Characterization and analysis of a Web Search benchmark

Zacharias Hadjilambrou

Yiannakis Sazeides

TR-15-3 August 2015

University of Cyprus

Technical Report

Characterization and analysis of a Web Search benchmark

Zacharias Hadjilambrou

Yiannakis Sazeides

UCY-CS-TR-15-3

August 2015

ABSTRACT

Web search as a service is very impressive. Web search runs on thousands of servers which perform search on an index of billions of web pages. The search results must be both relevant to the user queries and reach the user in a fraction of a second. A web search service must guarantee the same QoS at all times even at the peak incoming traffic load. Not unjustifiably the web search service has attracted a lot of research attention. Despite the high research interest web search has gained, there are still plenty unknown about the functionality and the architecture of web search benchmarks. Much research has been done using commercial web search engines, like Bing or Google, but many details of these search engines are, of course, not disclosed to the public. We take an academically accepted web search benchmark and we perform a thorough characterization and analysis of it. We shed light in to the architecture, the functionality and micro-architectural behaviour of the benchmark. We also investigate some prominent web search research issues. In particular, we study how intra-server index partitioning affects the response time and throughput, we explore the potential use of low power servers for web search and we also examine the potential causes of performance variability and tail latencies. Some of our main conclusions are: a) intra-server index partitioning can reduce tail latencies, b) low power servers given enough partitioning can provide same response times as conventional high performance servers, c) web search is a CPU intensive cache friendly application, d) Background process and power management schemes are main culprits for causing performance variability in web search.

Table of Contents

Table List

Figure List

Chapter 1

Introduction

1.1 Motivation

Web search is a wide used service. Web search must provide quickly relevant search results otherwise this will have negative impact on the search engine's revenue [21,8]. Web search runs on thousands of servers to provide tight QoS guarantees, e.g. tail latencies below 300ms, even at peak traffic loads.

Not surprisingly a lot of research has been aimed at improving the performance, efficiency and cost of operation of a web search service. In [15] the web search is used as an example of an interactive service that can benefit from full system power management. The possibility of using mobile cores for web search for improved cost and efficiency is studied in [19]. [20] examined how web search can benefit from heterogeneous cores while [10] looked at adaptive parallelism for improving response times. Other work has collocated web search with other type of datacenter workloads to explore various optimization options for datacenters [13,6,14]. Virtually all these earlier works are done using Google or Bing web search engines for which implementation details are not publicly available.

Some recent work [6,18] has been performed using the publicly available Nutch Web Search Benchmark of CloudSuite [5], but no previous work has discussed its generality, functionality or has otherwise performed a thorough characterization of the benchmark. The only previous benchmark characterization [5] is a low-level hardware characterization giving insights about its performance profile (ipc, cache misses, memory utilization, etc.).

In this thesis we perform a higher level characterization giving broader insights of the benchmark behaviour at the application level. We then explain the application level behaviour by doing micro architecture characterization. Our work tries to shed light on the architecture, functionality and possible weaknesses of the Nutch Web Search benchmark. This will help the research community to better understand the benchmark and, consequently, learn how to best evaluate and optimize this important benchmark. The specific contributions of our work are: 1) discuss the details of the architecture and the interactions between the basic components of the Nutch Web Search benchmark, 2) explain the functionality of each basic component of the benchmark, 3) perform a characterization of the input query stream, 4) carry out a characterization of the benchmark in real hardware deployment, 5) show that low performance/power index-servers with enough workload partitioning can reach the response times of a high performance/power server, 6) demonstrate that intra-server partitioning can reduce tail latencies, 7) study sources of performance variability, 8) perform micro architectural characterization and correlate the application level behaviour with the micro architectural behaviour, 9) perform prefetching characterization and last level cache size sensitivity analysis. Our work, unlike many previous research on web search [10, 23, 20, 19], does not limit its focus only on the index search component of web search, but rather considers the whole end-to-end execution of a query, including the search results' url and title fetch and summary generation. This enables a more comprehensive view and deeper insight about this benchmark.

The rest of the thesis is organized as follows. Chapter 2 provides a background on related work and terminology that will be used in the rest of the thesis. Chapter 3 describes the Nutch Web Search benchmark. Chapter 4 presents the experimental setup and discusses the experimental methodology. Chapter 5 presents several studies to characterize the application level performance behaviour of the benchmark. Chapter 6 presents a micro-architectural characterization. Chapter 7 provides a general discussion about the similarity of Nutch benchmark to commercial search engines. This thesis concludes in Chapter 8.

1.2 Thesis impact

A large part of this thesis was published in [43]. Also during this thesis a framework for evaluating web search was developed which is now used by the Computer Architecture Xi Group [45]. This framework was used in [44].

Chapter 2

Related Work and Background

This Chapter presents an overview of the scientific work preceding this study and a definition of the basic concepts and terms will be given.

2.1 Related work

The Nutch web search benchmark was introduced as a part of Cloudsuite [3] in [5]. In that paper the authors performed a micro architectural characterization of the benchmark. They results showed that: 1) Web search had relative high instructions per cycle (IPC) in comparison to the other Cloudsuite benchmarks, 2) web search performance did not benefit much from increased cache capacity or prefetchers. Our main differences with [5] are: 1) We perform both micro architectural as well as application level performance characterization and we correlate the application level performance with the micro architectural behavior, 2) we observe similar results to [5] but we explain those results by giving insights in the functionality and implementation of the web search benchmark.

Xi, Huafeng, et al. [27] used the same benchmark for characterizing real web search engines query traces. The used both real life queries as well as real life per query inter-arrival times. They compared the performance characteristics of real life inter-arrival times with synthetic poison distribution inter-arrival times. They conclude that poison distributions do not reflect accurately the real-life patterns. In our work we evaluate this benchmark using stress testing. Stress testing does not reflect real life situation but it is a reliable way to see how system is performing under the maximum load. In [27] authors also perform micro architectural characterization by showing IPC and caching behavior and their findings agree with our work [5]. In this work we compare the characteristic of different query traces but only from the point of view of the query content not the query inter-arrival time as in all experiment we use stress testing.

C. Badue et al. [1] compared the performance of document partitioning with term partitioning. Their results showed that term partitioning outperformed the document partitioning. In this work we use document partitioning and we perform a characterization of different degrees of partitioning. Term partitioning is not supported by the Nutch Web Search Benchmark.

Ren, et al. [20] evaluated web search performance on heterogeneous processors. They simulated and compared heterogeneous (combination of high power and lower power cores) and homogenous configurations under the same power budget. They showed that by scheduling small requests to lower power cores and long request to high power high performance cores they obtained performance gains. In this work we also aim in power efficiency by employing partition and use of many low power cores versus using fewer higher performance cores.

Pejman et a.l [18] proposed a processor chip design which employs many low power cores. This design reduces the size of caches to make room for more cores to improve the computation power per square millimeter. This is opposite of the prevalent typical server chip design which uses large high performance complex cores and large size caches. They show that their design improves throughput for web search and the other cloud workloads. They do not mention anything about the tail latencies which is an important aspect of the web search performance. In our work we consider both throughput and tail latencies as performance metrics.

Reddi et al. [19] examined the possibility of using mobile cores for Bing web search engine. They show that mobile cores cannot reach the response times achieved by the conventional high performance cores. They propose some micro architectural enhancements which may help the mobile cores reaching the desired performance levels. Some of the proposed enhancements are larger caches and better branch predictors. Indeed the 1MB L2 cache of the mobile cores used in the paper was too small to capture the necessary working set. In our work we also aim for optimizing web search energy efficiency by using lower power cores. We employ partitioning for improving the low power cores response time whereas [19] did not examined partitioning and parallel search for improving low power cores performance.

Jeon et al. [10] characterized intra server partitioning performance for Bing search engine and proposed adaptive parallelism. Intra server partitioning refers to running more than one index servers per physical machine. Each index server performs search on a disjoint part of the index dataset and this way parallel search is achieved. They proposed employing high parallelism when the utilization is low and low parallelism when utilization is high. In this work we also examined intra server partitioning. Unlike [10] we concluded that parallelism has benefits both on low and high utilization scenarios. On low utilization scenario partitioning is clearly better and for high utilization scenario it sacrifices throughput over better tail latencies.

Finally [29] characterizes the sources of variability and long tail latencies for different server workloads. They showed that by increasing process priority, by removing interrupts and by removing power management features they manage to close the gap between the observed and the theoretical ideal latencies. We perform a similar study but for the web search benchmark.

2.2 Definitions

Term: A word that a web search user sends to a web search engine as a part of query.

Query: Expression which defines what documents a web search user is searching for. Queries are composed of one or more terms. In this evaluation we consider only conjunctive queries e.g. the query "CS courses" means that the user is asking only for documents that contain both the terms CS and courses.

Hits: The document matching a query.

Term Frequency: The amount of appearances of a term in a document. A document has a term frequency number for every distinct term located in the document.

Document Frequency: The amount of documents that contain a term. Each term has a document frequency.

Index: A structure holding the terms and the relevant to a term documents. Each term points to a list of all documents that contain the term. The list contains pairs of document IDs and term frequencies (<docId,termFrequency>). The terms are alphabetically sorted to enable term search in a logarithmic time. The document frequency of each term is also stored in the index. Figure 1 shows a high level overview of the index.

Posting list / Document list: The list of <docId,termFrequency> pairs where a term point to. A pair is called a posting. We will refer to the terms posting list and document list interchangeably.

Tf.idf scoring: A document scoring scheme which quantifies a document relevance to a query. It takes in account the term frequency and the inverse document frequency of each term.

Top k docs/top k hits: The top k most relevant to a query documents.

Throughput: For web services throughput is the amount of requests a server can serve per amount of time. In our case for web search throughput will be measured in executed queries/second.

Response time / Query Latency: Is the time measured from the moment a client sends a request to a web service until the answer arrives back to client. Response times of single queries are very important for web search as long queries have negative impact in user experience and in a search engine's revenue [21,8]. Thus web search engines define their Quality of Service target in terms of response times. We will refer to the terms query latency and response time interchangeably.

Quality of Service and tail latencies: Tail latencies for web search and in general interactive web services are measured with various percentiles. For example a 99th percentile equal to 300ms means that the 99% of executed queries are executed in less than 300ms. Interactive web search services usually set quality of services targets (QoS) in terms of percentiles of response times.

Partitioning and Parallel search: Distribution of the index to different index servers to enable parallel index search.

Intra server partitioning: Parallel search on a single physical machine.

Dynamic voltage frequency scaling (DVFS): Technique applied in modern microprocessors for power savings. Each modern microprocessors has predefined voltage, frequency states also called performance states [33]. A user space process according to CPU utilization changes the cpu performance states. Usually on low utilization low power low performance is preferred and on high utilization a high power high performance state is preferred.

Core parking: Usually referred to very deep low power idle states (e.g. C6,C7 [33]). Core parking provides significant power savings but a core in core parking state requires significant amount of time to return to operational mode (C0).

Soft page faults: Page faults which do not cause disk accesses [38]. Some of the causes of soft page faults are: a) the page was brought by other processes into memory, b) the page was deallocated by the process but not yet written to disk.

Datacenter: A large facility which houses computer systems for providing compute and storage resources.

TCO: A datacenter's total cost of ownership. Usually broken down in capital (e.g. building, server acquisition) and operating (e.g. power cost) expenses [7].

Term	Document Frequency	<Document Ids, term Frequency>
Architecture	100	<1,1>,<2,3>,<3,1>,<4,100>,<5,1>,…
Courses	3	<1,2>,<2,1>,<4,1>
CS	4	<1,1>,<2,3>,<5,5>,<9,7>
……………	….	……………..

Figure 1 High level view of the index organization. Each term points to the documents which contain the term. The terms are alphabetically sorted to enable term search in logarithmic time. Term frequency specifies how many times the term appears inside the document.

Chapter 3

Benchmark Description

In this chapter we describe the benchmark's architecture and functionality. We also describe the benchmark inputs and the benchmark output metrics.

3.1 Architecture and functionality

The Nutch Web Search benchmark consists of four main components that are illustrated in Figure 2 with a diagram which describes the overall architecture and information flow. Namely the four components are: the client, frontend server and multiple index and document servers. The index may be partitioned across many index servers so that each index server gets to hold a disjoint part of the index. Documents servers hold the actual documents and they are used for fetching the summaries of the search results. The flow of executing a query goes like this:

1) The client sends a query to the frontend server.

2) The frontend receives the query and asks from each index server to return the most relevant to the query documents.

3) The index servers perform the search and respond to frontend with the document Ids and the relevance scores of the top-k relevant matching documents.

4) The frontend collects the results and sorts the documents according to their relevance score. In this step the fronted performs a check for duplicate results. The default configuration of the Nutch benchmark allows no more than two results from the same site. Depending on the query, steps 2 to 4 may repeat multiple times until the frontend is satisfied with the number of duplicates per site. At each repetition the server asks two times more search results from each index server. These subsequent searches are referred to as optimization searches.

5) After the frontend has the final top-k results, it sends a detail request to each index server whose search results are in the current top-k list. The index server responds to a detail request with the title and the url for each request.

6) As soon as the frontend has all the title and urls of the top-k results, it asks from the document servers the summaries of the top-k results. The frontend is aware of which documents each document server holds. Consequently, only the document servers which hold the documents that are in the top-k results are being asked for summaries.

7) The document servers generate the summaries and send them to the frontend.

8) When the frontend receives the summaries it assembles the final html response and sends it to the client.

We provide next a more detailed per component description.

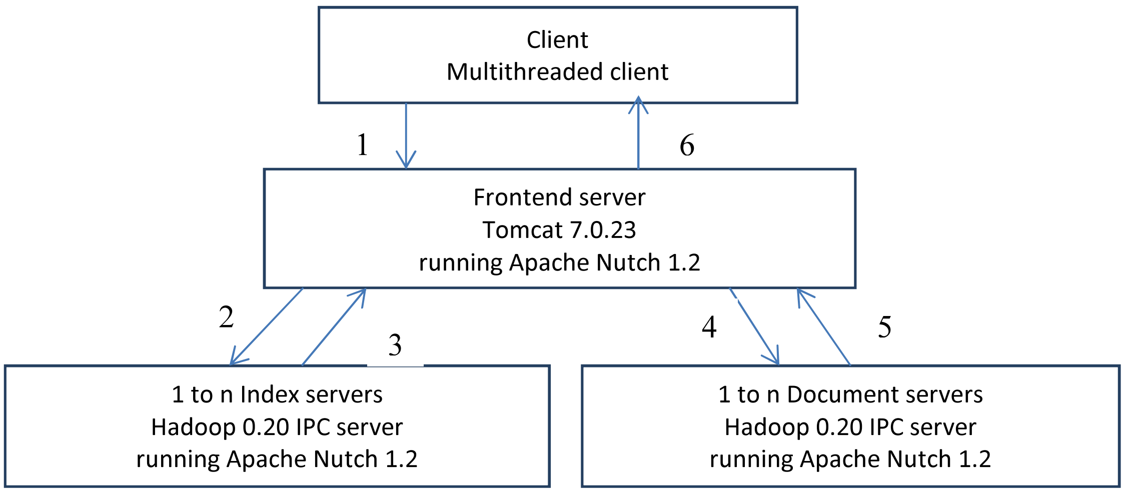


Fig. 2. The basic components of the Nutch Benchmark. The arrows show the information flow and interactions between components during a query execution. The numbers show the order of the execution flow.

Index Server The index server is a Hadoop 0.2 IPC (Inter Process Communication) server process running Lucene 3.0.31 search engine. The Hadoop IPC server consists of a listening thread which listens for incoming requests from the frontend server, the handler threads which actually perform the search or retrieve the details of a document and a responder thread for sending the responses to the frontend. The handler threads number can be set by the user and it's recommended to be slightly higher than the number of available cores [3].

The Nutch benchmark uses document partitioning, which means each index server holds an index for a disjoint set of documents. Another partitioning technique is term partitioning

[2]. In term partitioning each index server holds a disjoint set of terms. In term partitioning only the index servers that have query terms are involved in the search procedure. The index of our benchmark is a result of a crawling process which runs upon hadoop map reduce implementation [3]. Thus the index is stored as multiple partitions similarly to a typical hadoop map reduce output. Each partition stores a disjoint part of documents.

