Diploma Project

PERFORMANCE CHARACTERIZATION OF A SIMULTANEOUS MULTI-THREADING (SMT) AND INDEX PARTITIONING (IP) FOR AN ONLINE DOCUMENT SEARCH APPLICATION

Georgia Antoniou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2019

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Performance Characterization of a Simultaneous Multi-Threading (SMT) and Index Partitioning (IP) for an Online Document Search Application

Georgia Antoniou

Professor

Yiannakis Sazeides

The personal dissertation is submitted in partial fulfillment of requested obligations for receiving the degree of computer science from the department of Computer Science of the University of Cyprus.

May 2019

Acknowledgments

At this point I would like to express my deepest appreciation to all those people who helped me complete this work and made this journey unforgettable. This thesis would not have been possible without their valuable advices and their relentless support. I have matured so much both academically and personally because of them.

At first, I would like to give special thanks to my Professor Mr. Yiannakis Sazeides, for the opportunity to work with him on such an interesting topic. His constant belief and trust in me as well as his constructive criticism helped me overcome every obstacle that appeared during this project. Always maintaining a great atmosphere during meetings and managing to influence and motivate me to work harder and never stop believing in my dreams. Most importantly though I want to thank him for the endless amount of time he spent to teach me how to work right and for helping me understand how I must approach each problem that comes in my way.

I would like to extend my sincere thanks to the amazing team that Mr. Sazeides created Xi Lab. Starting with my supervisor Zacharia Hadjilambrou I want to thank him for the great patience he showed me and for all the practical suggestions he offered that often led to the solution of my problems. I would also like to show appreciation to Panayiota Nicolaou for helping me understand the importance of the team, the importance of listening and lastly the importance of helping others as she did many times with me. I would also like to thank my fellow students for their encouragement and constant support during the whole period of this project.

Lastly, I want to thank my parents George and Evgenia and my siblings for their profound belief in my abilities and in my work. Their trust and wavering support along with their advices are what keeps me going. Thank you for believing in me and my decisions and for never leaving my side even when things get hard. To my friends you offered me the most unforgettable memories and your friendship I will cherish forever.

Abstract

The number of internet users has increased dramatically during the last decade going from 30% to 45% of the population of the world. Technological solutions along with the fact that internet comes with services that optimize everyday procedures helped to this great expansion. This great demand though led many service providers to build datacenters in order to maintain high user experience and in parallel deliver their services to a global base client. But the building of datacenters comes with a lot of cost as a big number of devices have to be bought run and maintained. The greatest example of such a service is web search. Web search is one of the most known services with more than 5.6 billion searches per day. That along with the fact that the service must return relevant to the user queries results in a fraction of time automatically made the service, processing power expensive. For these reasons the specific service has been in the center of the research interests. In this work we report the results of a performance characterization of Simultaneous Multi-Threading (SMT) and Index-Partitioning (IP) when executing an online document search application which has similar characteristics as a web search machine. We use an academically accepted web search application and evaluate the specific application on real hardware. More specifically we start by explaining the background of the project which include the main components, the architecture of the benchmark, SMT, IP e.t.c Then we explain the way that we collect the data by introducing the four different system configurations each one representing a potential way of running the application. We continue doing an experimental evaluation. Our experimental evaluation shows that while SMT execution degrades single-thread execution latency, the multiple SMT contexts increase available throughput and can help decrease queuing latency. SMT is particularly effective in reducing the queueing induced by IP. Overall, we find that in every situation we have evaluated combining SMT and IP yields the best average and tail latency for the application, dataset and server type used in this study. This is true for both single and dual socket server configurations we have evaluated. Our analysis for the dual socket configuration reveals that IP is beneficial for both low and high utilization. The thesis also reports, among other, on the sensitivity (i) of tail latency to performance variability, and (ii) to the system configuration (c-states and pinning).

Contents

Chapter 1

Introduction

1.1 Motivation

Simultaneous multi-threading (SMT) [32] is a common feature of modern server CPUs [16][30]. Broadly speaking, SMT increases CPU utilization and computational throughput by concurrently executing multiple threads in a physical core while sharing many of the core resources (such as physical registers, execution units, caches and predictors). On the down side, SMT is detrimental to single thread performance because the simultaneously executing threads often content for limited shared resources both inside and outside of a core [29]. For latency sensitive workloads, such as interactive online applications, SMT induced single-thread performance degradation seems undesirable because it can lead to higher execution latencies [34][33].

Related work as well as anecdotal sources suggest that whilst most server CPUs support SMT, the datacenter and HPC operators often prefer to disable SMT or not utilize all available SMT contexts [34][33][19]; especially, for latency sensitive applications that need to respond quickly to incoming queries. This leaves the CPU hardware underutilized and inefficiently used in costly and power-constrained compute farms and centers. Previous work has attempted to identify latency-friendly ways to collocate batch and online workloads on SMT contexts to improve server utilization without violating QoS requirements (such as tail latency) [34][33]. Index-partitioning (IP) is a prevalent technique for increasing parallelization and reducing index search-time [17][18][31][7]. Across-server partitioning is essential when large working sets need to be searched [2][3]. This requires individual server tail latency to be much lower than the overall tail latency [8]. Intra-server IP, as the means to reduce a server tail latency, is the subject of several previous studies. Virtually all these works find that IP is beneficial to

average and tail latency at low utilization but has diminishing returns with higher load [17]. To leverage the benefit of IP at low-utilization some works propose dynamic adaptation of the degree of partitioning used according to a predicted load [17][7].

As far as we know, no previous published work has investigated the latency ramifications of using all SMT contexts of a core in combination with intra-server IP, when executing an online sensitive application. What is more important, though, is that our study has led to new insights about how to use SMT and IP to improve latency of an online time sensitive application.

The key finding of our study is that a CPU running an online application that has SMT enabled and uses IP provides the best average and tail latency for the application, input data, all query arrival rates and server configurations used in this study. This is in direct contrast to the commonly accepted view that SMT is not latency friendly for online applications and that IP provides latency benefits mainly at low arrival rates when CPU utilization is low. The key insights of our study, supported by both theoretical queuing analysis and empirical results, are:

1. The doubling of execution contexts by enabling SMT can dramatically reduce the queueing time of queries waiting to be serviced.
2. IP reduces the latency a CPU spends executing a query but it offers limited or no performance at high arrival rates due to large queueing delays.
3. Combining SMT and IP provides best average and tail latency by leveraging both reduced query execution and queueing times.

The CPU used in this study supports two-way SMT per core.

The gains from combining SMT and IP are dependent on:

a) the OS scheduler to be SMT-aware, which means that it will usually not schedule two threads to the same physical core when the number of runnable threads is less than or equal to the number of physical cores (which is true for most modern OS) [10],

b) the average increase in execution time (performance degradation) due to SMT needs to be offset by a larger average reduction in queueing time due to the additional throughput offered by SMT contexts. This trade-off is workload dependent and for the online application we evaluated it holds. Specifically, the single-thread performance degradation caused by SMT is 1.5X and the doubling of the SMT contexts offsets it.

c) The size of the partitions used by IP are balanced and enable a significant reduction in the execution time as compared to no partitioning. For our workload and index

dataset, 2-way IP reduces execution time by almost half indicating well-balanced partitions.

1.2 Contributions

The main contributions of this work are a characterization analysis and a number of conclusions that stem from investigating the implications of SMT and IP on the response-time of an online application. In particular, the main contributions of this work are:

1. A characterization of the SMT and IP ramifications in terms of average and tail latencies of the popular Lucene search-engine, when running on a dual-socket NUMA x86 server with multi-core CPUs, with two-way SMT cores.
2. A design space exploration of the implications among other of: (i) changing query inter-arrival rate, (ii) single and dual socket execution, (iii) CPU c-states configuration, (iv) execution with and without SMT, (v) execution with and without IP of the application's index dataset, (vi) the mapping of multiple application instances to cores, and (vii) the mapping of the data set for the dual-socketed execution.
3. An analysis based on queueing M/M/c model that provides the theoretical explanation for the observed average latency reduction due to SMT and IP.
4. Showing that SMT can reduce both average and tail latency of the Lucene search-engine at high utilization. This is when the query traffic is large enough so that the number of queued + serviced queries is larger than the number of physical cores. The analysis reveals that the SMT expected time degradation of a query is 1.5X. Therefore, the observed benefits come from the 2X throughput increase of SMT and the reduction in queuing times.
5. Observing at low utilization that SMT provides the same average and tail latencies as with the SMT disabled. This is due to the default OS scheduling policy of our system that is SMT-aware. It typically assigns one job to all physical cores before assigning a second job to a physical core. This avoids performance degradation caused by naively scheduling two threads on a physical core when other physical cores are idle.
6. Analysis of two-way IP across physical cores. The IP analysis across physical cores reveals that partitioning cuts execution time by half but its overall benefits diminish as query arrival rate increases because the query execution time speedup is offset by

queuing increase (similar to what prior work has shown [17][7]). This observation is true for single-socket runs. For dual-socket execution, the IP configuration provides the lowest latency across all utilization levels compared to a configuration without IP. This is the case because IP eliminates NUMA (Non-Uniform-Memory-Access) latency by restricting each search-server and its index-data to run on a dedicated socket and its local memory node.

7. Finally, most importantly showing that the combination of SMT with intra-server IP provides the best average and tail latency for both single-socket and dual-socket configurations for all inter-arrival rates we have evaluated.

Chapter 2

Background

2.1 Definitions

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

Query: Query is an expression composed by one or more terms. The queries are relative to the type of documents the user is searching for. In our evaluation we only have conjunctive queries, which means that the resulting documents must have all the terms of the query.

Index: Structure that contains the term and the relevant to the term documents. In more detail each term points to a list of all the documents that contain the term along with the number of times the term is being mentioned at those documents (Frequency). The terms are alphabetically sorted.

