnitude of decrease in execution time is minimal. On the other hand, the execution time stays the same in Figure 3.4. We can conclude that the performance of this workload doesn't depend on the L3 cache.

For our following experiments, we choose Jacobi as our sole workload. We would like to learn the level of cache interference that each workload can handle. In reality, we would have two competing and co-located workloads A and B, where each of them are assigned a base cache capacity. As A's performance goes down, to maintain its SLO it will try to access the cache lines associated with B. This may or may not influence the behavior of workload B. The aftermath fairly relies on the type of workloads. A positive scenario would be when a cache-intensive workload and I/O intensive workload are co-located. The constant interference from cache-intensive

workload on the other workload wouldn't have a deterring effect on the performance. However, if both workloads rely majorly on their L3 cache, it wouldn't be wise to co-locate them. In practice, we cannot profile each workload and accordingly allocate them on the servers.

To study the effects of the above-mentioned scenario, we use a base state with 4 MB L3 cache and polluted state with 1 MB L3 cache. This setup emulates the incessant intervention from other competing workloads.

- **Timeout:** 100 seconds
- **Arrival Rate:** 0.75, 0.95 % of Service Rate
- **Service Rate:** 0.0238
- **Base cache State:** 4 MB
- **Polluted Cache State:** 1, 2, 3 (MB)

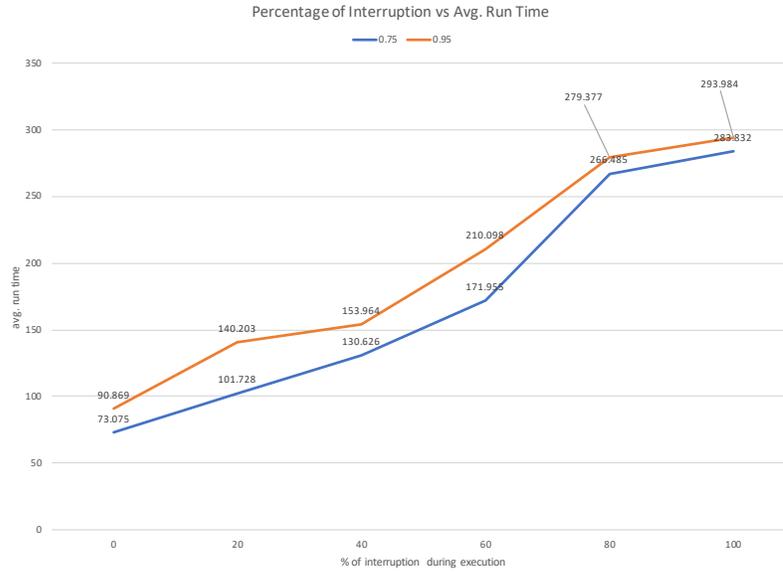


Figure 3.8: Percentage of Interruption vs Avg. runtime for Jacobi with base cache state as 4 MB and polluted cache state as 1 MB.

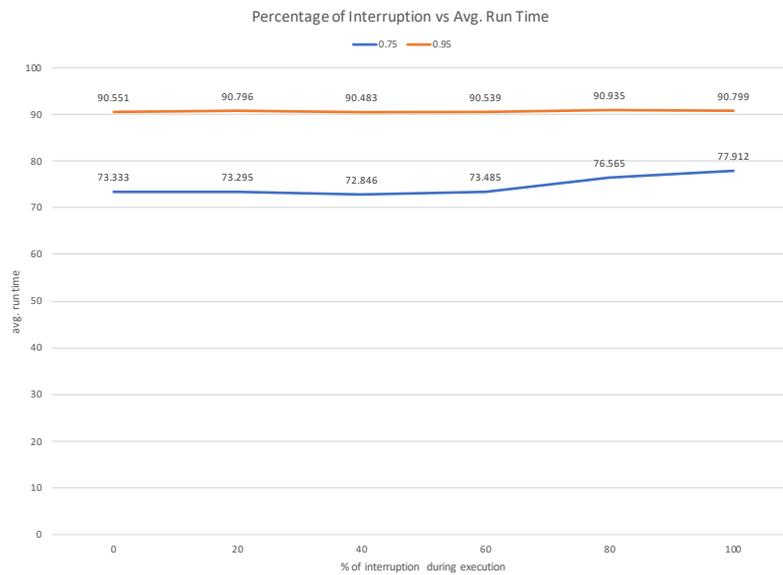


Figure 3.9: Percentage of Interruption vs Avg. runtime for Jacobi with base cache state as 4 MB and polluted cache state as 2 MB.