The index terms are stored in an array in alphabetical order. The alphabetical ordering enables binary search for fast term searching. A parallel array hold the pointer to a byte stream; each pointer points to the position where the <documentId,termFrequency> pairs of a term start. The documentId and termFrequency pairs are compressed using a variable integer format [42]. The idea of the variable integer format is that the first bit of each byte shows whether more bytes are remaining (if bit is equal to 1 more bytes are left otherwise no bytes are left to read). This way all numbers from 0-127 can be represented with one byte all number from 128 to 16383 with two bytes and etc. The document list can be sorted by score [12, 10], for example the PageRank [17]. Sorting the docs by score is beneficial both for performance and for relevance of results. It provides quick access to the most popular documents which are most likely to fit the user's information needs. In the Nutch benchmark the documents are sorted by DocId. Sorting docs by ID is useful for performing efficient merging of posting list for Boolean AND queries [12]. This is how the Lucene and consequently Nutch benchmark implements the index organization which is shown in Figure 1.

The Nutch benchmark uses for the search procedure a combination of Boolean retrieval, vector space model for representing documents and tf.idf weighting scheme for ranking documents [11]. Nutch allows only conjunctive queries (AND) for multi-term queries. The actual search procedure goes like this: first Nutch tries to find all documents which contain all terms of a multi-term query; then for each document found, a tf.idf score is calculated. The tf.idf weighting scheme gives high scores to documents which have many occurrences of the query terms and also to documents that contain many occurrences of rare terms. Each index part is (recall we mentioned earlier that our index consist of many disjoint parts) searched independently and all matching documents are fed to a scorer object which accumulates the document relevance scores and sorts the document ids by their relevance score. The pseudo code in Figure 3 gives an overview of the index search procedure for a single term query.

Web search engines usually perform an early termination of the search procedure either by using a cut-off latency [19] or perform an early termination when the quality of results is unlikely to improve with further searching [10]. Early termination is used to avoid having an index server to search for too long. The Nutch benchmark provides the option to stop searching based on cut-off latency or after a number of matching documents is found. We did not use any of these options for our evaluation and this is the default configuration of the nutch benchmark.

```
For each index part
    indexToPointer=binarySearch(term,termsArray)
    seekToPointer(pointers[indexToPointer])
    while document List Not Over
        docId=nextVariableInteger()
        termFrequency=nextVariableInteger()
        scorer(docId,termFrequency)
```

Figure 3. The pseudocode of the index search. For each index part a binary search is performed to find the byte at which a term's posting list starts. A seek operation is performed for going to the start of the posting list. After that the docId and term frequency of each posting are decoded and fed to a scorer object. The scorer accumulates the score for each document and maintains a priority queue for sorting the documents by score.

Two other parameters which are important for the index search are: 1) the amount of index dataset an index server holds, and (2) the number of index servers used. The larger the dataset an index server holds the longer its search time. However, by increasing the number of index servers we can reduce response time. Doing so increases the degree of partitioning which means each index server will get smaller index part thus will need less amount of time to respond to a query. Of course it is important the posting lists to be balanced across the partitions otherwise partitioning will not provide the expected performance benefit. Index partitioning can be done across server or inside the same server (intra server partitioning) [10]. Intra server partitioning can be applied by running multiple index search contexts on the same server machine with each context working on a different index part. The intra server partitioning represents a trade-off between throughput and response time latency. Having many index searchers on a CPU socket speedups the execution of a query but reduces the number of available cores for handling parallel requests.

Document Server The Document server, like the Index server, is also an instance of Hadoop IPC server with a listening, a responder thread and many handler threads. The document server contains the actual copies of the documents. The document server is used for fetching the summaries of web pages. In web search engines the web pages summaries can be dynamic or static [12]. Static summaries are preloaded and they are always the same regardless of the query that hit the document, while dynamic summaries are query dependent and they attempt to explain to the user why the document is retrieved for the query. The Nutch benchmark uses a dynamic summarizer. The dataset of the document server, like the index, is partitioned in many parts. Each part contains a disjoint set of documents. For expediting the access to the documents content, the document server uses a partitioning function (which takes as input the document url) for determining in which partition the document is located. A document server partition contains <url ,web page content> key value pairs and a small index which points to a fractions of keys which further expedites the summary generation procedure.

Frontend Server The frontend is a Tomcat web server running the Nutch application. Tomcat is multithreaded and spawns a new thread for handling a new query request. Frontend coordinates the entire query execution and it is the component which acts as a link between the client and the nodes which do the actual job: the index and the document servers.

Client The Client is a multithreaded process with each thread sending queries to the frontend. The client threads can send queries based on some inter-arrival distribution trying to mimic real users or queries can be sent in a stress test manner. In the stress test scenario a client thread sends a new query as soon as it receives the response for the previous query sent. Analysis of the performance while increasing the number of clients can be used to determine the maximum throughput capabilities of the system under test. In our evaluation we use only stress-test traffic.

Having discussed the architecture and the information flow of the benchmark we next describe the benchmark inputs, i.e. the queries.

3.2 Inputs

Web search engines can handle various types of queries. The most typical types of queries are multi-term queries. For a multi-term query search engines try to give a higher score or only consider as relevant document which contain many terms of the query (e.g. 3 of 5) [12]. Some search engines allow multi-term queries to be combined with Boolean expressions like OR, AND, NOT. Other types of queries are: (1) phrase queries which try to find documents which contain a phrase specified by user, and (2) proximity queries for terms within a specified distance. Phrase queries and proximity queries can be implemented with a positional index which holds the position of each term in a document. The Nutch benchmark considers all multi-term queries as AND queries and it can also execute phrase queries.

The processing time for a conjunctive query is determined by four factors: 1) the merging of the term posting lists, 2) the ranking of documents which come out of the intersection, (3) the number of optimization searches for duplicate elimination, and (4) whether the query needs detail and summary request (no need in the case with no matches). For testing a web search engine we can use a real traffic query stream or generate random queries built using index terms.

3.3 Metrics

A web search engine must maintain low mean and high-percentiles response times. Service guarantees as tight as 99% of response times within 300ms are usually set to keep users satisfied. Such service guarantees must be preserved even at the highest (peak) loads [4]. Relevance of the search results is also a crucial factor which contributes to user satisfaction and the eventual success of a search engine. The relevance of search results can be improved with more sophisticated ranking functions and larger web index [10].

3.4 Performance variability and tail latency sources

Web search engines in real deployments perform index search across thousands of servers in parallel. In such grand scale performance variability is much more likely to happen. A single node can slow down the whole query execution [2] with negative results to user satisfaction and thus negative impact in search engine's revenue [8], [21]. Web search as a service also suffers from implicit performance variability as some queries naturally take longer time to execute from the average case. These queries plus the various sources of performance variability are the main sources of the high tail latencies. Web search providers are trying their best to keep the tail latencies lower as possible. For example a typical quality of service guarantee for a web search service could be 99% of queries under 300ms. All the aforementioned facts give us a motivation for identifying and addressing the sources of performance variability and poor response times for web search engines. In this section we will mention commonly known sources of performance variability and we will evaluate how they affect the performance in section 5.6. Now we will present one by one the possible sources of performance variability we considered in our evaluation.

Core migration and Non Uniform Memory Architecture: The architecture of modern blade servers is usually the Non Uniform Memory Architecture (NUMA). A typical server consists of two sockets and each socket has its own memory node. Of course both sockets can access data from both memory nodes but accessing the remote memory node adds some performance penalty. It is typical practise in data centres to run more than one application per server node [6] or for QoS purposes do keep underutilized a server by not using all its cores [13]. Both practises require core and memory pinning of an application otherwise a running process can migrate from core to core depending on the Linux scheduler policy. The migration can cause loss in performance due to the fact that the architectural structures like L2 and last level cache of the core that a process is migrating in, needs time to warm up. Also the migration may cause memory accesses to remote node which of course induces some performance penalty and performance variability.

Power Management techniques (Core parking, DFVS): Deep sleep states like core parking and DVFS may hurt performance for various reasons. For example executing a query on a core which was previously idle may add significant amount of time to the query execution if that core is waking up from deep sleep state. The same problems applies to DVFS. The default Linux DVFS governor scales the frequency accordingly to utilization. A core starting query execution from idle state probably will need some time to scale the frequency up to the max. This may cause a longer query processing time in comparison to running always at the highest frequency.

Background jobs and interrupt processing: Background processes and services which are part of the Operating System as well as other user started background processes like for example monitoring processes may have performance impact on the web search performance. Our web search processes need to content with the background processes for CPU time. If a core is blocked by a background task for a long period of time, this greatly increases the latency of a query. Interrupt processing is also another variability source which is similar to background jobs as the CPU pre-empts the user process for executing other code. Real time priority scheduling as well as forcing interrupts to execute on other cores are techniques that seem to alleviate the performance variability caused by background jobs and interrupt processing [29].

Chapter 4

Experimental Details

This chapter will explain the experimental details of our evaluation. Subsection 4.1 will focus on the hardware and the benchmark configuration. Subsection 4.2 refers to the methodology we used for conducting the experiments and collecting statistics.

4.1 Hardware and Benchmark setup

For the experimental analysis we use dual socket blade servers based on Intel Xeon E5620 @2.4 GHz CPU. The processor supports frequency scaling with range from 1.6GHz to 2.4GHz. Table I provides the details of the blade server hardware. In all experiments we use 4 blade servers: one client, one frontend server, one index server and one document server. All machines are connected through 1Gb Ethernet.

Table 1. Server system parameters

Number of CPUs	2
CPU	Intel Xeon E5620 @ 2.4GHz
Number of cores per CPU	4
DRAM	32GB DDR3 1066MHz
Ethernet speed	1Gb

The client machine runs one multithreaded client process. We increase the number of concurrent requests by increasing the number of client threads. Frontend machine runs a Tomcat web server process which spawns a new thread for handling each new query request. Unless stated otherwise the index and document machines run one index server process and one document server process respectively. We keep the number of handlers of index and document servers equal to the number of clients. We constrain the index server process to run on a number of cores equal to the number of client threads, using the numactl command [16] for the experiments in Sections 4.1, 4.2 and 4.3. This is done because we obtained better performance with such configuration. For example, when running with 1 client we constrain the index server process to run on 1 core, with two clients on two cores etc. For cases with more than 8 clients we use 8 cores. We do not constrain the number of cores running the document server as the document server needs as much parallelism as possible. The reason is that the document server starts a new thread for generating each web page summary. Suppose there are eight concurrent query requests with 10 summaries to generate for each request, the number of simultaneous threads running on document server could reach 80. The servers are run with hyperthreading disabled to ease the analysis by avoiding performance differences arising from running in two different physical cores versus running on two logical cores mapped in the same physical core.

The experiments are run with the default benchmark configuration which does not use early query termination. This means the frontend always waits the index servers to finish searching and index servers search until they find all matching documents. Unless stated otherwise experiments use 100K queries from the AOL query log [24]. We are aware of the controversy surrounding this query log because of privacy issues. We want to state that we do not use the AOL log for people identification. We use the real life representative queries for performance characterization and analysis purposes. We also produce and compare results with a random query generator provided with the benchmark.

The faban client provided in the CloudSuite distribution does not run for same amount of work but for a fixed amount of time. In order to run all our experiments for the same amount of work we implemented a new multithreaded client for the experiments. We used stress testing for the experiments: each client thread sends a new query as soon as it receives the response for the previous one. For the index dataset we used the one which comes with the CloudSuite Simics images [22]. The dataset contains in total 18 parts of index and 18 parts of document server dataset (Segment dataset). The average index size part is 0.32 GB while the average segment size is 3.5GB. In total, we have 6.1GB index and 66 GB segment dataset. For all the experiments, unless stated otherwise, we are using a 5GB index and 50GB segment dataset. We choose to use the specific dataset parts and total size to make the index partitioning experiments more convenient. By convenience we mean that we can divide roughly equally the index dataset between 2 to 8 index server processes

4.2 Experimental Methodology

For our evaluation we had to run multiple experiments and each experiment we had to coordinate multiple processes across various physical machines and to collect various global or local relative to each physical machine statistics In this section we will explain the tools and the methodology we used for conducting our experiments.

4.2.2 End-to-end experiments

For conducting the end-to-end response time experiments we have written a bash script which was coordinating the execution of the experiment. This script used the ssh [31] Unix utility to connect to the various physical machines and start the frontend, the document and the index servers as well as the various monitoring utilities for collecting statistics. The client was executed at the machine that the executed script. After the client had finished with sending queries to the frontend server the script had to ssh back to all physical machines to shut down all servers. After shutting down all servers a post processing of the data was executed by the script to convert them to meaningful statistics. With the end of the data post processing the current experiment is considered over and a new execution of the script could be started for conducting a new experiment.

For collecting cpu, memory utilization statistics with use the top [30] Unix utility. We use the default refresh time which is equal to 3 seconds.

For calculating network statistics like upload and download rate we use a script which reads every second the /sys/class/net/eth0/statistics/rx_bytes and /sys/class/net/eth0/statistics/tx_bytes files. By calculating the difference between the current read value and the previous value you know how many bytes were uploaded or downloaded during the second passed.

For collecting performance counters we used the perf [32] unix utility. Perf allows the collection of the most popular micro architectural statistics like ipc, L1, L2, L3 cache misses, branch misspredictions etc. as well as microprocessor specific performance counters. Perf can collect performance counters for the whole process, for a specific thread or for a specific core.

Perf can also distinguish between user and system executed instructions. Perf can also attach to a running process for a specific amount of time or until the end of the process execution. We will explain in the next subsection how we exploited this perf feature for measuring per query performance counters.

Throughput, mean, 90th and 99th percentiles of end to end response times are collected by the client process. Per operation processing times e.g. index search times, detail retrieval, summary generation are recorded in the log file of the server which performed the operation.

In our experiments we have also explored different cpu frequencies. To achieve that we used the cpuspeed [34] user space process. Sending the -12 signal to the process forces a constant operation on the lowest frequency and sending -10 signal forces constant operation on the highest frequency. The -1 signal reset the DVFS operation to the default scaling strategy. Frequencies in between the min and max possible frequency can be set by setting the scaling_setspeed file which is usually located on this path /sys/devices/system/cpu/cpu0/cpufreq/. Changing DVFS settings and CPU frequency is only allowed for root users.

4.2.3 Per query experiments

As we will discuss in the next chapters of this thesis we conducted some experiments were we had to collect per query performance counters for the index search operation. In order to achieve that, we have to start and stop the perf monitoring at the start and stop of every query's execution. This is a not trivial job to do when you have a constant stream of queries arriving at inter arrival times of milliseconds. So we had to perform a controlled experiment where we would be sending one query at a time and we would attach the perf process to the index server a bit sooner than the query execution start and detach the perf process a bit later than the stop of the query execution. So we used a similar procedure like the one described in the previous subsection with the difference that this script uses the curl [35] Unix utility to send a query to the frontend server (because we had to send one query at a time) and attaches a perf process to the index server before the start of each query's execution. We time the perf execution by adding either the sleep or the usleep command at the end of the perf's command line. Basically this is the way to specify how much time the perf will be monitoring. You can use usleep if you want to specify millisecond scale. The trick is to put a larger than the query execution monitoring time so that to make sure all query related micro architectural events will be included. The question which arises is how representative will be the performance counters. Because this way we are monitoring for much more time than the query is running. We observed that when the index server doesn't execute queries it blocks on a system call which waits for incoming requests (similarly to the accept system call of C programming language) and during that time no instructions are executed. Figure 4 shows micro architectural statistics obtained for different perf monitoring times ranging from 1 to 4 seconds for queries which are processed in sub second time. The figure clearly shows that monitoring time does not affect the measurements thus with our methodology we obtain representative to the query execution performance counters.

We also had to find the actual thread of interest in order to get as representative performance counters as possible. For example, attaching to the whole java process would count unrelated to the query execution performance counter like listener, responder thread code and various other java related threads like garbage collectors and dynamic compilers. Thus we used the command "jstack –l processIDofTheIndexServer" which shows information about all threads our index server is running. This way we could get the Unix thread id (native id) of the handler thread which is the one which executes the queries. Knowing the thread id of the handler thread allowed us to attach the perf process only to this thread.