Posting List/ Document List: The list that the term points to. The list consists of pairs of document id and term frequency that are called posting.

Tf.idf Scoring (term frequency-inverse document frequency): It is a numerical statistic indicating the relevance between a word and a document. To do that it uses the frequency of the term in the document and the number of documents in a set that contains the term.

Partition: A part of the index. We usually assign different part of the index (partitions) to different index servers to enable parallel index search.

Intra Server Partition: All Index servers are in the same physical machines.

Tail latency: It's a metric used when evaluating the performance of a web search engine indicating the performance of the worst-case queries which are the one requiring

the most time. It is measured in various percentiles. For example, a 90th percentile equals to 250 ms means that the 90% of the executed queries are executed in less than 250ms.

Quality of Service: A set of thresholds for metrics need to be respected in order to have user satisfaction. In web search it is usually expressed using percentile of response time.

Response Time: It is the time between the moment the client sends the query to the web engine until the moment the client receives the results. It is an important metric as it is being used to calculate the tail latency.

Throughput: It is the number of queries the web service can serve per amount of time. From application to application the throughput differs. For example, in networks is the number of packets arrived at the destination per amount of time. In our case we measure throughput as queries/second.

C-States: They are power states of the CPU. The first state is called C0 and it is the state where the CPU functions normal (no energy savings). The higher the C number the deeper the sleep mode and the higher the delay until the CPU "wakes up" again and is 100% functional (go to C0 state). The idea of lower power states is to shut down or reduce the power going to the units of the CPU that are idle. This is being achieved by cutting clock signal or reduce voltage or both.

NUMA effect (Non-Uniform Memory Access): We come across this phenomenon on multi-socket architectures. At a multi-socket architecture we have multiple sockets (CPUS). Each CPU(socket) is nearer to a part of memory(called local memory). Ideally the CPU fetch data from it's local memory but it can also fetch data from a "remote memory" (part of memory which is nearer to another CPU). At this case the CPU performs a remote memory access which cost more cycle delay than the local memory access.

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

2.2 Simultaneous Multithreading

SMT is a characteristic of modern CPUs first appeared on intel Xeon processor family in 2002. The motivation for the development of this characteristic was to improve the overall performance of the CPU without increasing transistors (die area). So the idea of the technique is to increase the number of threads assign to one physical core (without increasing the number of all units per physical core) in order to increase the utilization and the throughput. Without the SMT, threads run sequentially, but the performance is not ideal as hazards prevent instructions from executing leaving underutilized some units.

Microarchitectural a CPU that supports SMT has small differences from a CPU that doesn't support SMT. This is because the threads running on the same physical core, either share the structures of a single physical core or use them sequentially. The number of duplicated units that a single physical core has when support SMT is small.

Figure 2.1 illustrates an abstract state of the pipeline when two threads A and B run on a core with and without SMT. When SMT is enabled we observe that the pipeline utilization is increased because the threads can use different core units simultaneously. Generally, the threads running on the same physical core don't execute completely in parallel as they share some units and are executed sequentially in some other units of the pipeline. More specifically when SMT is enabled the instructions per cycle (IPC) is increased but the IPC per thread is reduced due to resource contention. With online services the response time of each query is important so SMT is considered detrimental for the latencies of the queries.

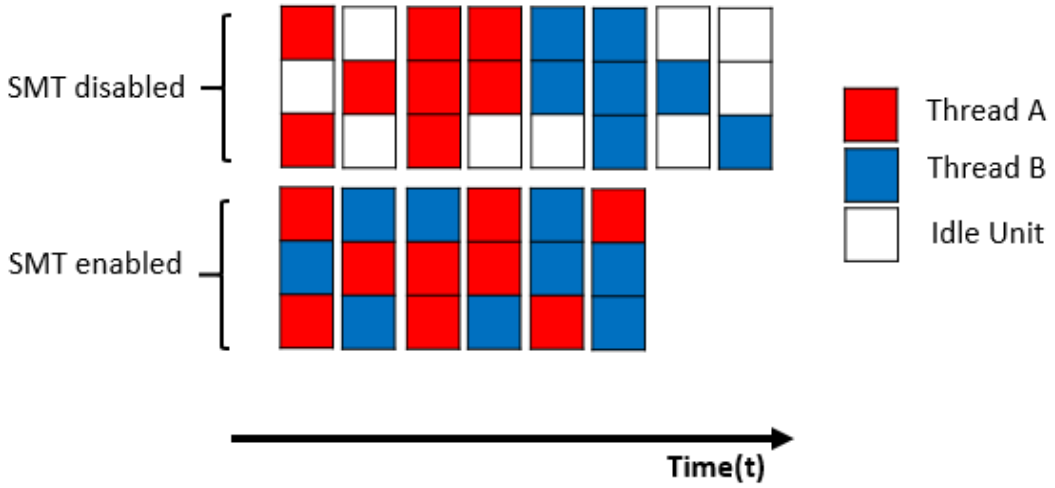


Figure 2.1 Pipeline state with and without SMT.

Most modern server CPUs offer two-way SMT [23][16][30], this means that up to two threads can simultaneously execute and share a core's resources. The impact of SMT is not always beneficial, even for throughput-oriented batch workloads (e.g. HPC workloads). Previous work shows that some applications see performance improvement with SMT and others do not [29][4]. Generally, SMT is considered beneficial if its single-thread performance degradation is lower than the increase in the number of logical cores.

We can perform several optimizations to improve the performance of SMT at application and OS level. Something that we came across in this work is the SMT – aware scheduler of the operating system. An SMT – aware scheduler first assigns threads to physical cores before start to assign to their SMT context. In this way in low utilization all threads of the system are running on different physical cores having all the resources available.

2.3 Queuing Theory

This subchapter uses the M/M/c queueing model [20] to evaluate analytically the average performance (e.g. the expected wait-time of a query in a server) of a server when it is used to service an incoming query stream. The model has three input parameters: the incoming query rate, λ, the expected number of queries a server core can service per second, μ, and the number of cores in the server, m. The model assumes that the arrivals follow a Poisson distribution and the service times are exponentially distributed. In our study, a Poisson process governs query arrivals, therefore, the model is a good match for what we evaluate. We make a simplifying assumption that service times are exponentially distributed. A more detail model of service times is beyond the scope of this work, but we like to point-out that the model provides a good basis for understanding the trade-offs and the sources of improvements from SMT and IP.
We analyze the following four server configurations using different values for the model parameters:
1. a server without SMT and without IP (noSMT-noPart or Baseline)
2. a server with SMT and without IP (SMT-noPart)
3. a server without SMT and with IP (noSMT-Part), and
4. a server with both SMT and IP (SMT-Part).

The noSMT-noPart server uses m cores each with expected service rate μ. The SMT-noPart configuration use 2m cores, to resemble 2-way SMT core design like in the real hardware evaluation of our study, with service rate μ/d, where d represents the SMT single-thread degradation. The noSMT-Part configuration uses m/2 cores each with 2μ service rate. This is used to represent a server employing 2-way partitioning, each query serviced by two-threads, assuming perfectly balanced partitions. Finally, the SMT-Part server uses m cores each with 2μ/d service rate.

Figure 2.2 shows a breakdown of average response time into wait and search times for the four server configuration assuming λ=150 and with the noSMT-noPart configuration using 8 cores, m=8, and each core capable of serving on average 20 queries per second, μ=20. Figure 2.2 represents a scenario where SMT is assumed to cause 1.5X single-

thread degradation, d=1.5, which corresponds to the actual degradation we have observed in real hardware for the application we evaluate.

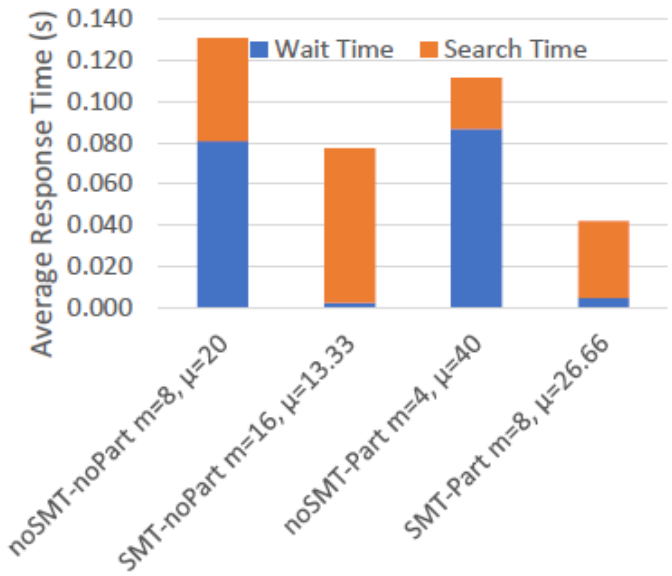


Figure 2.2 Search and Wait Time for different Configurations (λ=150, d=1.5X)

The analysis clearly shows that the Baseline, for the given configuration and arrival rate, suffers a lot of queueing delay (a query waits 80ms out of 130ms to be serviced). SMT-noPart virtually eliminates the queueing time of the baseline underlying the benefits of doubling the throughput in a queue-constrained situation while the search time suffers an increase due to SMT's single-thread degradation. The noSMT-Part configuration reduces the search time, but this comes at the expense of an increase in wait-time. This indicates that when arrival rate is high a reduction in search time has diminishing returns (IP is more useful at low utilization). In fact, the queuing time of noSMT-Part is worse than in the Baseline, with the noSMT-Part scenario been the most constrained by queueing. Finally, the SMT-Part configuration provides the lowest overall response time since its extra cores, as compared to noSMT-Part, help remove essentially the queueing delay. The search time is worse, due to SMT single-thread degradation, but this is offset by the drastic reduction in queueing time.