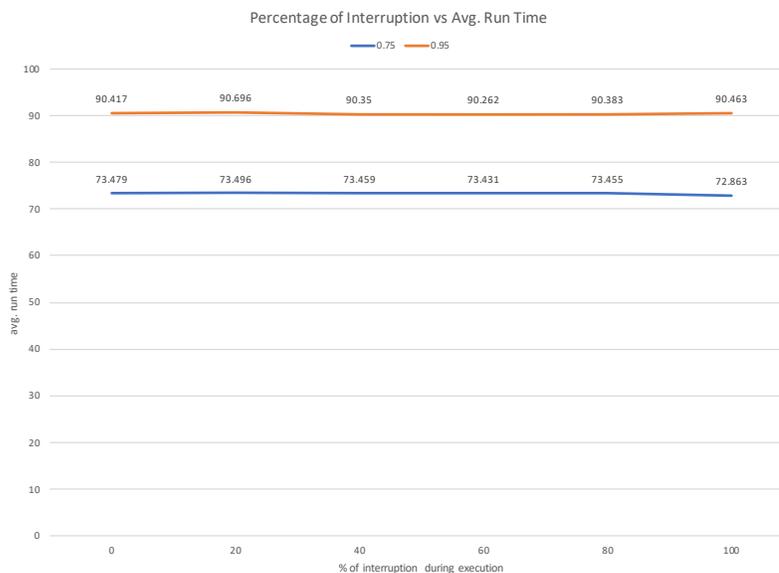


Figure 3.10: Percentage of Interruption vs Avg. runtime for Jacobi with base cache state as 4 MB and polluted cache state as 3 MB.

From Figure 3.8, we can see that as the percentage of interruption during execution increases, the average runtime of the workload increases. This trend is visible when the base cache state is 4 MB and polluted cache state is 1 MB. However, when polluted cache states are 2 MB and 3 MB, the increase in the average runtime of the workload is minimal. Also, as arrival rate increases, the average runtime accordingly increases. When base cache state and polluted cache state are 4 MB and 1 MB, we observe an average of 41.37 seconds increase in runtime of Jacobi workload for every 20% increase in interruption from other workloads. We can conclude that for a specific workload, on offline profiling we can understand the effect of L3 cache on its performance. As we mentioned before, this is not feasible in practice. For future work, we propose a machine learning approach to accurately map new workloads with cache ways. This approach is inspired by positive results from the work of Nathaniel et. al [22]. To

implement the machine learning model, we collected data from the following setup. Each server-worker unit has the ability to be launched as a separate Docker instance. The last-level cache is shared between all docker instances. We vary the arrival rate, timeout, cores, cache, and budget. The bulk data from these experiments is fed into a Random Decision Forest for training phase. The bias is greatly reduced while using random decision forests as they build deep trees.

## Chapter 4: Study on Service-Level Objectives

### 4.1 Introduction

Service Level Objectives (SLOs) are the fundamental constituents of Service Level Agreements (SLAs). SLA is an agreement between the customer and service provider. This is pre-established to avoid confusion or disputes between both parties. Quality, availability and responsibility are some of main aspects that are discussed quantitatively in SLOs. The common metrics in practice are Abandonment Rate, Average Speed to Answer (ASA), Time Service Factor (TSF), First-Call Resolution (FCR), Turn-Around Time (TAT) and Mean Time To Recover (MTTR). The result of not meeting SLOs would result in financial loss for the provider. For example, if the monthly up-time percentage drops below 95% in a month for a customer of Google Cloud, Google has to refund 50% of monthly bill for the respective service [12].