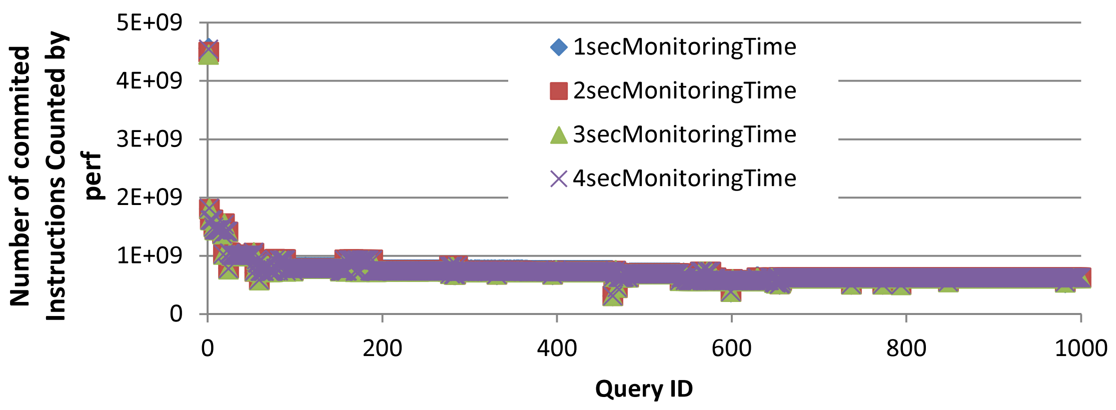


Figure 4 Instructions commited per query counted with perf for different monitoring times. Except very few outliers the instructions counted are not affected by the monitoring time. This proves that the index server sleeps the time is not processing a query thus our methodology of starting the monitoring before query execution and ending it after query execution, does not add irrelevant to the query execution micro architectural events.

Chapter 5

Characterization and analysis

5.1 Query Stream

The analysis in this section is obtained with one client thread sending requests. We do this to isolate the response times from the effects of queuing, contentions on shared resources etc. This way the analysis is focused only on the time actually needed to process the query. Figure 5 shows the cumulative frequency of the queries according to their number of terms. We have observed query lengths from 1 up to 72 terms. The short queries dominate the query stream. On average the query length is 2.67 terms.

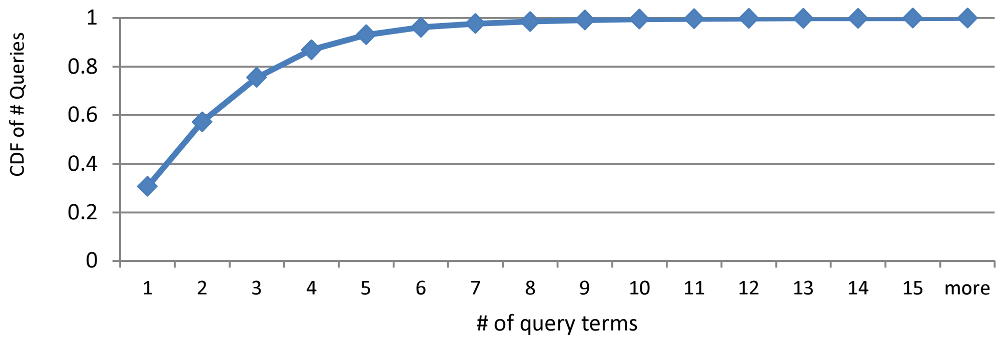


Fig. 5. Frequency of query lengths of the input query stream. The short length queries are the most frequent. Nearly 90% of the queries are less or equal to 4 terms and the average query size is 2.67 terms.

A cumulative distribution of frequency of response times is shown in Figure 6. The average response time is 22ms, the 90th percentile is 42ms and 99th is 107ms. Most queries are executed fast but they are few queries that require processing times of hundreds of ms. Figure 6 also shows the same distribution weighted by the response time of each query. This curve shows that queries with long response times are more prevalent. A noticeable fraction, around 6%, is shown to be due to queries with very long response times (more than 150ms).

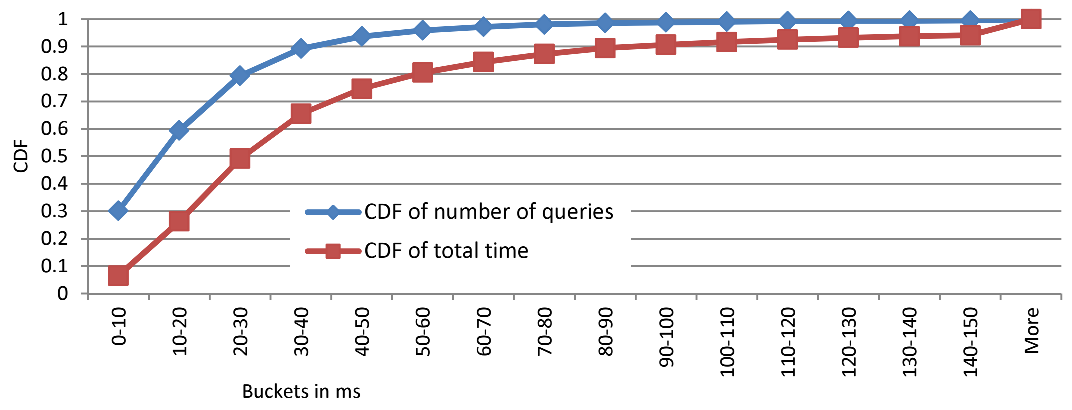


Fig. 6. CDF of the response times. Mean is 22, 90th 42 and 99th 107ms.

Figure 7 presents the correlation between the number of query terms and average end-to-end response times. The error bars represents the observed max and min for a given number of terms. The Figure also shows, on the secondary y-axis, the average number of documents matched for a given number of query terms. Please note that the y axis is logarithmic.

The results in Figure 7 clearly show an increase in the average response time with growing number of query terms. This behaviour is similar to what previous work has observed [23]. The behaviour of matching documents is a rather monotonic decrease with increasing query terms.

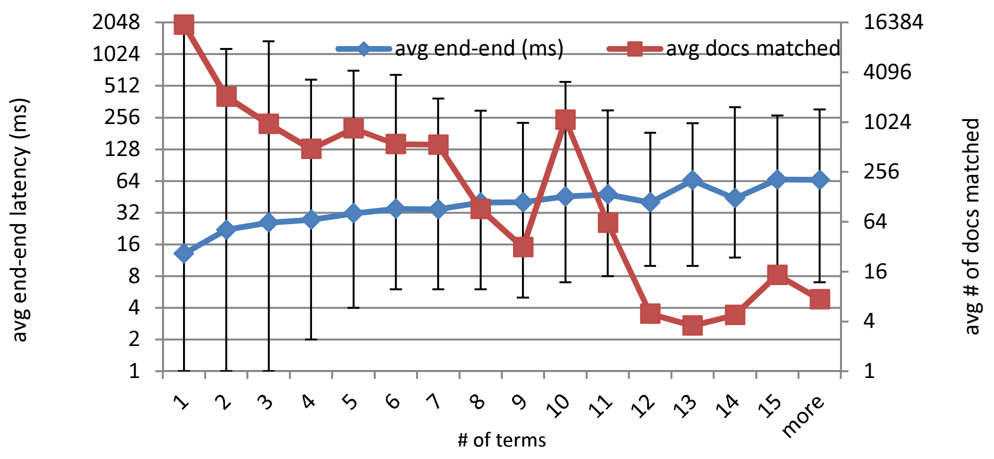


Fig. 7. Average number of response time and average number of matching documents as a function of the number of query terms. The response time increases with more terms whereas the number of docs matched decreases.

We observe high deviation in response times which decrease as the number of query terms grow. Below we discuss the various implications of these observations.

According to [23] the intersection of posting lists dominates the processing cost which explains why on average we observe higher response times for larger queries. The big deviation observed in response times especially for small term queries (e.g. 1 to 3 terms) can be explained by considering the number of documents relevant to a query. Let's take for example some single term queries. There are single term queries like "http" or "wiki" which can virtually be relevant to all documents in the dataset. The processing cost of ranking all those documents plus the possible extra optimization searches that may be required for duplicate elimination yields high response times.

Next we discuss why the documents matched are decreasing with increasing query terms. As mentioned already, we use conjunctive multi-term queries thus as the number of terms grow it gets more difficult to find documents which contain all the terms of a query. For queries above or equal to 12 terms the average number of matches ranges from 4 to 16. It is true that the 11 terms point is an outlier which breaks the rather monotonic behaviour. We inspected the actual queries and noticed that indeed there are two queries which had a lot of matches. The processing time of those two queries are higher and increased the average response time for all 11 term queries. The wide deviation in response times for queries of the same length underlines the difficulty in developing accurate heuristics for predicting response time based only on number of terms.

The results in Figure 8 show the breakdown of another key web search variable: the contribution of queries with a given number of terms to the queries with long tail latency (>99%). The results in this figure have a rather good correlation with Figure 5 that shows the breakdown of queries in terms of number of terms. The large contribution of queries with few terms is explained by the fact that these queries also contribute the most queries in the entire stream as shown in Figure 5. Also, as we have already mentioned, queries with few or one term, like for example "wiki", may end up with many optimization searches. This kind of queries are very time consuming as they both need many searches and they have a lot of matching documents. The bottom line of this analysis is that we do not observe the tail being dominated only by long term queries. Small term queries also experience long latency. We want to mention here that the tail for a production web search could look different. For example, some single term queries which need a lot of process time in our benchmark in a production system they could be easily answered much faster by just bringing the first n docs with the highest authority score [12]. Recall that Nutch benchmark does not sort a term's documents by a priority score.

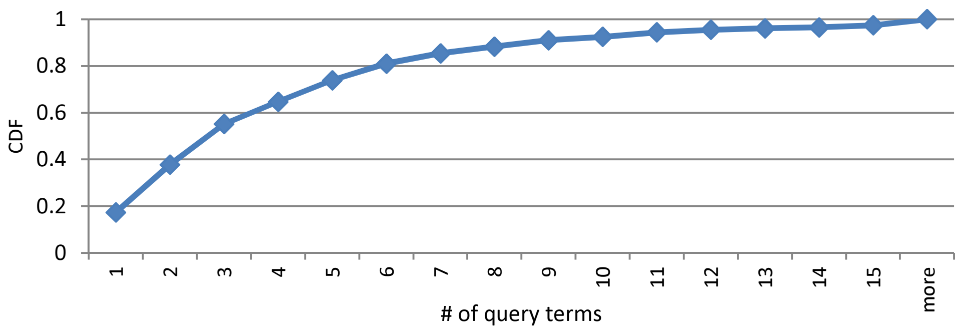


Fig. 8. The breakdown of 99th tail latencies as a function of the number of query terms. Small to medium size queries contribute more to the tail. This can be explained by the fact that those queries yield more matches so more processing time and duplicate elimination searches may be required.

We have also tried to use randomly queries generated using index terms. The Fig. 9 shows the corresponding behaviour of average response time and average number of documents matched, as query terms grow. There is no clear correlation between average response times and number of terms (contrary to what observed in Figure 7). A reasonable explanation for the difference is that it is unlikely to randomly generate meaningful large multi-term conjunctive queries. The intersection of posting lists of random multi-term queries may end very early in cases where most of the terms are completely unrelated. The results support this since the vast majority of the big queries have nearly no matches. It is clearly seen in Figure 9, that anything above 8 terms on average has essentially 0 matches. Those queries do not require optimization searches, detail and summary requests. This, consequently, lowers their processing cost. We decided, therefore, to use the AOL real query stream for the rest of the experiments because its behavior is more realistic and closer to what has been reported in previous work [23]. The AOL log file contains millions of queries and to limit, therefore, experimentation time we selected a region of 100K consecutive queries. We ensured that this region is representative by comparing its average and tail latency against other regions from different parts of the log file. This comparison has shown a standard deviation (STDEV) of 0.3ms for the average response time, 1.2ms for the 90th percentile and 6ms for the 99th percentile. These findings show that the various regions exhibit similar behaviour and provide confidence in selecting any one of them as representative of the AOL query log.

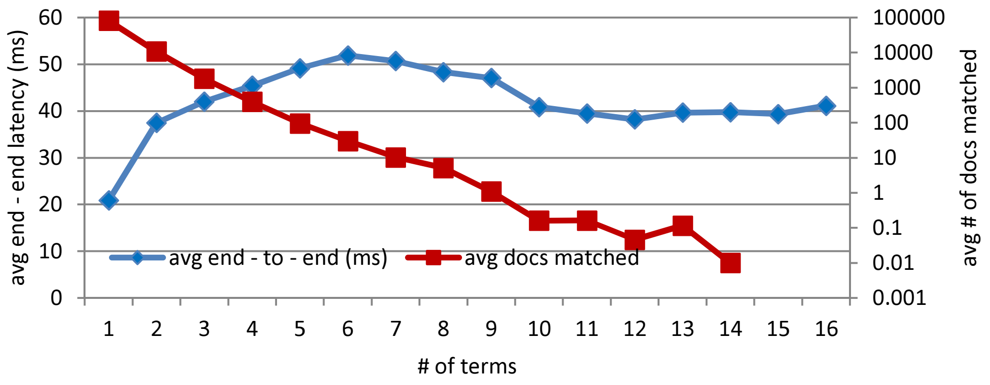


Fig. 9. Average number of response time and average number of matching documents for random query stream as a function of the number of query terms. The end to end response times do not have a consistent increase with number of terms. Also conjunctive large term random queries with 8 or more terms on average have nearly 0 matches.

5.2 Performance in relation to number of clients

In this Section we perform an investigation into how the number of client threads affects the system performance.

Figure 10 shows the throughput as well as the average, 90th and 99th percentile of response times with increasing number of clients. Our system has total eight cores available for handling parallel requests. Thus, the performance scales well from 1 to 8 clients. At that range the throughput scales nearly linearly and response times does not worsen much. A small degradation in response times is caused by increased contentions of shared resources as the number of active cores increases.

At 16 client threads the trends change. As the number of clients increases from 16 to 128 throughput remains constant or increases slightly and response times continuously increase (get worse) due to increase queuing delay (query simply waiting to be served). Figure 11 shows that the index server CPU utilization is strongly correlated with throughput. It is interesting that until 64 clients there is room to increase utilization. Beyond 64 clients the system is fully saturated and cannot get more throughput.

The results in this Section provide a validation of our experimental setup and also help select the configuration and traffic to use for the subsequent experiments. The results justify using a ratio of 1 client/core. Even though having a ratio of 2 clients/core or even more (16 – 64 client runs) can yield an increase in throughput, the increase in response times is much worse. Therefore, it is more beneficial to not run our system with more than one client per core. Running 1 client per core is a compromise that provides good enough utilization without sacrificing the response times.

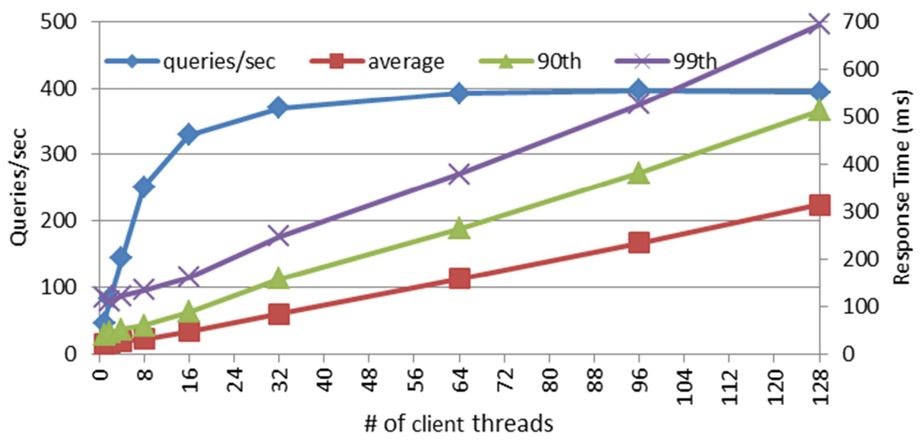


Fig. 10. Response times and throughput with increasing number of clients. Above 64 clients there is no improvement in throughput as there is contention between the many requests sent and long queuing delays.