One other interesting observation from Figure 2.2 is that the SMT-Part configuration offers more latency reduction than the sum of the reductions by SMT-noPart and noSMT-Part (89ms vs 53+19=72ms. Put it another way the reduction of queueing delay by SMT is not a linear function of the extra cores used. This seemingly counterintuitive

result can be understood by considering what the impact is on queueing time of increasing core count. These trends are shown in Figure 2.3 and 2.4 for the SMT-noPart and SMT-Part configurations. The graphs reveal that queueing delay, for the scenario analyzed, decreases exponentially with the number of cores and doubling the number cores can eliminate queuing delay.

The SMT single-thread degradation in Figure 2.3 and 2.4 is a linear function of extra SMT cores ranging from d=1X, no degradation, when no SMT cores are used to a maximum of d=1.5X, when the number of cores is doubled. This corresponds to the actual degradation observed in the real evaluation on hardware.

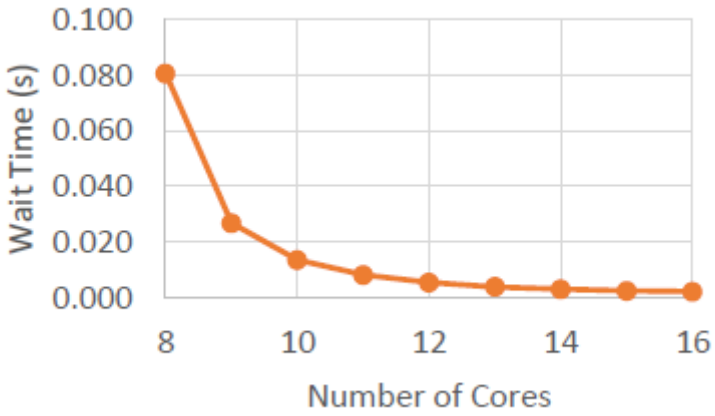


Figure 2.3 Wait Time vs # of Cores SMT-noPart λ=150

The model used in this chapter is quite useful in revealing the potential gains from combining SMT and IP and explain the sources of such improvements. But the model is an approximation of reality. For instance, real scheduling will be SMT-aware, not scheduled on same physical core two threads unless no physical core is available. Additionally, partitioning is not perfect and thread execution is asynchronous, implying some extra overheads from partitioning in realistic setups. Consequently, for the aforementioned and other model inaccuracies, to assess the potential benefits of SMT and IP we perform an evaluation using real hardware for a specific online application.

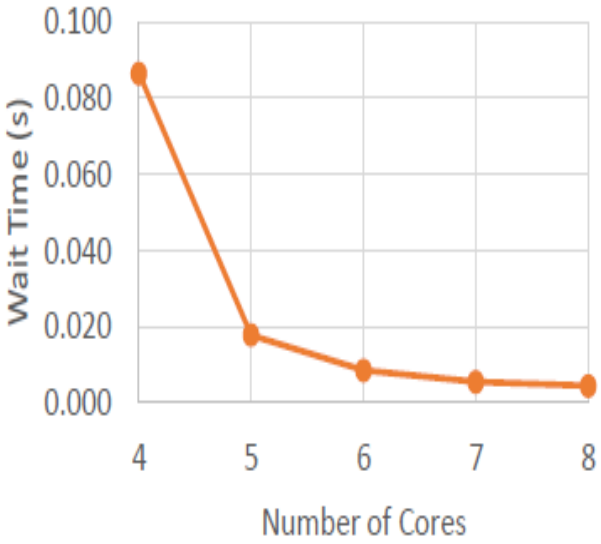


Figure 2.4 Queueing Time vs # of Cores SMT-Part λ=150

2.4 Sources Of Variability

Interactive applications often perform on many servers in order to take advantage of parallelism and maintain low latency. An example of this is web search which often performs index search across many servers in parallel. This is not necessarily good as slow down of a single server affects the tail latency of the whole query and indirectly the user satisfaction, diminishing the performance gains of parallelism. There are many sources of variability from the hardware, OS to application level that affect tail latency [tail of tales]. Web search providers search ways to reduce the impact of variability in order to guarantee a strict QOS usually 99% queries under 300ms. For these reasons at this chapter we are addressing the sources of performance variability and at a later chapter we introduce how we manage to reduce some of them in our experimental setup.

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 practice in data centres to

run more than one application per server node or for QoS purposes do keep underutilized a server by not using all its cores . Both practices 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 (C- states, 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 apply 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.

Chapter 3

Web Search Description

3.1 Benchmark Description

3.1.1 Architecture – Functionality – Information Flow - Components Description

In this chapter we describe the architecture and the functionality of the Nutch web search engine. The Nutch framework provides a lot of useful functionalities such as: a) crawl and build index, b) partition the index among multiple index servers, and c) coordinate index server execution with a frontend server that forwards the search requests to multiple index servers, collects their answers and sorts them. Fig. 3.1 shows the components and the overview of the search engine's architecture. The Nutch search engine structure consists of the front-end and index and, it is similar to what is described [1] in as the Google query serving architecture. The Nutch index search is based on the Lucene search engine which is a well known search engine. For instance, the Twitter's real-time [24] search is built upon Lucene. Therefore, this benchmark is representative as it has some real-world deployments. It must be noted though, that compared to the most widely used search-engines like Google or Bing, the search-engine used in this thesis has some differences mainly in document scoring and query processing .

Below we provide a description of the query-processing flow that is also illustrated in Fig. 3.1: 1) The client sends a query to the front-end server. 2) The front-end server 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 front-end server collects the results, sorts the documents according to their relevance score, assembles an html response and sends it to the client.

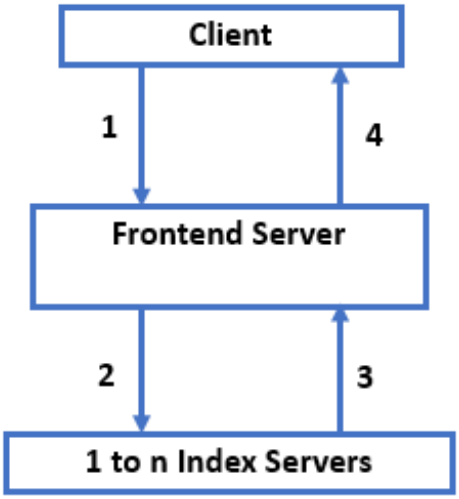


Figure 3.1 The basic components of our search infrastructure. The arrows show the information flow and interactions between components during a query execution. The numbers show the order of the execution flow.

Having that in mind a query's end-to-end latency is the sum of: 1) the time spent on the 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 which includes the time frontend spends on sending the query to the index servers, gathering the responses and sorting the documents by relevance score. Figure 3.2 shows on average how much time each query phase takes in relation to dataset size. This graph is obtained by executing sequentially 100K queries and calculating the average time spent at each phase. The sequential execution is enforced to isolate the latencies from the effects of queuing, contentions on shared resources, etc. The key takeaway from Figure 3.2 is that the client-frontend communication requires minimal

time and that index search time scales linearly with dataset. We provide next a more detailed component description.

Index Server The index server is a Hadoop 0.2 IPC (Inter Process Communication) server process running Lucene 3.0.1 search engine. The Hadoop IPC server consists of: i) a listening thread which listens for incoming requests from the front-end server, ii) the handler threads, which perform the index search or retrieve the details of a document and iii) a responder thread, for sending the responses to the front-end.

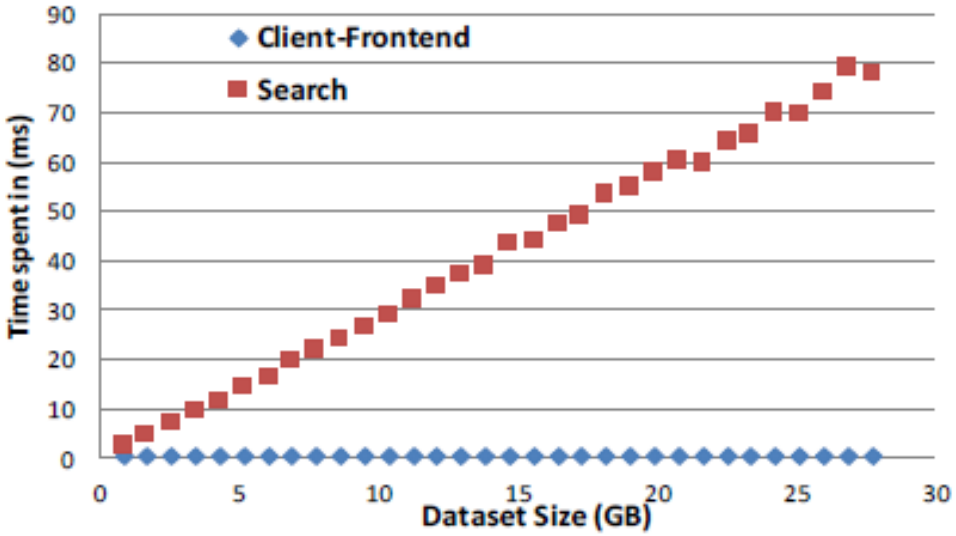


Figure 3.2 Time spent at client-frontend communication and search process of the benchmark in relation to dataset size.

Front-end Server The front-end is a Tomcat web server running the Nutch application. Tomcat is multithreaded and spawns a new thread for handling a new query request. The front-end 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 process thread used for sending queries to the front-end. A client thread can send queries based on some inter-arrival time distribution (open-loop) or in a

stress test manner (closed-loop). In the stress test scenario a client thread sends a new query as soon as it receives the response for the previous query sent. Having discussed the architecture and the information flow of the search-engine we next describe its inputs, i.e. the queries.