SLOs are integral to cloud services. They are used in networking infrastructure, micro-services, data centers and cloud applications. As the adoption of SLOs is increasing in the market, system managers are required to set up performance goals for SLOs. This serves as a motivation to study common practices and misconceptions in the design of SLOs.

## 4.2 Analysis and Discussions

We collected 9,634 documents that contained SLOs. By applying a Systematic Literature Review, we were able to come up with the following analysis. After finalizing features of importance such as response time, delay and reporting period, we were able to trim our dataset by eliminating unwanted SLO documents. Removing outliers was also a means of cleaning our dataset. By the end, we had 75 SLOs with 34 being academic sources and 41 being industry sources.

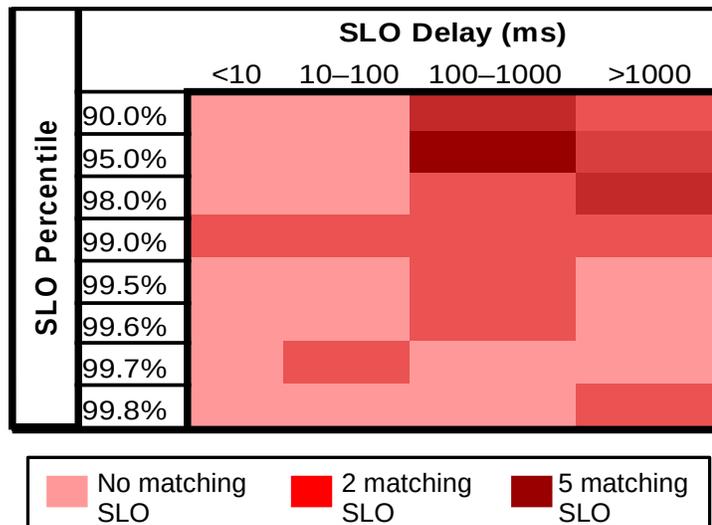


Figure 4.1: Heat map of delay and percentile goals from industry sources.

**Single-digit response goals are challenging.** From Figure 4.1, we can observe that there is only one matching SLO where the delay is lesser than 10 ms. This shows that achieving single-digit response goals is a challenging task. The growing availability of hardware devices that work in the microsecond range has instigated the “era of the killer micro-second” [3]. Improvements in redesigning the system stack

and modifying traditional low-level system optimizations to be micro-second aware will help in accomplishing single-digit response goals.

**Black swans prohibit extreme percentiles for complex software.** We can also observe from the lower half of Figure 4.1 that we don't have a majority of SLOs reported in the range of 99.8% and 99.5% percentile. Extreme SLO percentiles are not viable for complex software systems as even scarce occurrences of black swans could violate the SLOs. It is important to note that this observation is valid only for software systems that require high computing and interaction with hardware devices. Since black swans are correlated, it makes it even harder to have intense SLO percentiles.

**Evaluation of complex cloud services should vary percentiles.** Based on the SLOs collected across industry and academia, we observed that SLO percentile parameter varies in a scattered fashion from 90.0% to 99.9% for both types. On the other hand, the SLO percentiles for networking services is concentrated around 99.0 and 99.9%. We can infer that to evaluate complex cloud services, varying percentiles is advisable.

**Evaluation of infrastructure should vary the delay.** We observe that the delay parameter is distributed all across the range for the SLOs of networking services in academia and industry. Contrarily, the delay parameter is gathered around the 128ms region for application services. Hence, whilst devising SLOs for networking services, varying the delay is essential.

**We need evaluation tools for long reporting periods.** The SLO timeframe parameter specifies the time between two consecutive checks for SLO violations. The timeframe is fixed based on the complexity and type of service being offered. Complex services have reporting periods in days and months. For microservices, reporting

periods are in the order of seconds. However, lesser than 15% of SLOs have reporting periods in the range of 5 - 10 minutes.