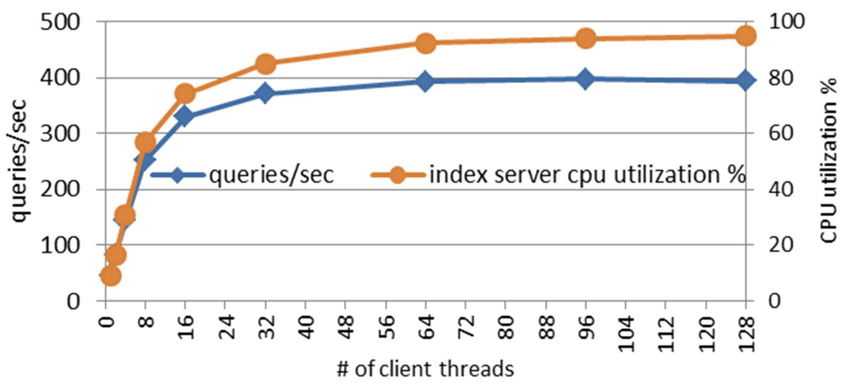


Fig. 11. Index server utilization strongly correlates with queries/sec. At 64 clients the system nearly reaches its maximum utilization. Beyond 64 clients the throughput remains constant.

5.3 Performance in relation to dataset size

Next we report on the findings about the sensitivity of the response time to the dataset size. The goal is to identify how the processing time is spent on the various phases of the benchmark as a function of the dataset size. We have already described the query flow in Section 2. We summarize and report the query flow in terms of the latency of the following four discrete web search phases: 1) the time spent on client – frontend communication (client sends request, frontend assembles the html response and sends it back to client), 2) the index search which is performed on the index server (including any optimization searches for duplicate deletion plus any time the frontend spends sorting the results, plus network communication between frontend and index servers), 3) the detail requests which are also performed on the index server, and 4) the summary requests which are performed on the document server. In all runs one client thread is used and for each run we incrementally add one more dataset part both for the index and the document servers.

From the results in Figure 12, it can be observed that the processing time for search increases linearly with the dataset size while the rest of the processing times remain the same. It is interesting to understand why summary generation time does not increase with dataset size. The explanation lies in the functionality of the document server, described in Section 2. The document server uses a hash function to determine in which partition a document is located, so it does not have to search all partitions to find a document. This behaviour helps to limit the time required for finding a document to be, more or less, the time needed for searching the average size of a segment partition, which does not change a lot across runs, plus the time required for dynamically generating a summary for a document. This targeted search approach cannot be implemented for index search as a relevant document may exist in any index dataset partition thus the search times are increased as we increment the index dataset. Adding more index dataset basically means adding more documents which is confirmed by the strong correlation seen in Figure 13. Figure 13 shows how the number of indexed documents increases with bigger index. This strong correlation means that with bigger index dataset the index server spends more time on traversing a terms posting list. This seems to explain the nearly linear behaviour relation between the index search and the index size. Figure 14 confirms this behaviour by showing that terms with bigger posting list require more index search processing time and again the relation between index search and document list in nearly linear. The more index dataset also means the addition of more terms but because the term search is performed in logarithmic time it doesn't affect much the query execution.

The results we have discussed also support what is reported in previous works [19,15,10], namely that index search is the most processing demanding part of a web search engine. It is clearly shown in the Figure 12 that the index search dominates the end to end response time and the index search becomes more dominant with increasing dataset size.

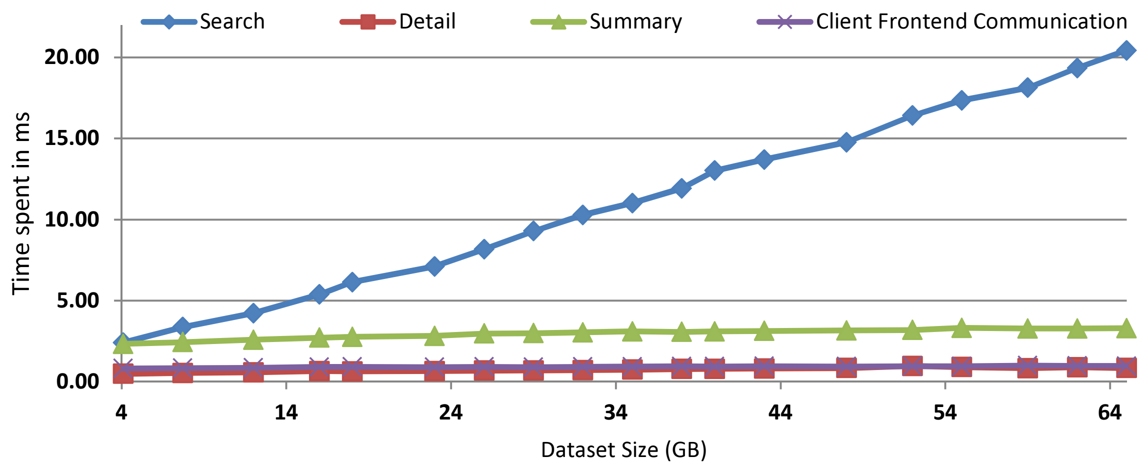


Fig. 12. Time spent at each phase of the benchmark in relation to dataset size. The index search is the most demanding phase and scales with dataset size.

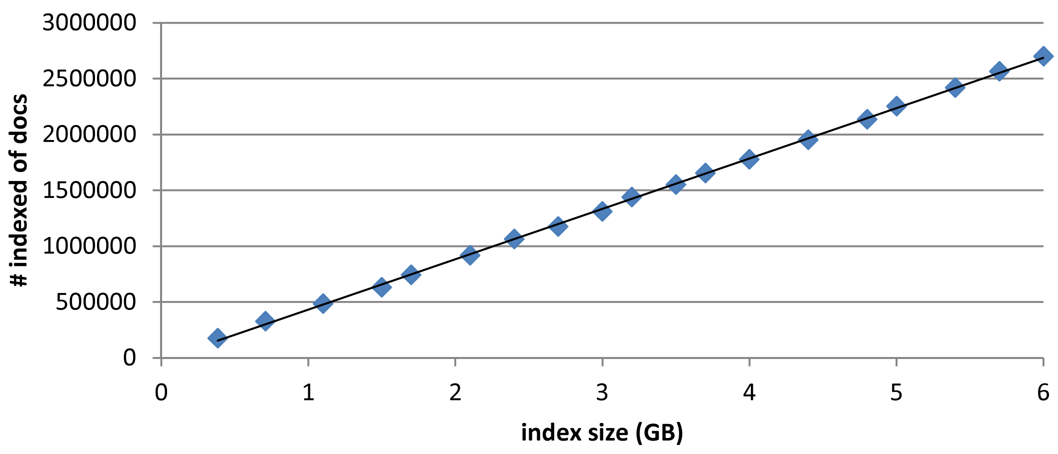


Fig. 13 Numbers of documents stored in an index vs the index size in GB. The linear relation between the number of documents and the index size in bytes clearly shows that the index size grows because of the increase of the indexed documents.

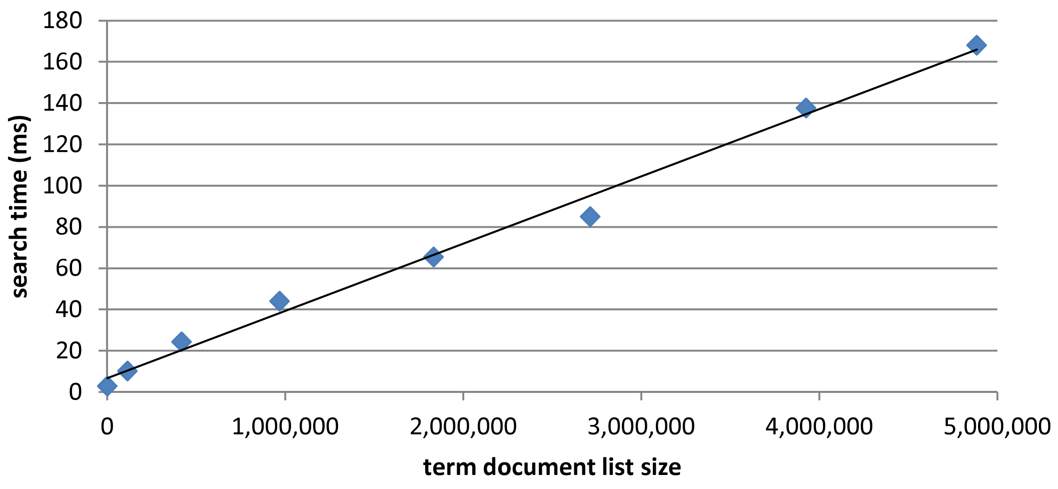


Fig.14 Single term query index search time vs term document list size. This graphs shows that the search is strongly correlated with a term's document list. The bigger the document list the more time is spent in index search.

5.4 Intra-Server partitioning

We first compare the response times of a setup using partitioning (two index servers working on different index parts of equal sizes) and a configuration without partitioning. The queries are executed sequentially one at a time. We observe that partitioning improves the index search times by 1.9x which yields an end to end response time improvement of 1.55x (index search time contributes 75-80% to the end to end response time) with a STDEV of 0.55 and 94.5% of the queries falling within the STDEV. The 1.9x speedup of the index search is quite high and is worth investigating if indeed the partitions have so good load balancing. As we already showed in the previous subsection the terms' posting lists indicate how much time is spent in index search. If the two partitions for most terms have equal posting lists this should be the reason for this good speedup.

Figure 15 show the histogram of how similar are the two partitions considering all terms. The similarity is calculated as ratio of the biggest posting list towards the smallest posting list. Values close to 1 mean absolutely equal partitions. By a first glance the results do not look good so great as the majority of terms have a ratio of above 2. But keep in mind that the average posting list size is just 34 postings. This implies that a lot of terms stored in the index are rare terms. The unbalanced terms usually belong to the rare terms and they contribute negligible time to a query's execution. Thus if we want to estimate how good is the partitioning load balancing we should look at terms with big posting lists. The histogram in the Figure 16 shows the ratios of the 1700 most popular terms. These terms have posting lists sizes of above 50,000 documents and they contribute a significant time in a query's execution as they add a processing time of approximately 5ms, which is near to the average index search time which is around 7-8ms. The figure clearly shows that there is more load balancing for these more popular terms. So we can conclude that our partitioning is quite good because there is good load balancing among the popular terms; which are the important ones as they add a lot of processing time to the query execution. Not popular terms do not have good load balancing but this fact is not so negative for a configuration which employs partitioning as those terms add insignificant amount of processing time. These findings indicate is worth further examining partitioning and two main questions which arise here are: (1) what is the best configuration to run an index server, and (2) are there any benefits from a partitioned index in a realistic setup?

A machine running an index server can have multiple index server processes each process working on a separate index part. Using many index server processes naturally reduces the response latency but at the expense of executing less independent queries in parallel. We study three different configurations of how to run the index server on a CPU. In particular, a socket in our Intel Xeon E5620 has 4 cores and we explore the following options: a) one index

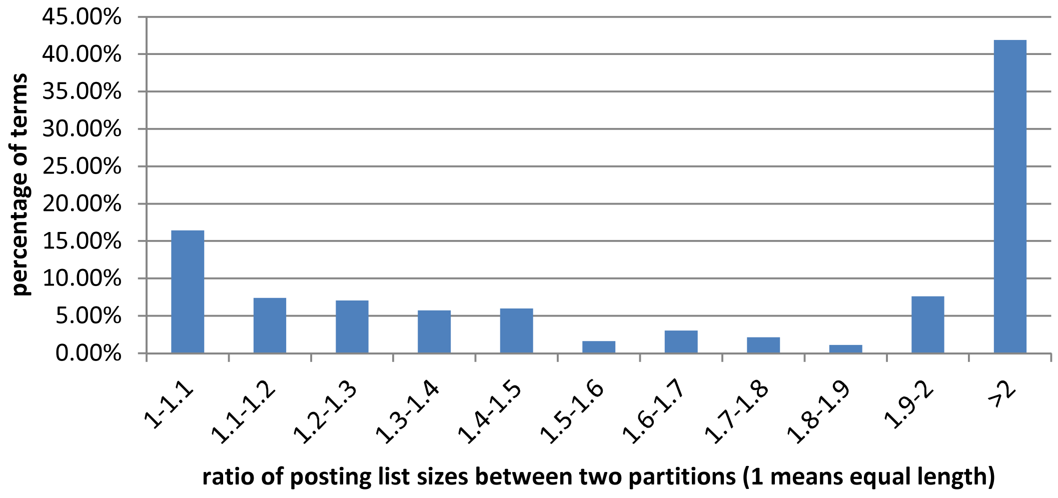


Figure 15 Histogram of the ratio of posting lists of all terms for two equal in size index partitions. Ratio was computed as biggerPosting/smallerPosting. Ratios of 1 mean absolutely equal posting lists. The majority of terms (40%) are imbalanced but the majority of this terms have small posting lists and they do not add much processing time to a query.

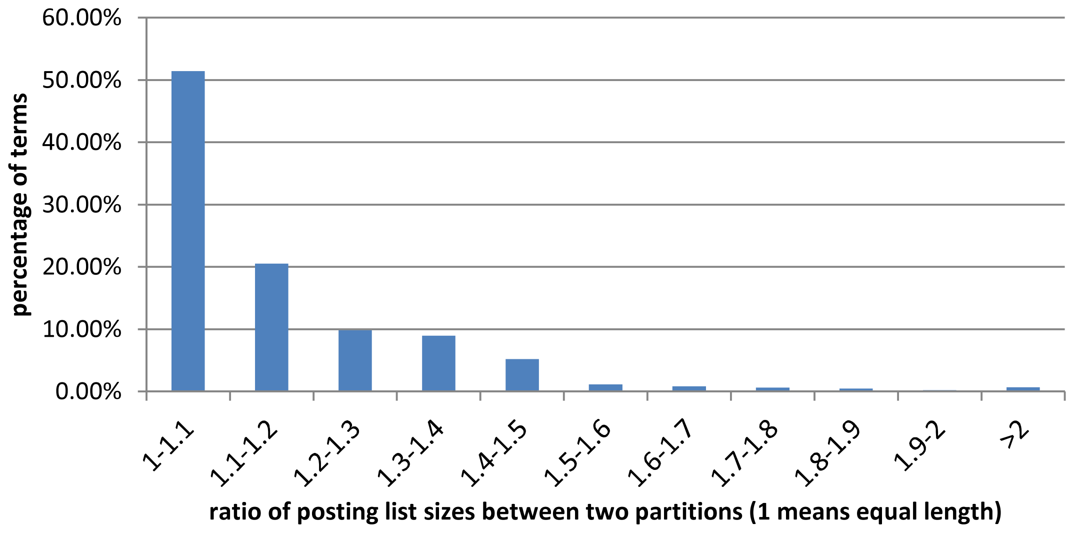


Figure 16 Histogram of the ratio of posting lists of the most popular terms for two equal in size index partitions. The terms considered in this histogram have document lists between 50,000 and 5,000,000 documents. Ratio was computed as biggerPosting/smallerPosting. Ratios of 1 mean absolutely equal posting lists. We can clearly see that there is much better load-balancing in this histogram compared to the Figure 15 which accounted all index terms.

server process and four cores per index server, b) two index server processes and two cores per index server process, and c) four index server processes and one core per index server process. We used the numactl command [16] to pin processes to specific cores and also to force the process to use only the CPU's own DRAM. To test the sensitivity of each option to incoming traffic we used from 1 to 8 clients in our experiments. For presentation clarity we will denote the different configurations with the triplet coding X_Y_Z, where X denotes the clients, Y the index servers and Z the cores per server, e.g. 4_2_2 means 4 clients, 2 index server processes, and 2 cores per index server process. The results of this analysis are reported in Table II and III.