3.1.2 Index Structure

Next let's discuss the implementation of the index organization. The structure of the index is also illustrated in Figure 3.3. The index terms are stored in an array in alphabetical order. The alphabetical ordering enables binary search and fast term searching. A parallel array holds pointers to a byte stream; each pointer points to the position in the byte stream where the <documentId,termFrequency> pairs of a term start. The list of a term's <documentId,termFrequency> pairs is called posting list. The documentId and termFrequency pairs are compressed using a variable integer format. The variable integer format enables saving space by using the first bit of each byte to show 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

Term	Document Frequency	<Document id, Term Frequency>
Characterization	30	<1,7>,<45,21>,<336,2>
Thesis	200	<5,45>,<7,50>,<36,40>,<78,65>
Workload	144	<4,44>,<89,25>,<100,25>,<120,25>,<400,25>

Figure 3.3 High level view of Index structure

represented with one byte, all numbers from 128 to 16383 with two bytes, e.t.c. The posting list can be sorted by score, for example the PageRank. 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 Nutch all posting lists are sorted by DocId which is useful for performing efficient merging of posting lists for conjunctive (AND) queries (posting list intersection).

3.1.3 Index Search Procedure

Now let's discuss the index search procedure. Nutch uses conjunctive multi-term queries, vector space model (for representing documents) and tf.idf weighting scheme (for ranking documents). The actual search procedure goes like this: First binary search is performed to find the posting lists for all query terms. Then posting list intersection is performed to find the documents that contain all query terms. For each document found, a tf.idf score is calculated. The tf.idf weighting scheme gives high scores to documents that have many occurrences of the query terms and also to documents that contain many occurrences of rare terms. To decide the top-k (e.g. top-10) most relevant to the query documents, the index server sorts the documents based on their tf.idf score. To summarize, the index search time for a conjunctive query is determined by: 1) the binary search for finding each term's posting list, 2) the merging of the term posting lists, 3) the ranking of the documents which come out of the intersection, and 4) the sorting of the documents by their relevance rank.

The posting list intersection is generally considered the most time-consuming part of the index search procedure, not only for Lucene but for other search engines as well. In order to optimize this part many web search engines usually perform an early termination of the search procedure either by using a cut-off latency or when the quality of results is unlikely to improve with further searching. This strategy aims to avoid cases where index server is searching for too long. The Nutch provides an option to stop searching according to a cut-off latency. With enabled cut-off latency search, queries that are prematurely stopped may suffer from poor result quality (low relevance of the search results). For instance, this may happen when an index server is slowed down for some reason (see sources of variability) and does not have enough time to go through its posting list. In practical terms, it is not trivial to compare server performance in terms of relevance of query results. Given that optimizations that expedite search time can be beneficial to deployments with and without cut-off, we use for our evaluation the default configuration of Nutch benchmark which does not use any cut-off latency. Consequently, the various server configurations we assess, are compared using latency metrics (both average and percentile).

3.2 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. 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.

3.3 Index Partitioning

As mentioned at a previous chapter the most time - consuming phase in our search procedure is the posting list intersection. For this reason, many techniques have been invented to optimize this phase. One of them is the partitioning. The partitioning improves the end to end latency of the query and not the relevance of the results. It is based on the observation that index search time scales linearly with dataset. The idea behind it is to have multiple index server processes search at a different part of the dataset. In this way each index server process searches a smaller part of the dataset (smaller time spent to posting list intersection) while at the same time taking advantage of the parallelism. We have two types of partitioning, term and document partitioning.

With **term partitioning** each index server holds an index for a disjoint set of terms. A query is sent only to the nodes that contain a term of the query in their part of the index. In this way we are taking advantage of the concurrency as with different query terms we hit different nodes. The disadvantage of this is that if all the terms of the query are in the same index then only one node is being used and we lose the benefits of the concurrency. An example of term partitioning is shown in Figure 3.4.

Partition 1

Term	Document Frequency	<Document id, Term Frequency>
Characterization	30	<1,7>,<45,21>,<336,2>
Thesis	200	<5,45>,<7,50>,<36,40>,<78,65>

Partition 2

Term	Document Frequency	<Document id, Term Frequency>
Workload	144	<4,44>,<89,25>,<100,25>,<120,25>,<400,25>

Figure 3.4 High level view of term partitioning

With **document partitioning** each index server holds an index for a disjoint set of documents. Whatever the query is and the number of its terms the query will be sent to all of the nodes. According to theory, this document distribution results in a more uniform distribution of query processing time across index servers. We use document partitioning with our benchmark. An example of term partitioning is shown in Figure 3.5.

Ideally the more we increase the number of index servers and the degree of partitioning the less the response time. For example if with one partition we have a response time of 88ms, ideally with two partitions we must have a response time around 44ms. This happens because each index server gets a smaller index part and, thus, needs less amount of time to respond to a query. For this to happen though we must have balanced posting lists across the partitions, otherwise, partitioning will not provide the expected performance benefit. Apart from that the execution of the query must start at the same time for all index server process.

Index partitioning can be done across servers or inside the same server (intra server partitioning). Intra server partitioning is realized by running multiple index search contexts on the same server with each context working on a different index part. Intra server partitioning represents a trade-off between throughput and response time latency. Having many index searchers in a CPU socket can speed-up the execution of a query but reduces the number of available cores for handling in parallel multiple queries and can, thus, increase queueing time. In our evaluation we perform intra server partitioning.

Partition 1

Term	Document Frequency	<Document id, Term Frequency>
Characterization	28	<1,7>,<45,21>
Thesis	95	<5,45>,<7,50>
Workload	69	<4,44>,<89,25>

Partition 2

Term	Document Frequency	<Document id, Term Frequency>
Characterization	2	<336,2>
Thesis	105	<36,40>,<78,65>
Workload	75	<100,25>,<120,25>,<400,25>

Figure 3.5. High level view of document partitioning

Chapter 4

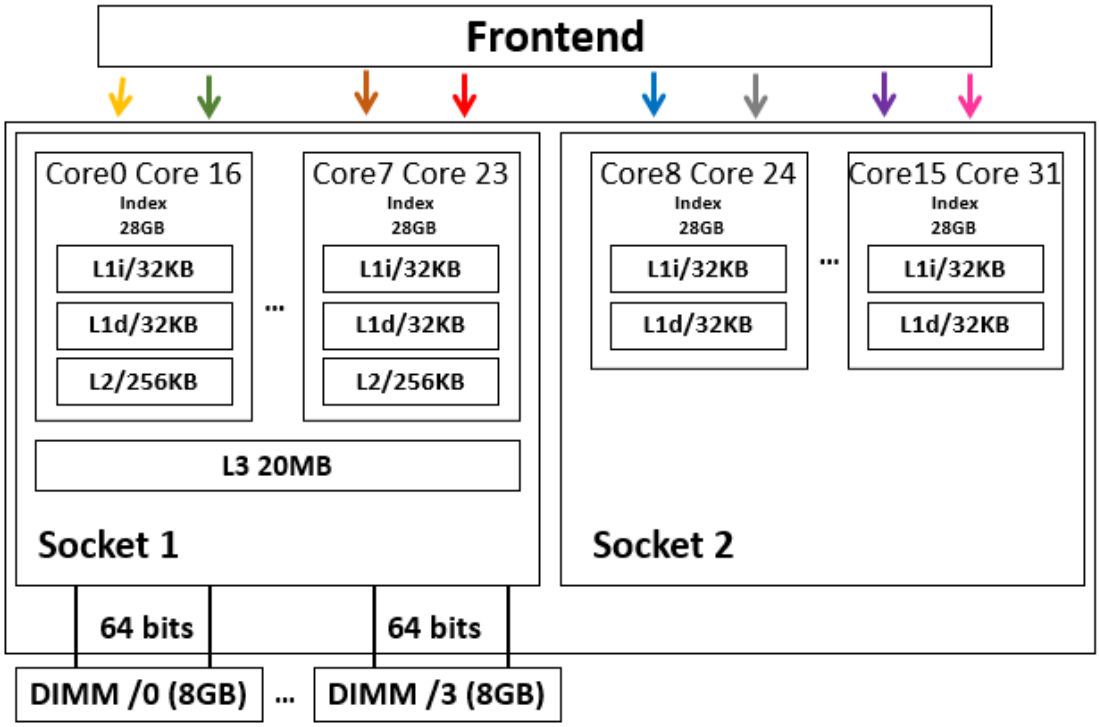
Design Space

4.1 Scenarios Description

In order to test the impact of Index Partitioning and Simultaneous Multi-Threading on the performance of our benchmark we created and evaluate four configuration scenarios. In this chapter we are going to present these scenarios and justify our choices for the configuration. The four configurations scenarios are: a) the NoSMTnoPart configuration in which both SMT and IP are disabled, b) the Part configuration in which IP and parallel search among physical cores for each query is enabled c) the SMT configuration in which SMT is enabled, and d) the SMTpart configuration in which SMT is enabled along with IP across SMT contexts.