## Chapter 5: Conclusion and Future Work

Our work has broadly connected the various aspects of computational sprinting. First, we developed a sprinting mechanism to speed up web queries. This was achieved by implementing a chrome extension and the Alexa Top 500 webpages were used for experiments. Secondly, we addressed the problem of over-provisioning workloads in the industry by devising a system that employs the Intel Cache Allocation Technology tool to dynamically allocate L3 cache. A workload profiler and queue simulator were used to realize the same. Lastly, a systematic literature review was carried upon SLOs in industry and academia. Multiple conclusions were drawn that will help system managers in fixing SLOs in the future.

## Bibliography

- [1] Ankur Agrawal et al. “Approximate computing: Challenges and opportunities”. In: *ICAC*. 2016.
- [2] Md Tanvir Al Amin et al. “Social Trove: A Self-Summarizing Storage Service for Social Sensing”. In: *ICAC*. 2015.
- [3] Luiz André Barroso et al. “Attack of the killer microseconds.” In: *Communications of the ACM* (2017).
- [4] Mike Belshe and Roberto Peon. “SPDY protocol”. In: (2012).
- [5] Peter Boncz, Thomas Neumann, and Orri Erling. “TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark”. In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2013, pp. 61–76.
- [6] Marco Brocanelli and Xiaorui Wang. “Smartphone Radio Interface Management for Longer Battery Lifetime”. In: *ICAC*. 2017.
- [7] Calin Cascaval et al. “ZOOMM: a parallel web browser engine for multicore mobile devices”. In: *ACM SIGPLAN Notices*. Vol. 48. 8. ACM. 2013, pp. 271–280.
- [8] Xin Chen et al. “GOVERNOR: Smoother Stream Processing Through Smarter Backpressure”. In: *ICAC*. 2017.
- [9] Intel Corporation. *Intel turbo boost technology in Intel core microarchitecture (Nehalem) based processors*. 2008.
- [10] Nan Deng et al. “Adaptive Green Hosting”. In: *IEEE International Conference on Autonomic Computing*. 2012.
- [11] Dalibor D Dvorski. “Installing, configuring, and developing with Xampp”. In: *Skills Canada* (2007).
- [12] *Google Compute Engine Service Level Agreement (SLA)*. 2018. URL: <https://cloud.google.com/compute/sla>.
- [13] Md E Haque et al. “Few-to-many: Incremental parallelism for reducing tail latency in interactive services”. In: *ACM SIGARCH Computer Architecture News* 43.1 (2015), pp. 161–175.

- [14] Andrew Herdrich et al. “Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family”. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2016, pp. 657–668.
- [15] Chang-Hong Hsu et al. “Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting”. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2015, pp. 271–282.
- [16] Eaton K. *How one second could cost Amazon \$1.6 billion in sales*. 2013. URL: <https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales>.
- [17] Jaimie Kelley et al. “Measuring and Managing Answer Quality for Online Data-Intensive Services”. In: *ICAC*. 2015.
- [18] Haohui Mai et al. “A case for parallelizing web pages”. In: *Presented as part of the 4th {USENIX} Workshop on Hot Topics in Parallelism*. 2012.
- [19] Rick Merritt. “ARM CTO: power surge could create ‘dark silicon’”. In: *EE Times, Oct* (2009).
- [20] James Mickens. “Silo: Exploiting JavaScript and DOM Storage for Faster Page Los.” In: *WebApps*. 2010.
- [21] Alan L Montgomery and Christos Faloutsos. “Identifying web browsing trends and patterns”. In: *Computer* 34.7 (2001), pp. 94–95.
- [22] Nathaniel Morris et al. “Model-driven computational sprinting”. In: *Proceedings of the Thirteenth EuroSys Conference*. ACM. 2018, p. 38.
- [23] Nathaniel Morris et al. “Sprint ability: How well does your software exploit bursts in processing capacity?” In: *2016 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE. 2016, pp. 173–178.
- [24] Ravi Netravali et al. “Polaris: faster page loads using fine-grained dependency tracking”. In: *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 2016.
- [25] Ajay Panyala et al. “Approximate Computing Techniques for Iterative Graph Algorithms”. In: *ICAC*. 2017.
- [26] Nathaniel Pinckney et al. “Shortstop: An on-chip fast supply boosting technique”. In: *2013 Symposium on VLSI Circuits*. IEEE. 2013, pp. C290–C291.
- [27] Arun Raghavan et al. “Computational sprinting”. In: *IEEE international symposium on high-performance comp architecture*. IEEE. 2012, pp. 1–12.
- [28] Daniel A. Reed and Jack Dongarra. “Exascale Computing and Big Data”. In: *Commun. ACM* 58.7 (June 2015), pp. 56–68. ISSN: 0001-0782. DOI: 10.1145/2699414. URL: <http://doi.acm.org/10.1145/2699414>.