Before discussing the findings we like to mention that an early analysis of the results with partitioning reveals that running the same query across the different configurations does not produce exactly identical results. The reason is that the global statistics like idf scores (inverse document frequency scores) are different with or without partitioning, because an index server process which holds only a part of the index does not have available all the information about all the indexed documents. This issue is discussed in [1,12] where it is suggested to use background updates of statistics or pre-calculation of correct idf values and distribute them to the index servers. These possibilities indicate that partitioning can be used without influencing the quality of responses in real setups. Nonetheless to ensure we get the same search results, across the different configurations in our setup, we experimented with all idf scores set equal to 1. This provides the same search results across the various configurations. We note that we do not observe any differences in response times between running with normal idf scores and idf scores set equal to 1.

Table II shows the runs from 1 to 3 clients where the system is underutilized. Recall, from the Section 5.2, that one client per core is needed to utilize the system sufficiently and at the same maintain low enough response times. In the situation with low incoming traffic partitioning improves both the average as well as the tail latencies. This is expected since, as pointed out in previous work [10], partitioning is beneficial during periods of low utilization. Table III shows the cases where a single socket is stressed to its throughput limits (4 to 8 clients). Interestingly, the results show even at high utilization partitioning improves 99th percentile with similar or slightly worse average and 90th percentile. We try to understand next why under stress conditions, less query parallelism, but more parallelism for index search due to partitioning, the average is not hurt while 99th tail improves.

Table II. Results of intra server partitioning exploration with low utilization. Partitioning (2 and 4 index server processes) improve both average and tail latencies over non-partitioning (1 index server process) at all traffic level (1-3 clients). In bold you can see the best per client metrics.

Clients	Index Server processes	cores per index server process	GB/Server	Ops/sec	Mean (ms)	90th (ms)	99th (ms)
1	1	4	5	45	22	42	107
1	2	2	2.5	70	14	26	64
1	4	1	1.25	**95**	**10**	**18**	**38**
2	1	4	5	85	23	44	110
2	2	2	2.5	95	21	40	84
2	4	1	1.25	**122**	**16**	**30**	**61**
3	1	4	5	113	26	50	116
3	2	2	2.5	**165**	**18**	**34**	**79**
3	4	1	1.25	148	20	40	80

Table III. Results of intra server partitioning exploration at high utilization. Partitioning improves 99th percentile. In bold you can see the best per client metrics.

Clients	Index Server processes	cores per index server process	GB/Server	Ops/sec	Mean (ms)	90th (ms)	99th (ms)
4	1	4	5	151	26	**49**	119
4	2	2	2.5	**163**	**24**	**49**	96
4	4	1	1.25	157	25	53	**92**
5	1	4	5	**186**	**27**	**50**	120
5	2	2	2.5	172	29	59	**106**
5	4	1	1.25	161	31	62	108
6	1	4	5	**196**	**30**	**57**	127
6	2	2	2.5	177	34	67	**117**
6	4	1	1.25	172	35	69	118
7	1	4	5	**201**	**34**	**64**	134
7	2	2	2.5	187	37	73	**122**
7	4	1	1.25	177	39	75	129
8	1	4	5	**209**	**38**	**70**	138
8	2	2	2.5	197	40	78	**133**
8	4	1	1.25	187	42	80	137

Qualitatively, assuming partitioning speeds up a query by the degree of partitioning used then the average is rather insensitive to partitioning. This is the case because the benefits from reducing an individual query's processing time is nullified by an increase in its queuing time, waiting for the other queries to complete (that with no partitioning are processed in parallel). In a realistic system many parameters influence the benefits from partitioning, such imbalanced partitions, scheduling order and overheads, resource contention. Therefore, to better understand the behaviour of response latencies with partitioning at high utilization, we show in Figure 17 the fraction of queries that are improved with partitioning (configuration 4_2_2) as compared to no partitioning (4_1_4) as a function of the response time of queries with no partitioning (x-axis). The results clearly show that the longer the response time of the query with no partitioning the more likely it is for partitioning to help. In particular, 90% or more of the queries are improved when their response time with no partitioning is 80ms or more whereas for short queries only around 50% are improved. The cumulative weighted percentage of improved queries shows that 4_2_2 improves 60% of the total query stream with the long queries having a rather small contribution. The fact that long queries are nearly always executed faster with partitioning is backed up from our previous findings. Recall that terms with large posting lists ("popular terms") add high processing time to the queries, so queries made of these terms is very likely to be the ones which consist the 99th percentile (the 1000 slowest queries). Recall also that the posting lists of the most popular terms are well balanced among the partitions. This two facts combined means that the execution time of long queries in a partitioning configuration should be significantly improved which is what we observe in Figure 17. So to conclude, the fact that all long queries are nearly always improved with partitioning, is a reasonable explanation why we see lower 99th percentile for the partitioning configuration even for scenarios with high queuing effects.

The consistent tail improvement with partitioning (Table III results from 4 to 8 clients) has another simple explanation. Consider two differently configured dual-core index servers. The one uses 2-way partitioning and the other does not use partitioning. Let's assume two queries that are executed independently and in parallel in the non-partitioning setup with latencies X and Y. When the same queries are executed in the setup with partitioning, each query's processing time is theoretically reduced by two but the two queries are processed sequentially.

Therefore, each query's processing time will be the others queuing time. Therefore, the slowest query will have a response time equal to (X+Y)/2 which will always be less or equal to the latency of the slowest query for the non-partitioned configuration. When the two queries have equal latency no improvement is possible. Consequently, partitioning can help reduce tail latencies as long as co- executed queries have sufficiently large difference in their (non-partitioned) response times. The response time improvement from partitioning compensate for the increase partitioning causes in queuing time. In cases when co-executed queries have similar response times or their search times are very small the partitioning will make things worse as the queuing overhead will dominate over the reduction in search time.

We provide two examples to better illustrate how partitioning can influence response latency. The example in Figure 18 shows a case where it improves a slow query. We assume two clients sending queries at the same time. Client 1 sends only queries of 15ms (B) and client 2 queries of 10ms (A). The two systems under comparison are 2_1_2 and 2_2_1. The 2_1_2 executes each query in a separate core parallel. The 2_2_1 executes queries sequentially interleaved because it has only one core available but each query is assumed to be served in half the time. As shown in the Figure, the 2_2_1 makes the average slightly worst but the maximum latency gets less. The average is getting slightly worse because the fast queries have the same latency as the slow queries.

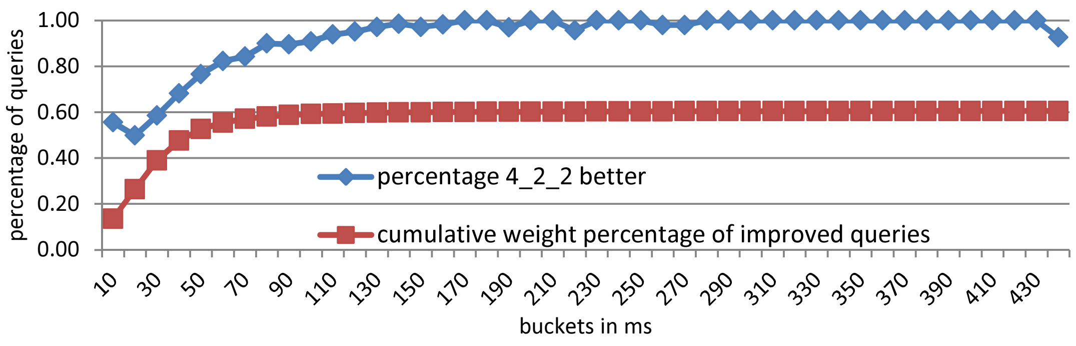


Fig.17. The x axis separates the queries based on their response time for 4_1_4 setup. The graph shows what percentage of each query type 4_2_2 improves over 4_1_4 and the cumulative weight percentage of the improved queries. 4_2_2 improves nearly all long latency queries but those queries are a small portion the query stream. Overall 4_2_2 improves 60% of the total query stream.

Figure 19 presents an example where queries have the same latency. In this case 2_2_1 does not improve the maximum latency because whatever is won from partitioning is lost from the queuing. These examples are simple but they provide a clear explanation of how partitioning helps tail latencies and does not hurt average. They also clarify the trade-off offered by intra-server partitioning: increase the latency of fast queries and at the same time decrease the latency of slow queries.

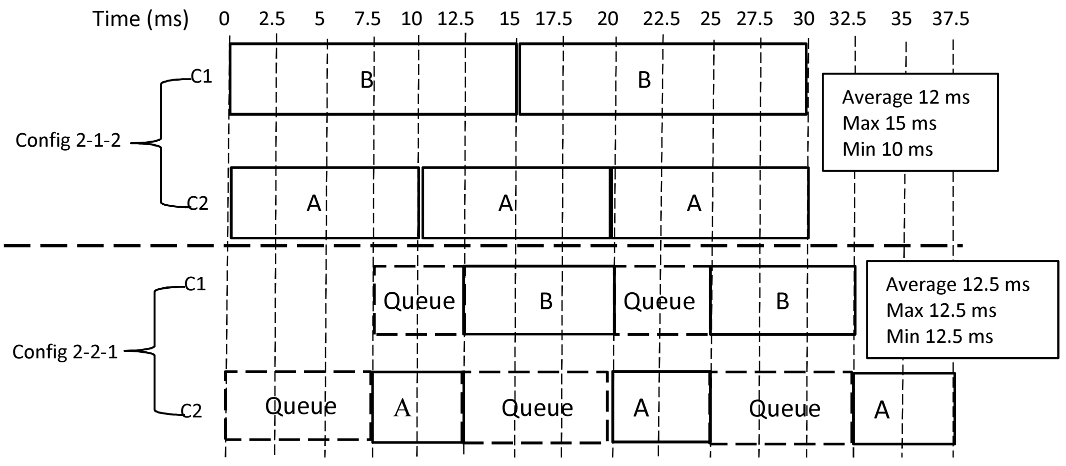


Fig. 18. An example where partitioning reduces slow query's (B) latency and increases fast query's (A) latency. We assume a steady state of execution that why the first A query has added latency from query B.

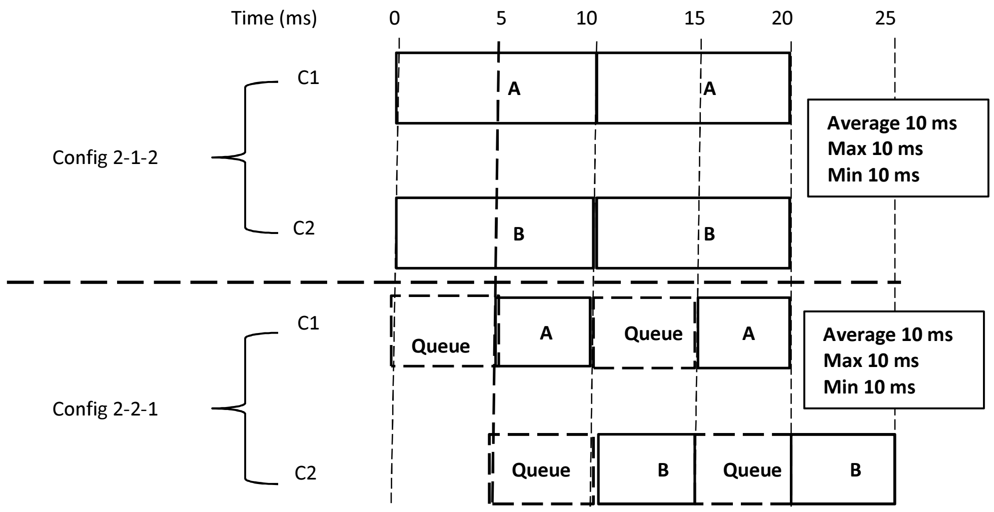


Fig. 19. A case where partitioning does not help tail. Both queries have the same latency. Whatever is won from partitioning it is lost in queuing.

The question that remains is whether co-executed queries have enough variability in search times in order to give a partitioned index configuration enough opportunity to reduce tail latencies. We believe that a possible indication of this variability is the difference in single client response times from consecutive queries. Figure 20 shows a cumulative distribution of difference in search times of consecutive queries used in this study. Note that 60% of the time we have difference of less than 10ms. We also observe that 20% of the consecutive queries have difference from 10 to 20ms and another 20% have differences that are more than 20. Index partitioning leverages these differences to improve the tail latencies shown in Table III.

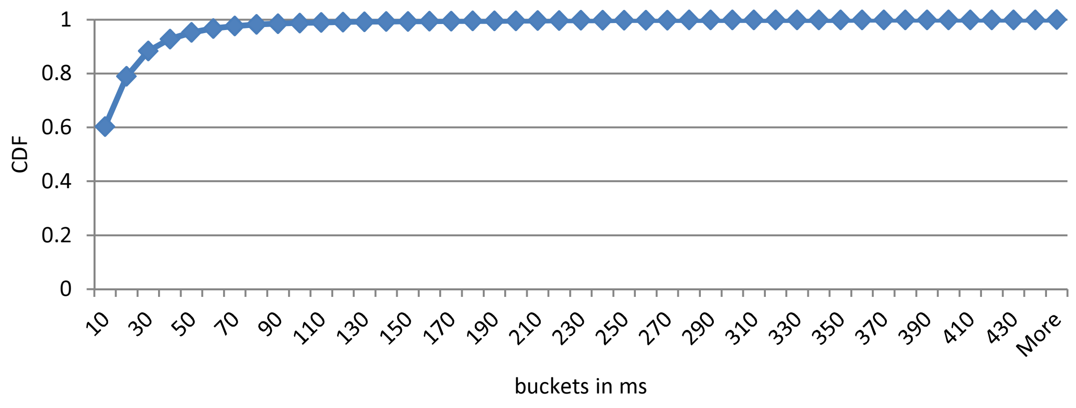


Fig. 20. A cumulative distribution of the difference in response times between consecutives queries. 20 % of the pairs have a difference of more than 20 ms.

5.5 Index Server performance in relation to CPU frequency

We have already shown that index search time grows linearly with index dataset size. This behaviour provides good motivation for exploring the use of low power servers for web search with smaller index dataset. Already previous work has investigated the use of low power servers for web search [19, 18]. Specifically, [19] reports that low power cores cannot meet the response times of conventional servers like Intel Xeons and they propose some micro-architectural enhancements to help low power cores performance. A report of throughput comparison, without comparing response times, between low power and high performance servers is given in [18]. In order to approximate the performance of a low power server we reduce the frequency of our Intel Xeon CPU from 2.4Ghz to 1.6Ghz. We used 1.6Ghz clock frequency for the low-power because it is the lowest frequency setting we could go without changing the BIOS settings.

Figure 21 shows that for 1.5x less dataset the Xeon@1.6Ghz achieves the same average response time as the Xeon@2.4Ghz. This may indicate that for the Nutch benchmark – with sufficient degree of partitioning – low power servers can match the performance of high performance servers. Table IV shows how two sockets of 1.6Ghz Xeons compare to one socket of a 2.4Ghz Xeon. We select as baseline configuration the 4_2_2 which achieves better average, 90th and the second best 99th (see Table III). We use 4 clients to explore all possible 1.6Ghz index server configurations. The best configuration for 1.6Ghz is the 4_8_1 as it is better in all statistics. Again we observe that partitioning helps to reduce tail latency.

From this analysis we conclude that many low power servers can help improve throughput, as stated in previous work, but also response time when enough partitioning is applied. One could argue that our analysis gives some advantage to the 1.6GHz setup as we give each slow server 2 times less dataset instead of 1.5 times less (the 1.6GHz is only 1.5X slower than 2.4 GHz). But this may be desirable for another reason. When using a larger number of low power servers, as compared to high power servers, it becomes more likely that a server will slow down, due to poor load balancing, the whole query execution [2,9]. Giving each low power server less amount of work reduces the chances a single server to slow down the whole query execution. However, performance is not the only metric that concerns the design of a data center infrastructure, a TCO comparison must be done to justify the benefits of a DC built with more low power servers as compare to fewer high power servers.