The physical representation of the four configurations for dual sockets architecture are illustrated in Figure 4.1 and 4.2. The NoSMTnoPart scenario is very similar with the SMT configuration with the difference that SMT configuration offers more contexts for serving independent queries (of-course at the expense of slower execution time per query). The idea of these scenarios is the same, each physical or logical core is serving one query. Similarly, the Part configuration is similar to SMTpart with the difference that the SMTpart uses all SMT contexts not just physical cores (in this way is serving more queries in parallel). The idea in these scenarios is each core serving half of the query in half of the time. SMT compared to NoSMTnoPart can help reduced queueing time since it can serve more independent queries at any given moment. As it is being observed and below in our case the extra contexts are more helpful in reducing queuing time than the single thread speedup that noSMT scenario has. On the other hand, partitioning increases queuing (As for each query we have double number of cores

SMT



NOSMTnoPart

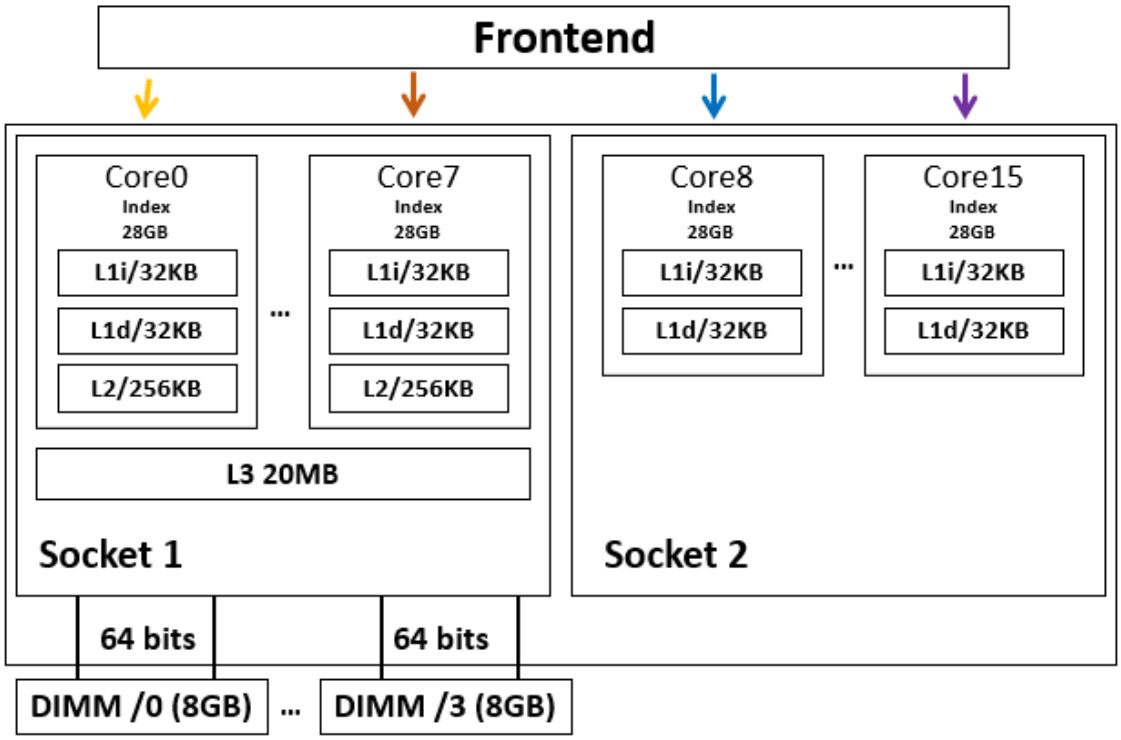
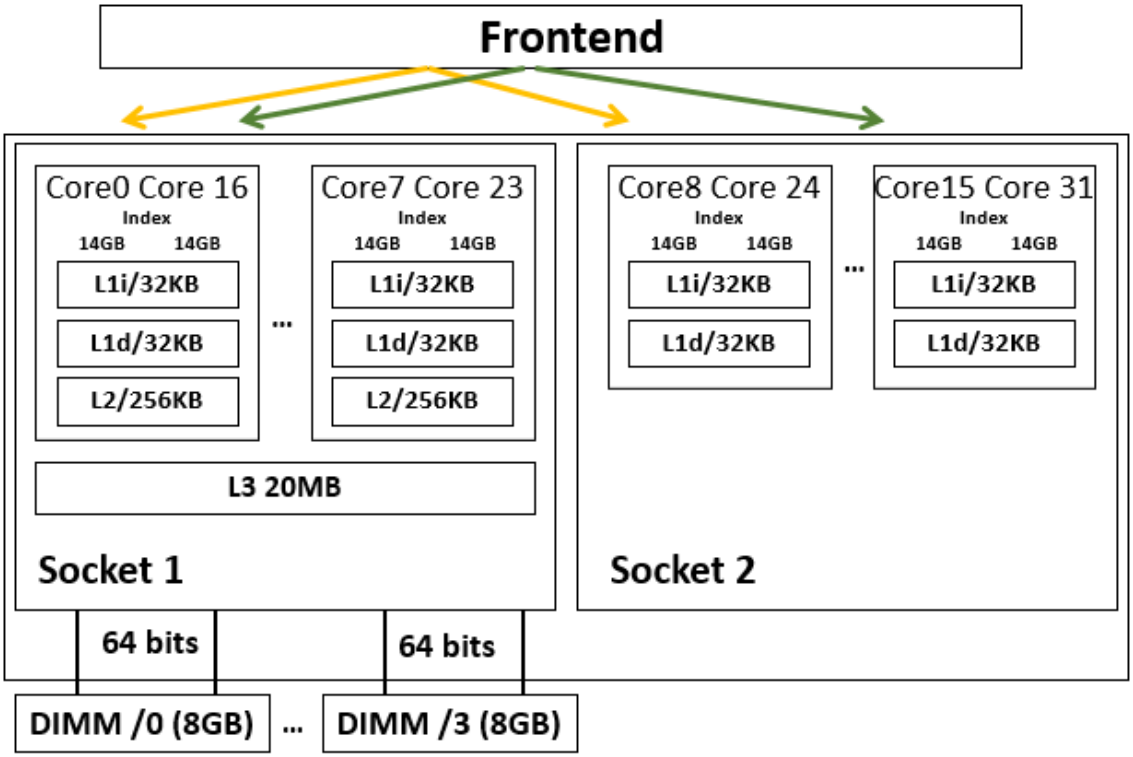


Figure 4.1 Illustration for design space exploration for no partition configurations

SMTpart



Part

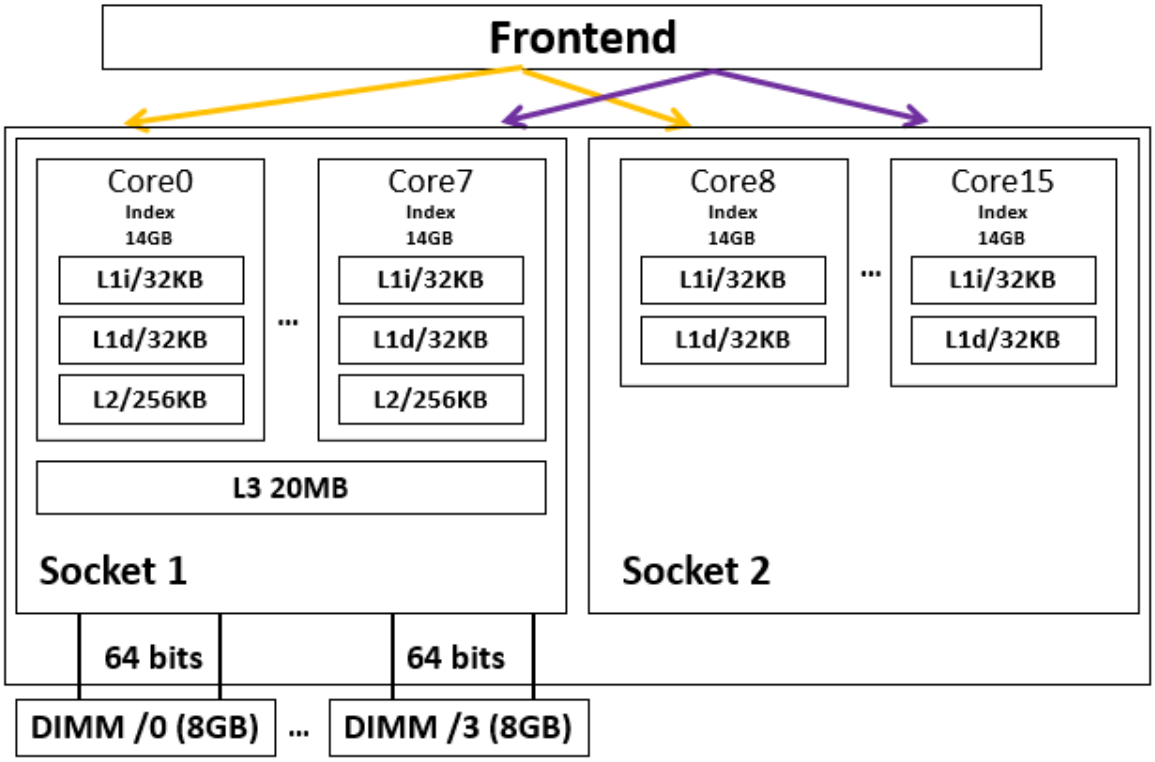


Figure 4.2 Illustration for design space exploration for partition configurations