- [29] S Ren and MA Islam. “Colocation demand response: Why do I turn off my servers?” In: *ICAC*. 2014.
- [30] S Ren et al. “Exploiting processor heterogeneity in interactive services”. In: *ICAC*. 2013.
- [31] Siva Renganathan et al. “Preliminary Results on an Interactive Learning Tool for Early Algebra Education”. In: *IEEE Frontiers in Education*. 2017.
- [32] *Rodinia: A Benchmark Suite for Heterogenous Computing*. 2018. URL: [http://lava.cs.virginia.edu/Rodinia/download\\_links.htm](http://lava.cs.virginia.edu/Rodinia/download_links.htm).
- [33] Jim Roskind. “QUIC: Multiplexed stream transport over UDP”. In: *Google working design document* (2013).
- [34] *StatCounter Global Stats: Browser, OS, Search Engine*. 2018. URL: <http://gs.statcounter.com/>.
- [35] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. “Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs”. In: *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13)*. 2013, pp. 265–277.
- [36] Christopher Stewart, Terence Kelly, and Alex Zhang. “Exploiting Nonstationarity For Performance Prediction”. In: *European Conference on Computer Systems*. 2007.
- [37] Christopher Stewart, Matthew Leventi, and Kai Shen. “Empirical Examination of A Collaborative Web Application”. In: *IEEE International Symposium on Workload Characterization*. 2008.
- [38] Christopher Stewart and Kai Shen. “Performance Modeling and System Management for Multi-component Online Services”. In: *Symposium on Networked Systems Design and Implementation*. 2005.
- [39] Christopher Stewart et al. “A Dollar From 15 Cents: Cross-Platform Management for Internet Services”. In: *USENIX Annual Technical Conference*. 2008.
- [40] Christopher Stewart et al. “Profile-driven Component Placement for Cluster-based Online Services”. In: *International Middleware Conference*. 2004.
- [41] Michael B Taylor. “Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse”. In: *DAC Design Automation Conference 2012*. IEEE. 2012, pp. 1131–1136.
- [42] *Top Sites in United States - Alexa*. 2019. URL: <https://www.alexa.com/topsites/countries/US>.
- [43] *Using Page Speed in mobile search ranking*. 2018. URL: <https://webmasters.googleblog.com/2018/01/using-page-speed-in-mobile-search.html>.
- [44] Cheng Wang et al. “Effective Capacity Modulation as an Explicit Control Knob for Public Cloud Profitability”. In: *ICAC*. 2016.

- [45] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. “Speeding up web page loads with shandian”. In: *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 2016, pp. 109–122.
- [46] Zichen Xu et al. “CADRE: Carbon-Aware Data Replication for Geo-Diverse Services”. In: *ICAC*. 2015.
- [47] Huazhe Zhang and Henry Hoffmann. “Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques”. In: *ACM SIGARCH Computer Architecture News* 44.2 (2016), pp. 545–559.
- [48] Kuangyu Zheng, Bruce Beitman, and Xiaorui Wang. “CoSmart: Coordinating Smartphone with Desktop Computer for Joint Energy Savings”. In: *ICAC*. 2017.