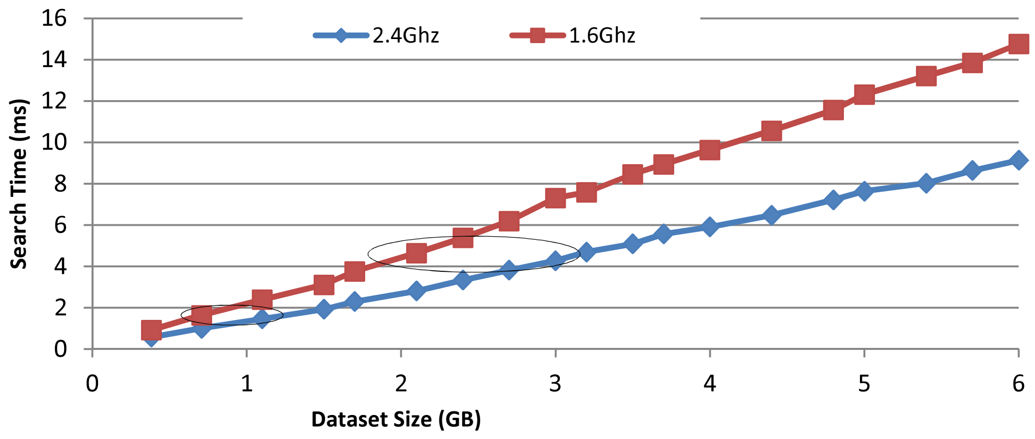


Figure 21. Index search time for an index server running at 2.4Ghz and at 1.6Ghz vs Index dataset size. The 1.6Ghz achieves the same response times for 1.5x less dataset. For example look at the points in the oval shapes.

In particular, we estimate how much should the cost be for the 1.6GHz server (Motherboard (MtB) plus chips acquisition cost) to be profitable in terms of TCO in comparison to the baseline. Our baseline is a datacenter of single sockets servers running E5620@2.4GHz. The low power servers must offer high density and high integration to be profitable. We study three integration scenarios: a) a pessimistic – single socket MtB, b) the

Table IV. Intra server partitioning for 1.6GHz Xeon and how it performs to the baseline best configuration of 4_2_2 (highlighted). The 4_8_1 is the best 1.6Ghz configuration.

Index servers	Cores Per server	GB Per index server	Total Cores used	Core Frequency (Ghz)	mean (ms)	90th (ms)	99th (ms)
1	8	5	8	1.6	39	75	187
2	4	2.5	8	1.6	25	48	109
4	2	1.25	8	1.6	23	49	103
8	1	0.625	8	1.6	23	45	90
2	2	2.5	4	2.4	24	49	96

scenario we actually examined dual socket MtB, and c) one optimistic scenario with four socket MtB. In all cases we keep stable the number of total processors for the 1.6GHz server so that a 1.6GHz datacenter has always twice the processors of the baseline 2.4Ghz configuration. The amount of disks, drams, as well as, the average datacenter utilization remains the same. We assumed that with lower frequency and voltage the 1.6GHz power consumption falls to 34W peak and 13 W idle as compared to 80W peak and 30W idle at 2.4GHz. We estimated the baseline datacenter's server cost to be $616 assuming that a single socket MtB costs $225 plus the price of an E5620 processor which based on Intel's website costs $391.

We estimated TCO using the tool proposed in [7]. The TCO results, shown in Figure 22, reveal that with more integration the low power servers can be more profitable. With single socket MtB for server cost up to $200 the 1.6GHz option is more profitable. For dual-socket cost of up to $600 and for four-socket cost up to $1500, it may be profitable to use 1.6Ghz instead of 2.4GHz processors. It is interesting that if we just care about power efficiency (as power budget is one of the most significant constrains in large scale computing) the two under clocked Xeons are a better choice over the one Xeon at the nominal frequency. The two Xeon at 1.6Ghz give a total peak CPU power consumption of 68W which is 17% lower than the 80W of the 2.4Ghz Xeon. So the 1.6Ghz configuration can provide for many cases better performance and better power consumption.

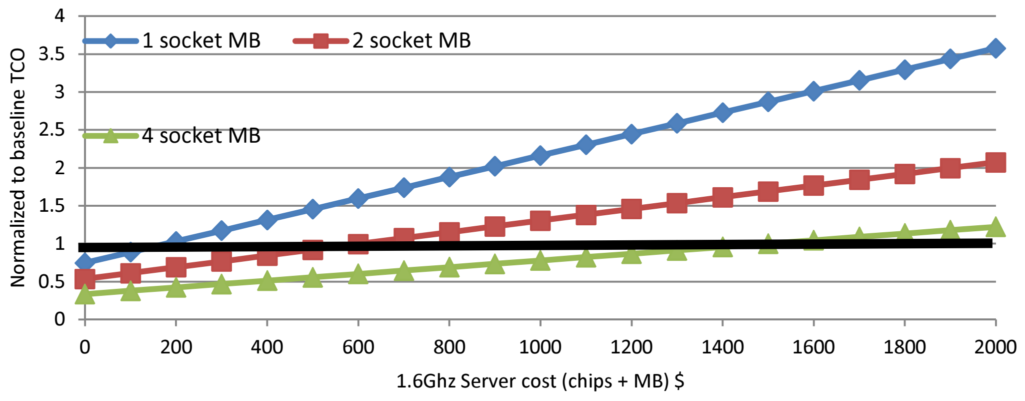


Fig. 22. Relative to baseline (2.4GHz datacenter) TCO of 1.6GHz server datacenter. We see that the more sockets per motherboard the more money the 1.6GHz can cost without surpassing the TCO of the 2.4GHz. The integration is the key for the success of low power servers.

5.6 Performance variability characterization

In this section we evaluate how DVFS, job migration, NUMA architecture and background processes affect the performance of the index search. We focus on index as it is the most demanding part of the benchmark and variability is more prevalent to index search as it is an operation performed in parallel across many servers. Figure 23 shows the performance benefits by mitigating each source of performance variability. The results were obtained with single client single core run. The following experiments were conducted: a) the naive run where we did not try to address any performance variability source, b) the DVFS_OFF where we disable DVFS and constantly run at the max frequency, c) corePinning where we mitigate the thread migration by pinning the index server to a specific core, d) is DVFS_OFF combined with corePinning and in e) we combine DVFS_OFF with memPinning for avoiding the access to the remote memory node (addressing NUMA related variability). We observe that there is a 20-25% gap between the naive run and the rest ones both for average and 99th percentile. We observe that it is enough to disable DVFS or force core pinning to get significant boost in performance. The combination of them or adding memory pinning for removing non uniform memory access does not yield extra performance benefits. This behaviour implies that the culprit for reduced performance is actually the DVFS. It seems that when forcing core pinning, you actually force the core to run constantly at the max frequency and that is why it yields the same effect as the "DVFS_OFF" experiment. The other possible implication of these experiments is that the default Linux DVFS governor does not set the frequency to the highest performance state fast enough. The fact that taskset and numactl do not yield any more performance benefits than just turning off DVFS may imply that this benchmark is not memory intensive (we will investigate this in the chapter 6).

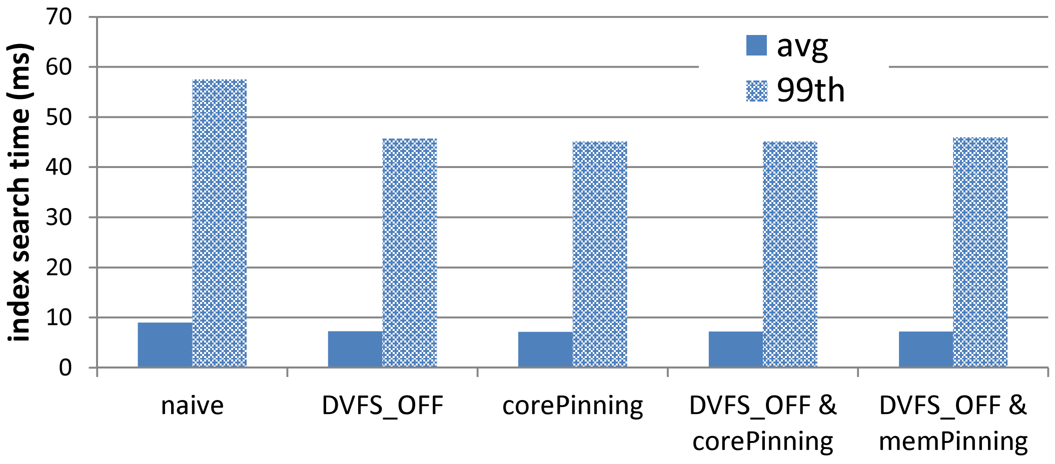


Figure 23 Performance benefits of removing various sources of performance variability by using core and memory pinning and disabling DVFS. By disabling DVFS or applying core pinning we get the same performance benefit. Combination of those two or memory pinning do not yield any more benefits.

While with the aforementioned tunings we got a significant improvement in average response time and 99th percentile, those two metrics do not present the complete story. Between identical runs of the same query stream and the same system configuration we have observed that approximately 200-300 queries of the total 100000 executed show significant slowdown or speedup. This is shown in Figure 24 where we have sorted the differences of the query execution times between the two identical single client – single core runs. Approximately 100 queries were executed faster in runB (negative values) and approximately 100 queries were executed faster in runA (positive values). To summarize approximately 200 queries out of 100000 seem to encounter variability issues of more than 50 ms.

The possible reasons of this variability are: 1) Background jobs and interrupt processing, 2) Waiting for I/O and in our case Hard disk drive (HDD) I/O 3), soft page faults caused by memory allocations and de-allocations [38], and 4) micro-architectural issues like better or worst caching due to address mapping.

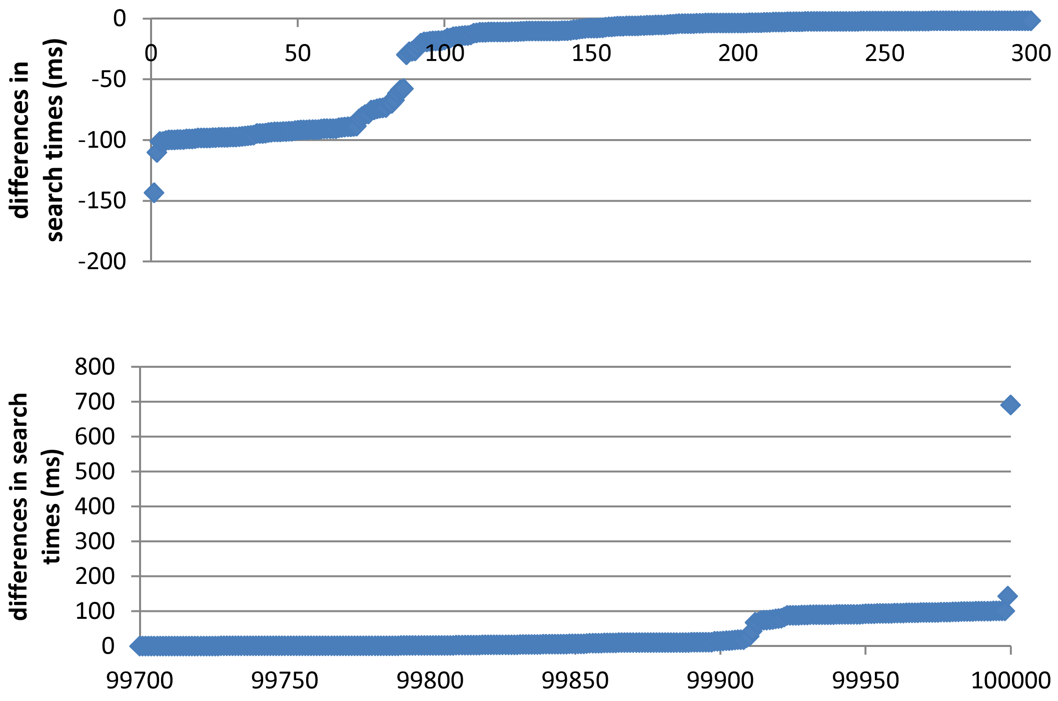


Figure 24 The differences on search times of 100,000 queries of two identical runs runB-runA sorted from smallest to largest. We show only the beginning and the tail of the graph to emphasize the variability. Immediately we notice that for majority of queries the differences are small but there are around 100 queries which are executed much faster on the second run and around 100 which are executed faster on RunA.

For investigating the per query variability we had to obtain per query performance statistics. So we used the per query experimental methodology described in 4.2.3. We estimate the effect of the background jobs and interrupt processing by looking at the idle time of the index server process. The idle time is estimated using the following formula indexSearchTime – (perfUtilityCountedCycles / cpuFrequency). Keep in mind that we have set up the perf utility to count only cycles related to the index server handler which is the thread that process the query. So the rationale behind the idle time formula is that the differences between the time counted by perf (which is equal to cycles/cpuFrequency) and the actual index search processing time is the time spent on background jobs or interrupts which do not contribute to the query execution. For determining if we have disk activity I/O we used the ps Unix utility [40] and we checked the maj_flt statistic of our index server process. This statistic shows the number of hard page faults that are related to the process of interest. That statistic was always zero in all our runs which makes sense as our working set fits into the main memory, nonetheless we confirmed that disk I/O was not the reason for the variability. Soft page faults statistics could be obtained from perf or from the min_flt statistic of ps and the micro architectural statistics like cache misses were collected from perf.

We tried to be as fine grain as possible and examined the causes of variability equal or bigger than 2ms. Our results in Figure 25 show that the main culprit is the idle time. We hardly could find cases were the IPC was the culprit (only 5% of all queries examined). In those cases the caching behaviour seemed like the most possible culprit. Idle time seem to be also caused in some cases by soft page faults as in some case where we had a lot of page faults we also had a lot of idle time. Soft page faults are most likely caused by the memory dealocations performed by the java garbage collector and is implications to performance require more comprehensive examination.

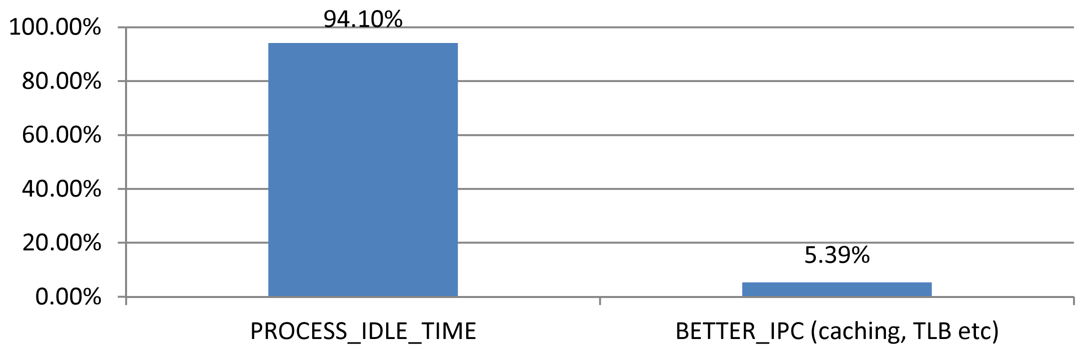


Figure 25 Breakdown of reasons that causing performance variability. The idle time is the culprit and not variability in IPC. The idle time is caused by background jobs.

To confirm that the idle time and consequently the background processes were the cause for the performance variability we tried setting the index server scheduling priority to real time. By setting the priority of the index server to real time, the index server can pre-empt other processes with normal priority. We observed that by setting the priority to real-time we nearly eliminated the variability (the chrt utility can be used for setting real time priority [40]).

Figure 26 show that we nearly eliminated the variability with real time scheduling. In Figure 27 we can see that the real time priority works well both for single and multiple client runs. Multi-client runs have slightly more variability but in general as we also showed in sub section 5.2 as long as we use 1 client per core there is little performance impact when increasing the number of active cores.