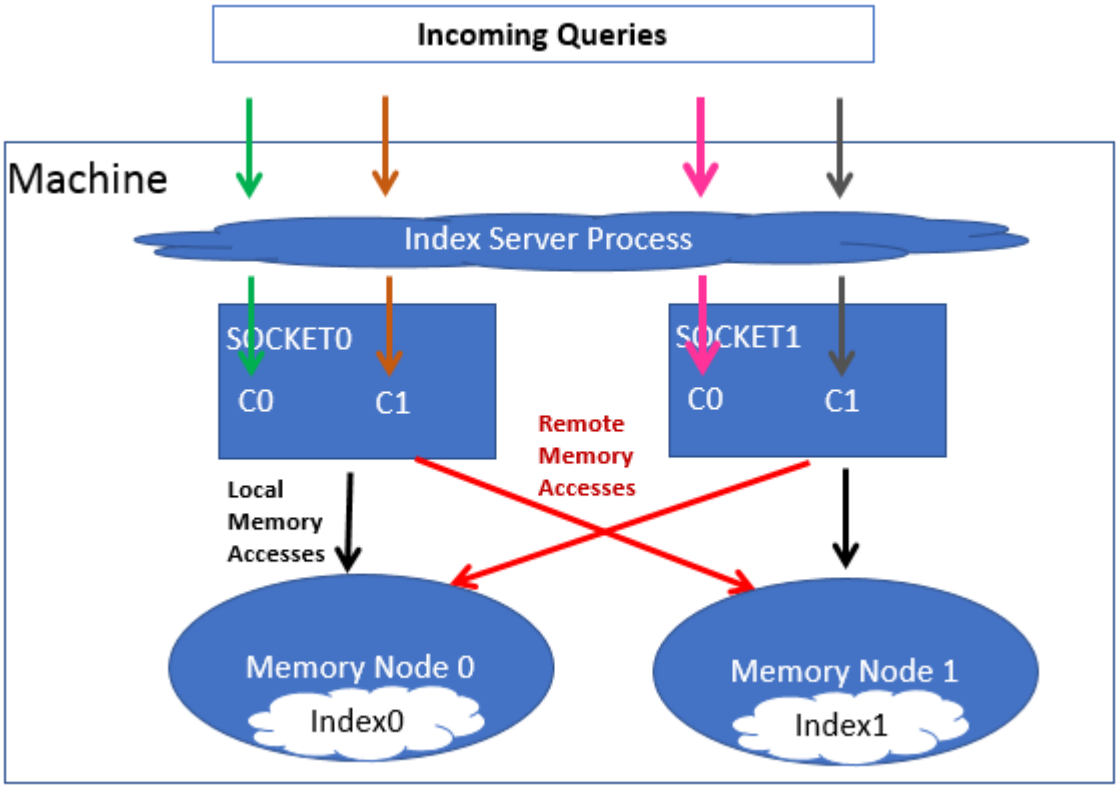
which means less parallelism), but with good load-balancing among partitions posting lists the pure search-time is reduced by up to 2X, which can be very beneficial at low-utilization (scenarios where few queries per second arrive at the server).

We evaluate each configuration scenario with single-socket and dual-socket execution. Because our CPU is dual socket for the single socket experiments, we had to take a different approach. In single socket runs we restrict the index servers to use cores from only one socket and its local memory node. For dual-socket runs the index servers can use cores in both sockets. Our server is dual-socket and is NUMA, i.e. each socket has its own local-memory node. Accessing local memory node is faster. The other memory node (remote node) that belongs to the other socket has longer access latency.

4.2 Optimizing Server Configuration

Figure 4.3 shows the mapping of the index server process and memory organization for the non-partitioned (NoSMTnoPart, SMT) and partitioned (Part, SMTpart) configurations for the dual-socket setup. For illustration purposes the CPUs have only two execution contexts C0 and C1 (can be either SMT contexts or physical cores). For the non-partitioned configuration, a single index server process is launched. That process controls all available cores. Each incoming query is served by a search thread that is scheduled to run on any of the available cores (any socket). All the index data is distributed among both memory nodes. Since the query execution can be scheduled to any core (any socket) and the query related index data can reside on any memory node, non-partitioned configurations may incur costly remote memory accesses (RMA). Regarding the index-partitioned configurations, we deploy them by splitting the index-dataset in two partitions and launching two index server processes. Each index server is working on a different index partition. We restrict each index server process to use only the resources of one socket and we mount the dataset of each process to the local memory of its socket. We find that the partitioned configurations can benefit considerably from assigning each index-server and its index-partition to a dedicated socket and its local memory node. Such approach removes the performance costly

No partition Configurations



Partition Configurations

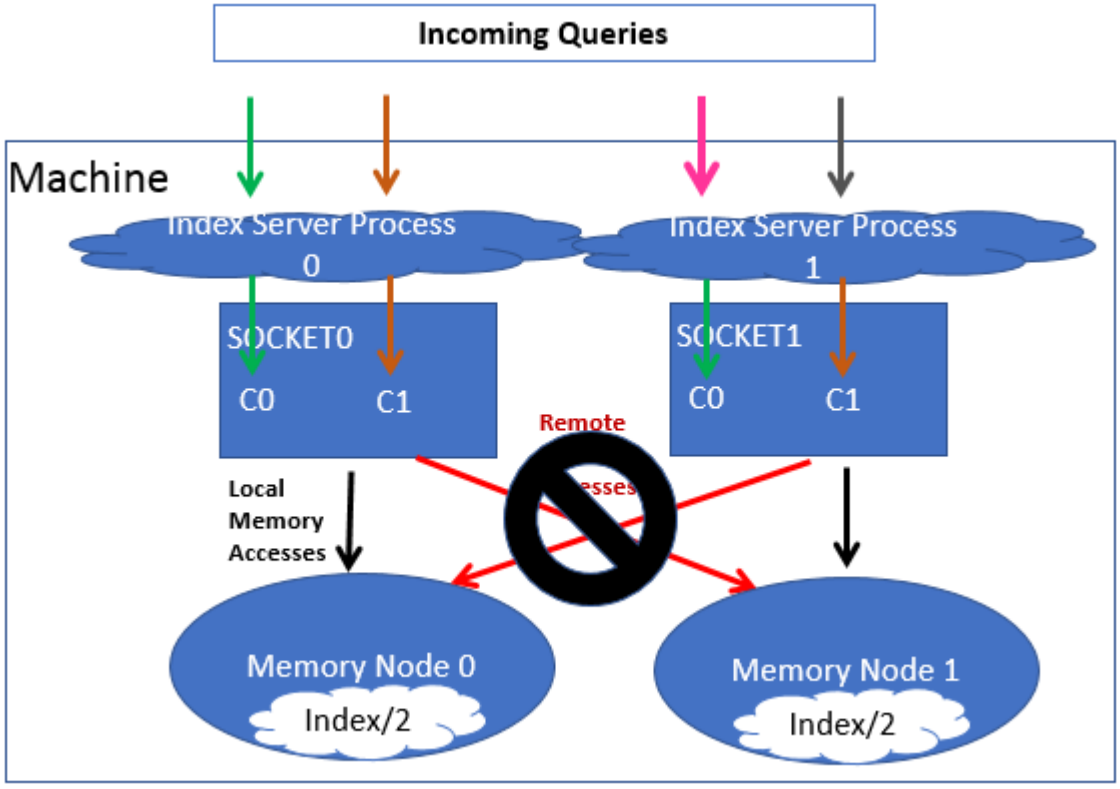


Figure 4.3. Illustration of process and memory organization for dual-socket and non-partitioned and index-partitioned configurations.

remote memory accesses and improves performance (this is clearly demonstrated in the experimental results). The reduction of remote memory accesses in the partitioned configuration is illustrated in Figure 4.3.

Besides the effect of RMA accesses we also investigate how the C-states and thread assignment impact latencies. Regarding the latter, particularly for the SMTpart configuration, we find that it is beneficial to ensure that SMT contexts mapped on the same physical core will execute threads that belong to the same index server. This can be conducive to improve cache locality and result in faster query latencies.