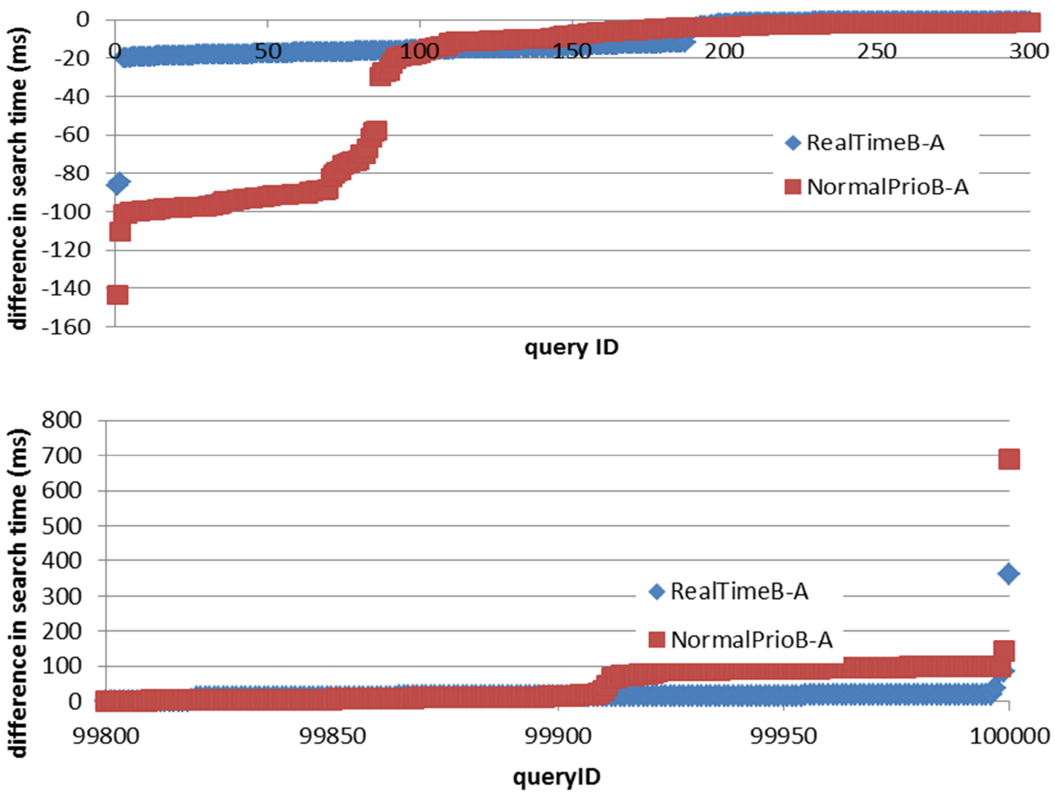
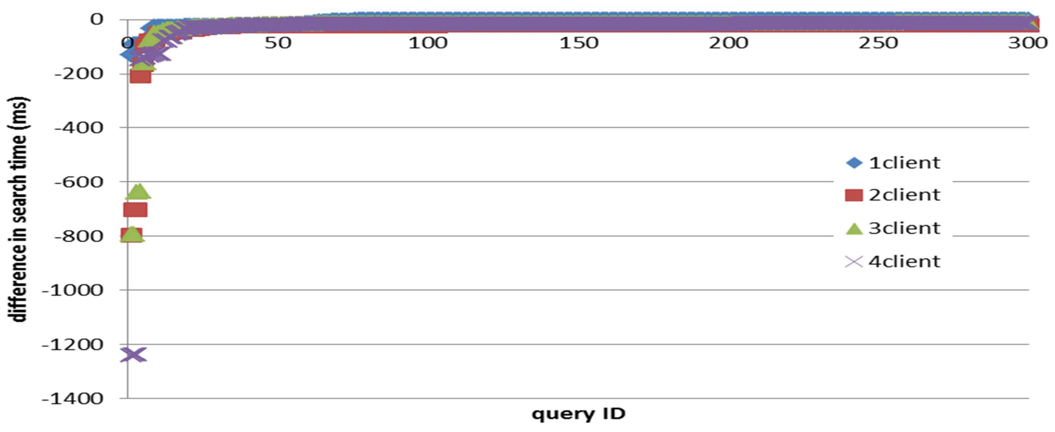


Figure 26 How real time scheduling improves performance variability. In order to emphasize the variability we skip the points were the variability were zero. So the top graph shows the 300 queries which ran faster in runB and the bottom graph shows the the queries which ran faster in runA. Clearly with real time scheduling the variability is diminished.

A further examination of the performance variability revealed that the most variability is caused by the monitoring process we were using in our framework e.g. top. Nonetheless in real setups is very common to use auxiliary processes for monitoring as well as co-locating batch jobs with QoS critical application [13, 14, 41]. Thus will believe that the performance variability study we conducted has the following contributions. We have characterized the performance variability which is induced by typical monitoring processes. The aggregate statistics e.g. average may not show a significant performance overhead but if we look the problem at a per query granularity we encounter significant variability. This is an important issue for a service like web search which runs parallel search on top of thousand machines. We have showed that the variability can be mitigated by setting QoS critical application's scheduling priority to real-time priority. This way monitoring or other useful background processes can co-run with the QoS critical application without causing performance variability.



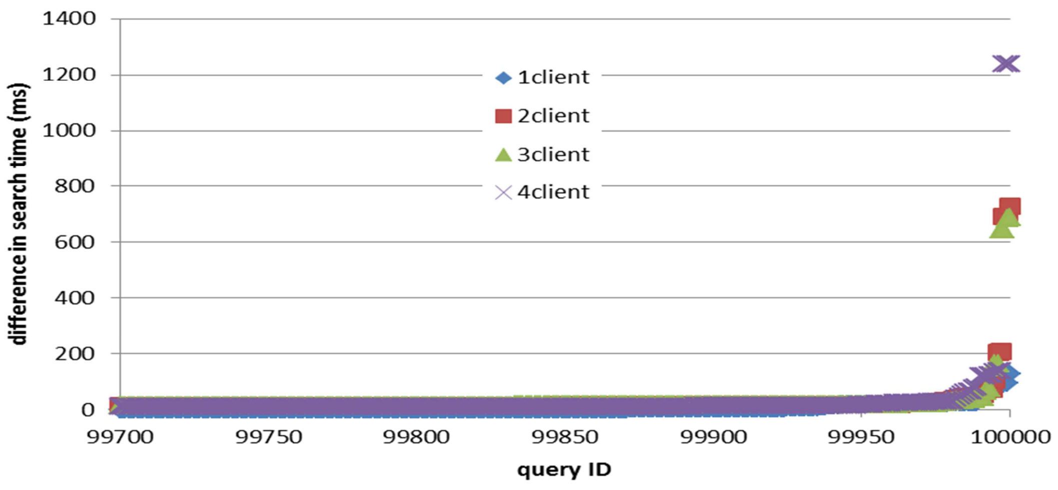


Figure 27 How real time scheduling performs on multi client runs. We can see that the real time priority works well both for single and multiple client runs.

Chapter 6

Micro architectural characterization and optimizations

We performed micro architectural characterization to better understand the high level behaviour of this benchmark. Section 6.1 explains the micro architectural behaviour of the document and index servers. Section 6.2 explores the impact of prefetching in performance. Section 6.3 explores the cache size sensitivity of the index search. Section 6.4 explores how query temporal locality affects performance. And section 6.5 explains the correlation between the results of this chapter and the high level characterization shown on the previous chapter.

6.1 Characterization and analysis

Table V shows the average per query IPC, instructions executed and the last level cache misses per 1000 instructions (L3MPKI) for the 2 main benchmark processes index and document server for a single client run.

Table V The micro architectural statistics of the 2 main benchmark processes per query.

Process	IPC	INSTRUCTIONS_COMMITED	L3MPKI
DocumentServer	1.49	35,837,932	1.139886
IndexServer	1.44	67,340,923	0.413163

The client and frontend server processes do not justify much discussion because they mainly do network I/O thus we concentrate on the Document and the Index server processes which are the processes which do the most work. The instructions committed confirm that the most time of the benchmark is spent on index search as the index server has nearly double instructions committed than document server. Both processes have fairly high IPC (nearly 1.5 instructions are committed per cycle) and low L3MPKI.

Let's analyse first the behaviour of the document server. The L3MPKI of the document server is a bit higher than the index server's which makes sense as document server uses 10x more dataset than the index server. Nonetheless the fact that the document server performs a key-value operation for retrieving documents (as described in Chapter 3) implies that the document server does not read much data from the main memory which justifies why the L3MPKI is low. Also the document server performs dynamic summary generation which is a compute intensive procedure which justifies the fairly high IPC.

Now let's interpret the index server micro-architectural behaviour. Index server has low L3MPKI despite the fact the index size is equal to several GBs. One explanation to this behaviour is that the index server actually does not read so much data from the memory. We estimate (by taking in account the posting lists of each query's terms) that for the query stream we used in our experiments, the average per query amount of postings that the index server must read is approximately 50,000 postings. Assuming 8byte postings (4byteDocid, 4byteTermFreq) the size of 50,000 postings is equal to 400KB which is not much. Also recall from the Chapter 3 that the postings are compressed which further reduces the postings size. As shown in Table VI we estimate that compression can reduce the postings footprint by 60%. This means that the average postings footprint read per query is further reduced to approximately 160KB. Of course there are queries which touch millions of postings like the query in Table VI. But the compression does a great job and the L3 misses are significantly reduced. Table VI shows that without compression we would expect to read an amount of data 3 times bigger than our Xeon's LLC but due to compression the actual amount of data read is slightly more than 12MB.

Table VI Characterization of the compression efficiency. The compression applied to the index significantly reduces the L3misses. The amount of data read from dram is calculated with this formula: L3misses * 64BYTES_BLOCKSIZE

query	Doclist size	Actual L3 misses	Expected L3Misses W/O compression	Data Read From DRAM (MB)	Expected Data Read From Dram W/O compression (MB)	Compression Savings
org	4,884,604	241,771	610,576	15	39	*60.40%*

The other reason for the high IPC and the low L3MPKI of the index search operation is the compute intensive nature of the postings decoding algorithm. The index server spends a significant percentage of time in decoding the postings. We know this because we have shown in section 5.3 that the index search time is dependent on the size of the posting list. The algorithm for decoding the postings is presented in Figure 28.

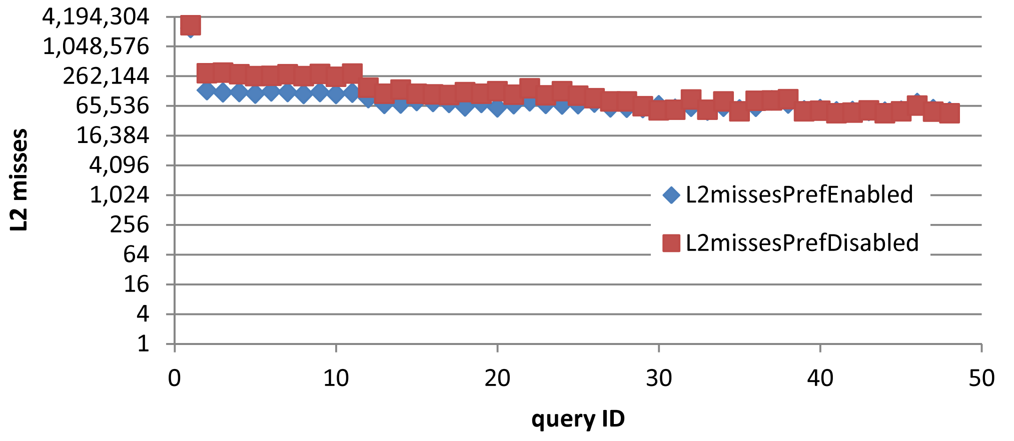
```
byte = data[i++];
 int value = byte & 01111111; //get the first byte value
for ( shift = 7; (byte & 10000000) != 0; shift += 7) //while first bit not equal to zero
{
        byte = data[i++];  //read next byte
         value |= (byte & 01111111) << shift; //get the value of next byte, add it to value
 }
```

Figure 28 The decoding algorithm. We extracted this algorithm from the Lucene source code.

In order to decode either the document id or the term frequency we have to read byte by byte the byte stream and perform the steps shown in Figure 28. We can count at least 8 instructions per byte read. If you multiply that by 64 bytes which is the size of a cache block in total 512 instructions are executed per cache block. Assuming a cold cache this would translate approximately to a L3MPKI of 2. So the low L3MPKI is actually a characteristic of the decoding algorithm. The total index server L3MPKI as we can see in Table V is even lower. This is expected as prefetching or postings cached from previous queries would reduce the L3misses. Also the 8 instructions per byte we calculated is just an approximation of how many instructions are executed for the decoding algorithm. The real code executes even more instructions per byte like increasing the count of postings read and checking if the document frequency number is reached.

6.2 Prefetching Evaluation

One would think that prefetching would help the index search performance due to the streaming nature of reading the postings but this is not the case. The time spent on reading the posting lists from memory is negligible in comparison to the time spend on processing. Thus even though prefetching will decrease the misses, the impact in search processing time is small. Figures 29 and 30 show this behaviour. We have captured the per query L2 and L3 misses of single term queries of various processing times with L2 prefetching enabled and disabled. We want to say here that our CPU has L2 HW prefetcher which can prefetch data both from DRAM and LLC. In the Figures 29, 30 the x axis is sorted from the slowest to the



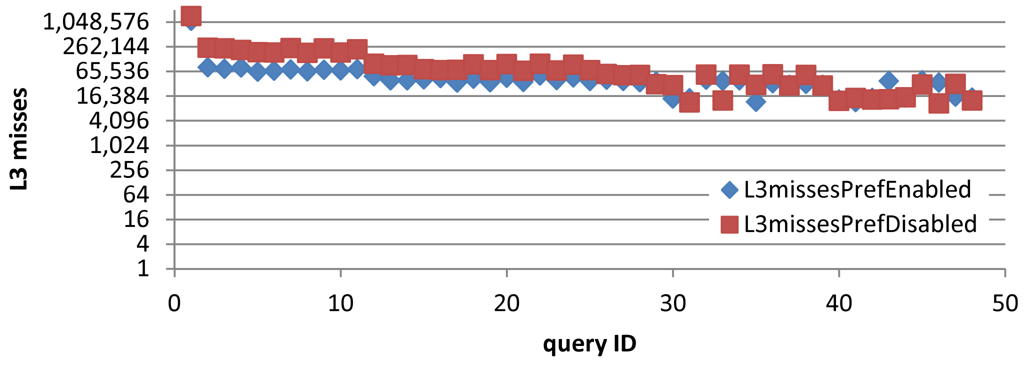


Figure 29 The L2 and L3 cache misses with L2 prefetching enabled and disabled. The queries are sorted from the slowest to fastest. At least for the first 10 queries which have millions of matching docs the prefetching significantly reduces the L2 and L3 misses.

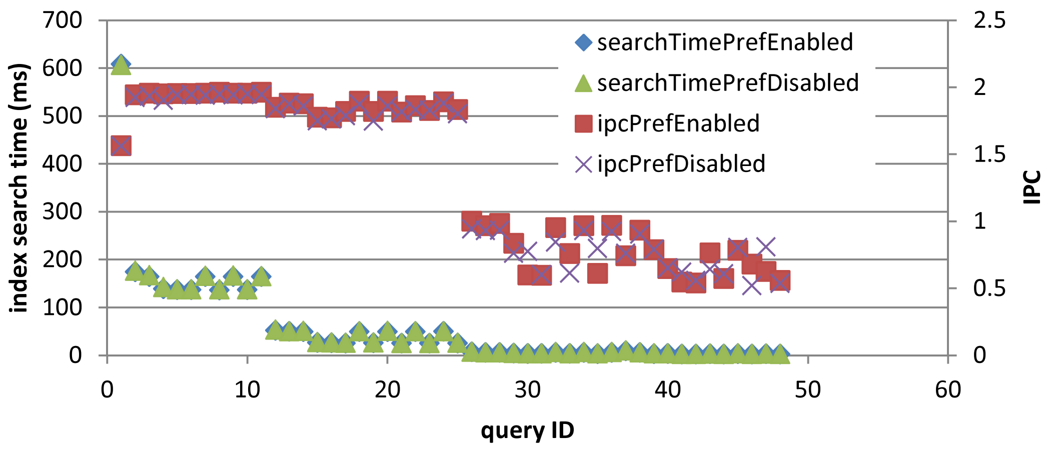


Figure 30 IPC and index search times for L2 prefetching enabled and disabled. The queries are the same ones used in Figure 29. While the Figure 29 shows that L2 and L3 misses for some queries are significantly reduced the reduction does have any impact in IPC and index search time.

fastest queries. The slow queries have millions of matching documents. We can see in Figure 29 that at least for the very slow queries (the first 10) the L2 misses are reduced approximately two times and the L3 misses are reduced 4 times with prefetching enabled. But the IPC and the index search times are identical as shown in Figure 30. This implies that prefetching optimizes only a negligible portion of the query execution time. This results suggest that it is worth examining whether energy savings can be achieved by disabling prefetching.

6.3 L3 Cache Size exploration

The low L3MPKI of the index search suggests that index server could live with smaller L3 cache which potentially translates to either cost (cheaper CPU) or power savings (less leakage). Also it can translate to better datacentre utilization as we may be able to co-allocate web search with other processes without hurting the web search QoS. We used bubbles [13] to emulate reduced cache size and explore the index search degradation. Bubbles are processes which perform random accesses on a user specified size array. This way stress on shared memory resources like LLC, DRAM and etc. is achieved. In our experiments we co-allocated one index server process with multiple bubble processes; each bubble process performed random memory accesses on a 3MB array. We performed 3 experiments: a) with 1 bubble, b) with 2 bubble2 and c) with 3 bubbles. We allowed the index server to use only one core and we ran the experiments using one client. Effectively this way we measured the single client index search performance with 9,6 and 3MB of L3. We tried to explore intermediate values like 2MB but we observed that the system was giving at least 3MB per core even if the bubble was using less than 3MB. Figure 31 shows the results of our evaluation. We can see that for 9MB the index search shows negligible degradation. For 6MB we have a 6% degradation and for 3MB we have a significant degradation of 9%. Given those results we would suggest than an ideal cache would be somewhere between 6 to 9MB. For sure a core running index search doesn't need all the 12MB of cache and can co-allocate with at least one other process with small memory footprint without any degradation.

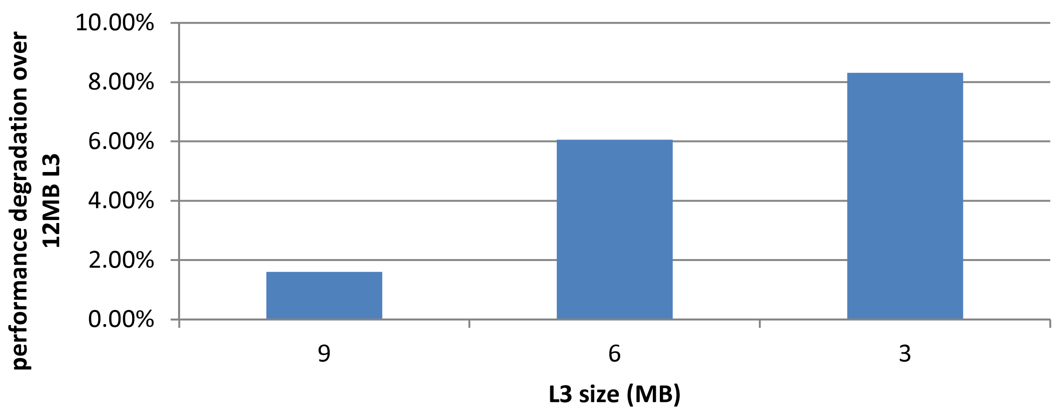


Figure 31 Cache size exploration for index search. The figure suggests that the index server can work with 9MB without any significant performance degradation. Even with just 3MB of L3 the performance impact is just 8% over the performance with 12MB L3.

6.4 Query temporal locality

In this section we will show an optimization technique which intuitively seems correct but when applied it doesn't work. One of our ideas for optimizing the index search performance was to optimize the query stream for temporal locality. The idea was that posting lists would be cached and this would speed up the execution. In Figure 32 we show the results of evaluation of this technique. Figure 32 shows the index search performance of 1) a sorted query stream, 2) the normal stream we used in the rest evaluation and 3) a stream which tries to maximize the distance between identical queries. One would expect that sorted stream would provide the best performance and the max distance stream would be the worst, due to good and bad temporal locality respectively; but the results do not show a clear trend and in general the differences between the three streams are very small. To conclude optimizing the query stream executed by an index server for optimal temporal locality is not something will benefit the performance and it doesn't worth bothering it with. The main reason this idea didn't work is because index search by default seems to have a very good caching behaviour as we showed in section 6.1, thus is difficult to see any benefit from this or any other caching technique.

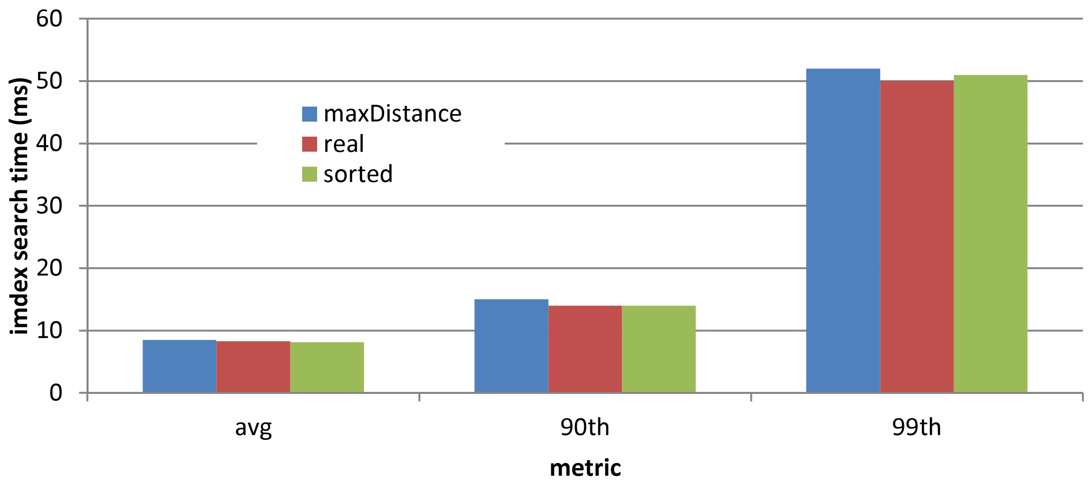


Figure 32 Performance of various streams with different query temporal locality. The sorted provides the best temporal locality, the max distance the worst and the real is somewhere in the middle. The results show that index search is insensitive of the temporal locality of the query stream.

6.5 Correlation with application level characterization

To conclude this chapter we will summarize the application level observations which are confirmed by the micro architectural characterization performed in this chapter. The compute intensive nature of index search and the small memory pressure explain the following observations: 1) In section 5.5 the performance ratio between the 2.4Ghz and the 1.6Ghz cores were exactly 1.5 (as the ratio of frequencies), this is a result of the fact that the memory time in index search is small, 2) In sections 5.4 and 5.5 intra server partitioning was not negatively affected by the fact that we were using multiple index server processes, this probably would not be the case if web search was a memory intensive process, 3) also due to the small memory pressure in 5.2 we did not observe a big response time degradation when increasing the number of active cores and in 5.6 caching was not the cause of performance variability for index search.

Also the studies we performed in this chapter namely prefetching evaluation, cache size exploration and performance sensitivity to query temporal locality, confirm the CPU intensive, low memory pressure nature of index search.

Chapter 7

How Representative is Nutch Benchmark?

This section briefly discusses the expected similarities and differences of the Nutch benchmark to commercial search engines. The benchmark architecture (frontend-index-document servers) is similar to what is described in [25] as the Google's query serving architecture although it is important to note that the exact query execution steps as well their implementation may be different. Maybe the main and most important difference is in the ranking functions. The Nutch benchmark uses the Lucene index searcher which uses the tf.idf scheme [11] while for example Bing search engine uses machine learning [19]. Despite the ranking function difference, the index search procedure used in Nutch is not completely different to Google or Bing. In particular, all of them perform posting list intersection which is a time consuming part of the search. Overall, we believe that this benchmark is an adequate proxy for studying web search performed in real deployments. A fact that enhances this statement is that Lucene (the Nutch benchmark's indexer) is used by many corporations. For instance, the Twitter's real time search [26] is built upon Lucene.

Chapter 8

Conclusions / Future Work

8.1 Conclusions - Contributions

In this work we describe and characterize on real hardware an academically accepted Web Search Benchmark. To the best of our knowledge no other web search benchmark was proposed to the research community. The specific contributions of this work are: 1) We shed light on the benchmark's software architecture and functionality through application and micro architectural characterization, 2) we identify the limitations and differences of this benchmark as compared to commercial search engines, 3) we explore the possibility of using low power servers for web search as well as the effects of intra-server partitioning and 4) we study the sources of web search performance variability.

The key conclusions from our work are : 1) index search times scale linearly with the amount of index dataset, 2) summary generation processing time does not scale with the amount of dataset and remains constant, 3) index search is the most time consuming operation of the web search, 4) we motivate the use of low power servers for index search , 5) we show that intra-server partitioning can help tail latencies by sacrificing slightly the average response time at high incoming traffic, 6) the number of matching documents is the most reliable proxy to predict index search times and not the number of query terms, 7) the use of compression for storing the index plus the fact that the amount of data read for executing each query is small compared to the whole index data, explain the high IPC and the good caching behaviour of index search, 8) summary generation is also compute intensive operation with good caching behaviour, 9) index search does not benefit from hardware prefetching and it is an operation with low sensitivity to LLC size, 10) DVFS and background jobs are the stronger culprits for performance variability.

8.2 Future work

For future work we plan to run the benchmark on a real low power server, e.g. Atom or ARM, to determine if they could match the QoS of a high performance server and also perform a TCO comparison. Also, we plan to create a scheduler which will work better in conjunction with the intra-server partitioning. The scheduler will be aware of how well various terms posting lists are partitioned and aim to schedule the index search dynamically with different number of cores. Other research directions are the evaluation of index search cut off latency, the comparison of stress test traffic with real life query inter-arrival times and the comprehensive characterization of more cloud applications e.g. web serving, media streaming and etc. We also plan to add power measurements to the experiments performed in this thesis in order to make this work more complete. Also power measurements will enable us to evaluate: 1) how much energy we save from disabling prefetching, 2) The energy-efficiency of a technique which reduces the CPU frequency when we are ahead of the QoS target and increases the frequency when we are drifting away from the QoS target.

Last but not least we would like to evaluate how representative are the benchmark's inputs. For example in our evaluation we found out that the most time consuming queries are single terms queries with popular terms. In a realistic setup those queries would probably be captured by the memcache server and not reach at all the index servers [23]. [23] states that the queries which reach the index servers are mostly multi terms consisting of not so popular terms because the popular queries are captured by memcache. So an interesting question which we will aim to answer is how you should compose your query stream. From our experience during this work and the sources from related literature we can conclude that a representative query stream of real – life deployments given a crawled index would be multi term queries which take long time to complete and single term queries of not popular terms. Also that stream should not have consecutive identical queries assuming a memcache. So one of our future goals is to evaluate this benchmark using this kind of query stream.

References

[1] Badue, C., Baeza-Yates, R., Ribeiro-Neto, B., & Ziviani, N. (2001). Distributed query processing using partitioned inverted files. Computer Science, 1(3), 4.

[2] Barroso, Luiz André, Jimmy Clidaras, and Urs Hölzle. "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines." Synthesis Lectures on Computer Architecture 8.3 (2013)

[3] CloudSuite web search site http://parsa.epfl.ch/cloudsuite/search.html

[4] DeCandia, Giuseppe, et al. "Dynamo: amazon's highly available key-value store." SIGOPS 2007.

[5] Ferdman, Michael, et al. "Clearing the clouds." ASPLOS. 2012.

[6] Yang, Hailong, Alex Breslow, Jason Mars, and Lingjia Tang. "Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers."ISCA-40 2013.

[7] Hardy, Damien, et al. "An analytical framework for estimating TCO and exploring data center design space." ISPASS, 2013

[8] T. Hoff. Latency Is Everywhere And It Costs You Sales – How To Crush It, 2009. http://highscalability.com/latency- everywhere-and-it-costs-you-sales-how-crush-it

[9] Hölzle, Urs. "Brawny cores still beat wimpy cores, most of the time." IEEE Micro 30.4 (2010).

[10] Jeon, Myeongjae, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. "Adaptive parallelism for web search." In Proceedings of the 8th ACM European Conference on Computer Systems, pp. 155-168. ACM, 2013.

[11] LuceneScoringExplanation http://lucene.apache.org/core/4_0_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html

[12] Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze. Introduction to information retrieval. Vol. 1. Cambridge: Cambridge university press, 2008.

[13] Mars, Jason, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations." Micro-44,2011.

[14] Mars, Jason, and Lingjia Tang. "Whare-map: heterogeneity in homogeneous warehouse-scale computers." ISCA40, 2013.

[15] Meisner, D., Sadler, C. M., Barroso, L. A., Weber, W. D., & Wenisch, T. F. Power management of online data-intensive services. ISCA-38, 2011.

[16] numactl webpage http://linux.die.net/man/8/numactl

[17] Page, Lawrence, Sergey Brin, Rajeev Motwani, and Terry Winograd. “The PageRank citation ranking: Bringing order to the web.” (1999).

[18] Lotfi-Kamran, Pejman, et al. “Scale-out processors.”ISCA-39, 2012.

[19] Reddi, Vijay Janapa, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. “Web search using mobile cores.” ISCA37. 2010.

[20] Ren, Shaolei, Yuxiong He, Sameh Elnikety, and Kathryn S. McKinley. “Exploiting processor heterogeneity for interactive services.” USENIX, 2013.

[21] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. 2009

[22] Simics images page http://parsa.epfl.ch/cloudsuite/downloads.html

[23] Tatikonda, Shirish, B. Barla Cambazoglu, and Flavio P. Junqueira. “Posting list intersection on multicore architectures.” SIGIR 2011

[24] Pass, G., Chowdhury, A., and Torgeson, C. A picture of search. InfoScale ‘06

[25] Barroso, Luiz André, Jeffrey Dean, and Urs Holzle. “Web search for a planet: The Google cluster architecture.” Micro, Ieee 23.2 (2003): 22-28.

[26] Busch, Michael, et al. “Earlybird: Real-time search at Twitter.”, ICDE, 2012

[27] Xi, Huafeng, et al. "Characterization of real workloads of web search engines." Workload Characterization (IISWC), 2011 IEEE International Symposium on. IEEE, 2011.

[28] http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family.html

[29] Li, Jialin, et al. "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency." Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014.

[30] http://unixhelp.ed.ac.uk/CGI/man-cgi?top

[31] http://unixhelp.ed.ac.uk/CGI/man-cgi?ssh+1

[32] https://perf.wiki.kernel.org/index.php/Main_Page

[33] http://www.acpi.info/

[34] http://linux.die.net/man/8/cpuspeed

[35] http://curl.haxx.se/docs/manpage.html

[36] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. 2011. The impact of memory subsystem resource sharing on datacenter applications. In Proceedings of the 38th annual international symposium on Computer architecture (ISCA '11). ACM, New York, NY, USA, 283-294. DOI=10.1145/2000064.2000099

[37] Mazouz, Abdelhafid, et al. "Evaluation of CPU frequency transition latency." Computer Science-Research and Development 29.3-4 (2014): 187-195.

[38] Ezolt, Phillip. "A study in Malloc: a case of excessive minor faults."Proceedings of the 5th Annual Linux Showcase and Conference. 2001.

[39] http://linux.about.com/od/commands/l/blcmdl1_ps.htm

[40] http://www.unix.com/man-page/linux/1/chrt/

[41] Kambadur, Melanie, et al. "Measuring interference between live datacenter applications." Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press, 2012.

[42]https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/store/DataOutput.html#writeVInt(int)

[43] Characterization and Analysis of a Web Search Benchmark,
Z. Hadjilambrou, M. Kleanthous and Y. Sazeides,
IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, March 2015

[44] The Implications of Different DRAM Protection Techniques on Datacenter TCO, P. Nikolaou, Y. Sazeides, M. Kleanthous and L. Ndreu, The 11th Workshop on Silicon Errors in Logic - System Effects (SELSE-11), March 2015

[45] https://www2.cs.ucy.ac.cy/carch/xi/