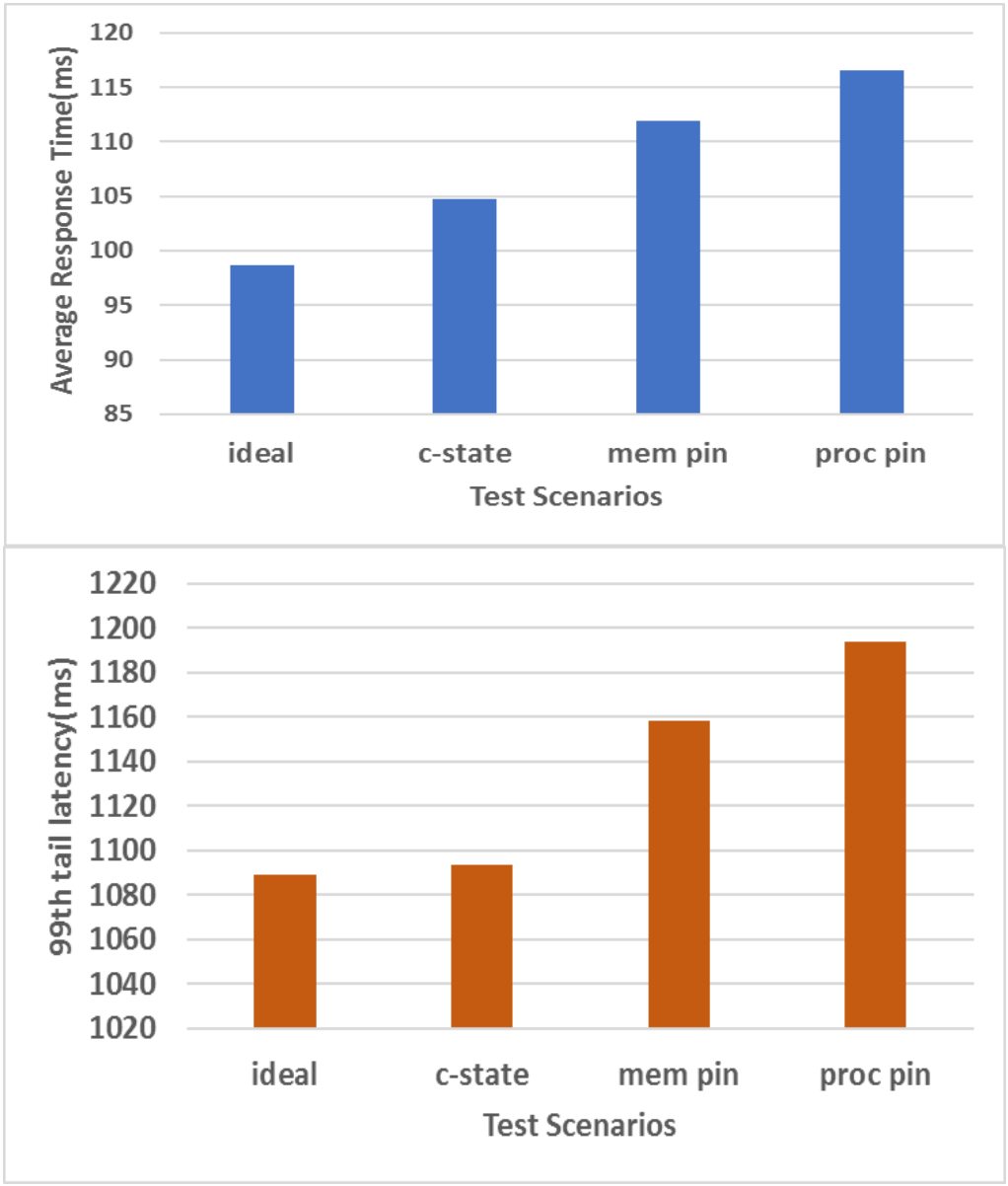


Figure 4.4 Optimizing Server Configuration for dual-socket SMTpart

Figure 4.4 illustrates on a dual-socket SMTpart configuration the latency overhead of C-states, RMA and not ideal thread pinning. The figure shows the average latency and 99th percentile of four situations: a) the ideal run where we disable c- states, perform memory-pinning to remove RMA and perform ideal thread pinning, b) same as ideal but with c-state enabled, c) same as ideal but without memory pinning, and d) same as ideal but without thread pinning. The figure clearly shows that the ideal configuration outperforms the rest and confirms prior work that characterized various sources of system and hardware latency variability (e.g. power management schemes etc) [21]. Consequently, for the rest of the paper the SMTpart dual-socket runs are performed with the ideal configuration. Also, for all other experiments we keep c-state disabled.

Chapter 5

Experimental Details

5.1 Experimental Details

For the experimental analysis we use dual-socket NUMA blade servers based on Intel Xeon E5-2665@2.4 GHz CPU. Table 5.1 provides the details of the blade server hardware. The experiments are conducted with 3 blade servers: one client, one frontend server and one index server. The machines are connected through InfiniBand. Our blade server has two Sockets, each Socket has 8 physical cores, thereby, the total number of physical cores is sixteen. Each physical core supports two-way SMT. Therefore, the total number of available SMT contexts is 32. The server has in total 64GB of DRAM with each memory node having 32GB.

For the index dataset, we are using a 28GB index crawled from various internet pages using the Nutch crawler [15]. The dataset is broken in 128 chunks of approximately 220MBs each. This facilitates index-partitioning since these chunks can be combined to form larger index partitions. For instance, the partitioned configurations for the dual-socket setup uses two index server processes on the index server machine. We use the first 64 chunks to form the first partition and the rest 64 chunks to form the second partition. Essentially, each index server works on 14GB of data (64 chunks of ~220MB). For single-socket runs we use 20GB of index data and for the partitioned configuration we use two partitions of 10GB each. The reason we use 20GB index instead of 28GB for the single-socket runs, is to ensure that the index, along with OS and process data fits comfortably in one memory node of 32GB.

For all single-socket runs, we use the numactl [13] command to restrict the index-server thread and memory affinity to one socket and its local memory node. Additionally, for the single-socket partitioned configurations we restrict each index server to utilize only

4 SMT contexts (4 physical cores). Whilst for dual-socket partitioned experiments we run one index-server on one CPU socket and the other index-server on the other CPU socket (as discussed previously). Thereby, for the dual-socket SMTpart configuration each index server utilizes 8 SMT contexts (8 physical cores).

For all experiments, we also mount the index dataset in the appropriate memory nodes using the tmpfs [11] command. Explicitly, preloading index into memory instead of relying on the OS filesystem cache, improves performance and removes variability between runs. For single-socket experiments the index data is mapped to the memory node that is local to the active socket. For the dual-socket SMT and NoSMTnoPart configurations, the index is mapped to both memory nodes with OS controlling how much data will be mapped to each node (we empirically found that this provides the best performance). For dual-socket experiments with partitioning, since we use one index-server per socket, we load the index-data of each index-server to its local memory node (ideal configuration Figure 4.3).

For query stream we use 100K real life queries taken from the AOL query log [25]. Queries are sent with interarrival time meaning that on average a query is being send every x ms based on a Poisson distribution. We use different inter-arrival times (or inter-arrival rates) to explore the SMT and IP implications for various levels of CPU utilization. We run each experiment 10 times and we report the average of 10 runs for each metric (CPU utilization, average latency, 99th percentile etc.).

For performance counters measurements we use the perf Linux utility [14], and for measuring NUMA access the NUMATOP command [12]. For measuring CPU utilization, we rely on the mpstat command.

Number of Sockets	2
CPU	Intel Xeon E5-2665 @2.4GHz
Number of physical cores per Socket	8
Number of logical cores(SMT contexts) per CPU	2
DRAM	8x8 GB DDRE 1600MHz
NUMA Memory nodes	2 each has 32GB
Ethernet speed	InfiniBand
OS	Ubuntu 16, Kernel 4.4.0-63

Table 5.1 Server Configuration

Chapter 6

Experimental Evaluation

6.1 Dropped Queries and CPU utilization

In our first part of our analysis, we determine at which query arrival-rate each configuration suffers dropped queries. We evaluate the scenarios we have mentioned above (NoSMTnoPart, Part, SMT and SMTpart) but for both single and dual socket runs. Figure 6.1 shows the number of dropped queries for all single-socket configurations and Figure 6.2 for all dual-socket configurations as a function of traffic rates. At traffic rates up to 125 queries/sec we do not observe any dropped queries. For single-socket runs, the minimum rate with dropped queries is at 150 queries/sec for the NoSMTnoPart and Part configuration. For dual-socket runs, again the NoSMTnoPart configuration is the first to suffer dropped queries at 200 queries/sec. The SMTpart scenario doesn't suffer dropped queries for the dual socket runs that's why there is no indication of it at the graph. This is interpreted as an indication that disabling SMT is counterproductive for the search application we evaluate.

The goal of this analysis is to decide which arrival-rates to use for the latency analysis comparison. We pick traffic rates where none of the configurations suffers dropped queries. Consequently, for single-socket configurations we will compare latencies for traffic rates up to 125 queries/sec and for dual-socket traffic rates up to 175 queries/sec.

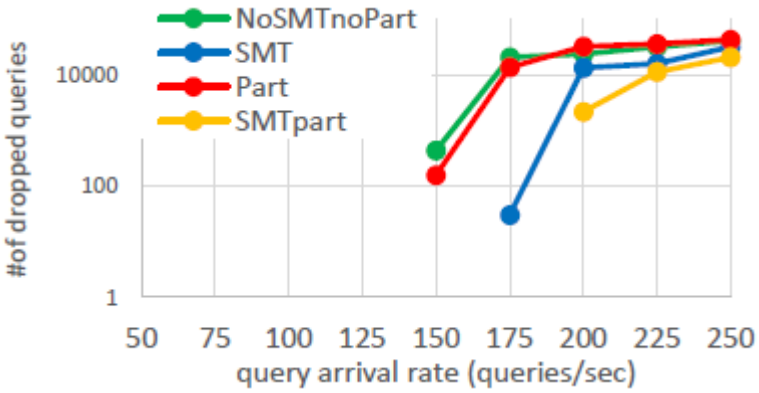


Figure 6.1 Single Socket dropped queries characterization.

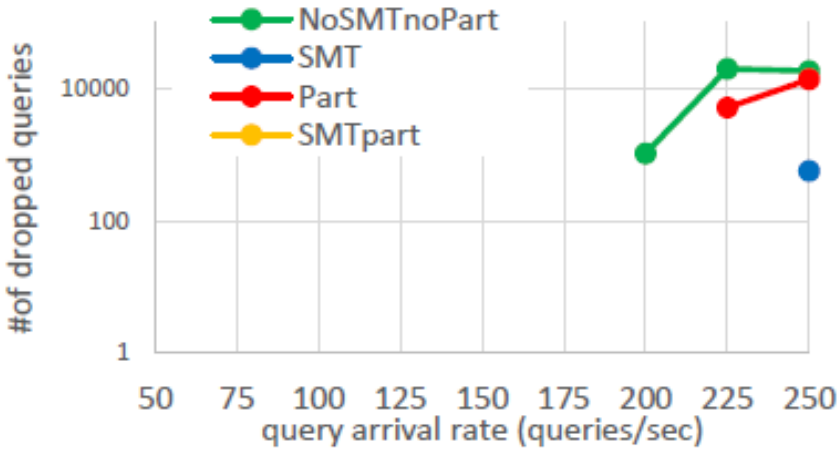


Figure 6.2 Dual Socket dropped queries characterization

Figure 6.3 shows the physical core CPU utilization for the baseline single-socket NoSMTnoPart configuration. We can observe that the selected arrival rates stress a wide spectrum of CPU utilization from low to high utilization level. Particularly, we observe a 36% utilization at 50 queries/sec and 80% utilization at 125 queries/sec. The 80% utilization means that on average 6-7 physical cores from the total 8 cores that our CPU socket has are active (We use the 80% of 8 cores so : 80% * 8 = 6.4). For dual-socket runs Figure 6.4 shows that at 50 queries/sec we have 25% CPU utilization (4 cores active out of total 16) and at 175 queries/sec 75% (75% * 16 = 12 12 cores active).

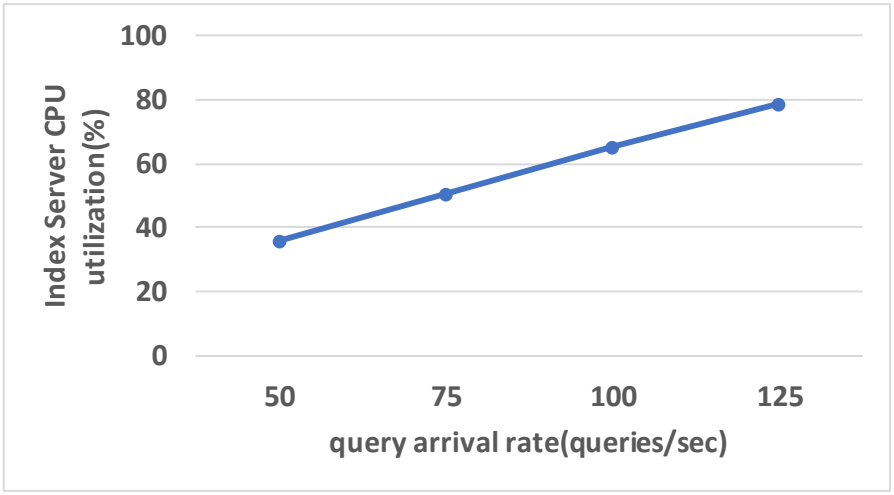


Figure 6.3 Single Socket CPU utilization for baseline NoSMTnoPart Scenario.

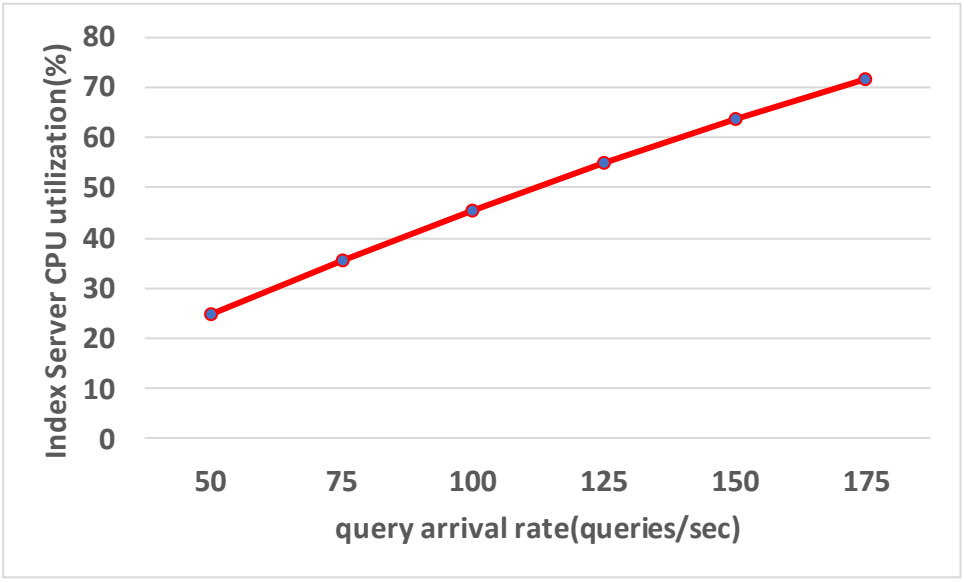


Figure 6.4 Dual Socket CPU utilization for baseline NoSMTnoPart

6.2 Single – Socket Latency Results

Figure 6.5 compares the average latency and Figure 6.6 the 99th percentile latency for each configuration. The first observation is that Part configuration offers worse 99th than the baseline NoSMTnoPart configuration despite offering a slightly better average at all arrival rates. We explain this behavior with latency variability analysis. It is known that more parallelism suffers from higher probability a single server to slow down the whole query execution [9][8]. The STDEV results in Figure 6.8 support this.

We observe that both SMT part and Part have higher STDEV latency than SMT and NoSMTnoPart respectively. The high STDEV of Part configuration indicates that it suffers from latency variability and while this doesn't affect negatively the average it affects the 99th percentile.

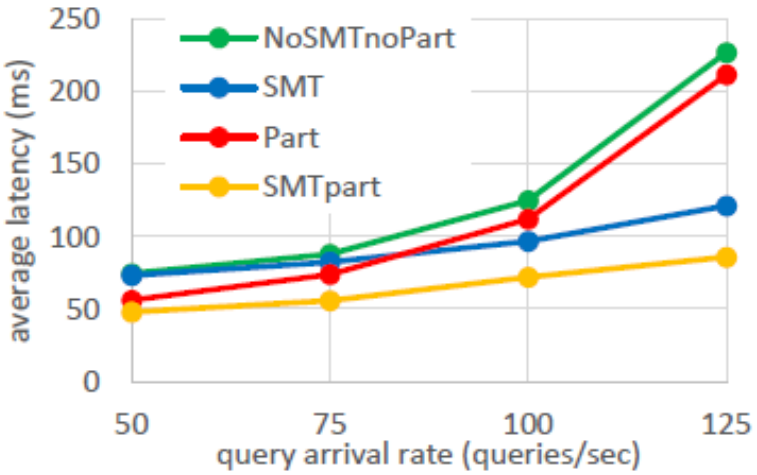


Figure 6.5 Single-Socket average latency

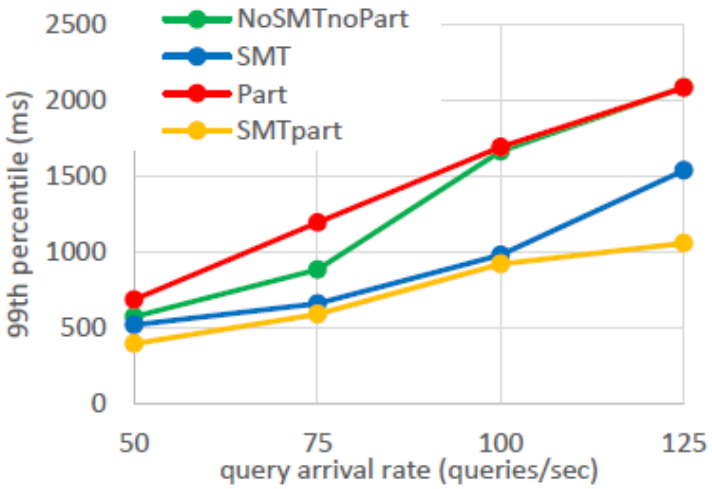


Figure 6.6 Single-socket 99th percentile latency

Regarding, SMT and SMTpart configuration both configurations achieve lower latencies than the baseline. Particularly, the SMT behave similar to NoSMTnoPart in terms of average latency for traffic rates up to 75 queries/sec. This is because the SMT-aware OS scheduling does not map two threads to the same physical core at low utilization (where the number of queries to be served is less or equal to the number of

physical cores). Still, in terms of 99th percentile latency the SMT provides lower latencies than NoSMTnoPart. This is attributed to cases where queries incur queue latency even at low utilization. For these situations, the SMT configuration clears up the queued queries faster providing lower latencies. This is supported by Figure 6.7 that shows the queueing times for all traffic rates. It is clearly shown that the SMT configuration provides the lowest queueing times at low traffic rates. At high traffic rate (100 queries/sec and more) SMT provides significantly faster latencies than NoSMTnoPart both for average and 99th percentile. SMT is beneficial for the average at high traffic rates because it provides 2x more throughput (execution contexts) while inducing 1.5x single thread degradation (calculated by measuring the single-thread IPC when two search threads run on the same physical core).

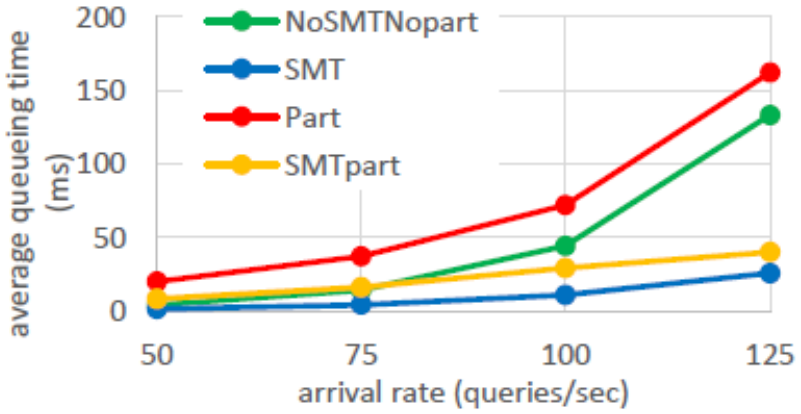


Figure 6.7 Single Socket average queueing time.

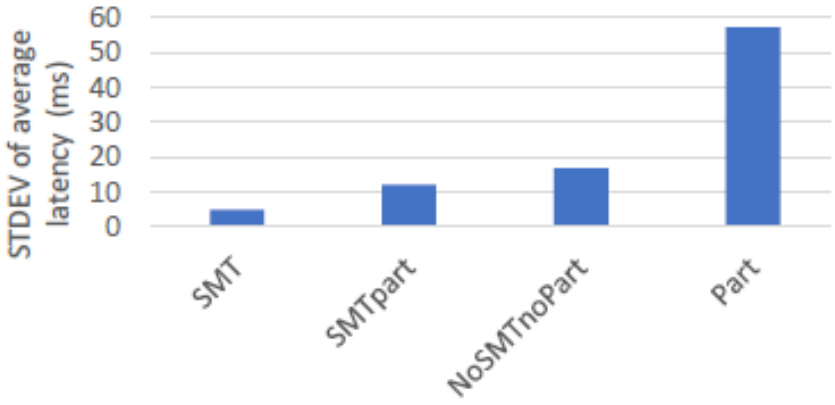


Figure 6.8 Single-socket STDEV results at 125 queries/sec.

Regarding SMTpart, this configuration provides the lowest average and 99th percentile latency at all arrival rates, which demonstrates the great synergy between SMT and IP. This improvement is credited to the good load-balancing among partitions. As shown in

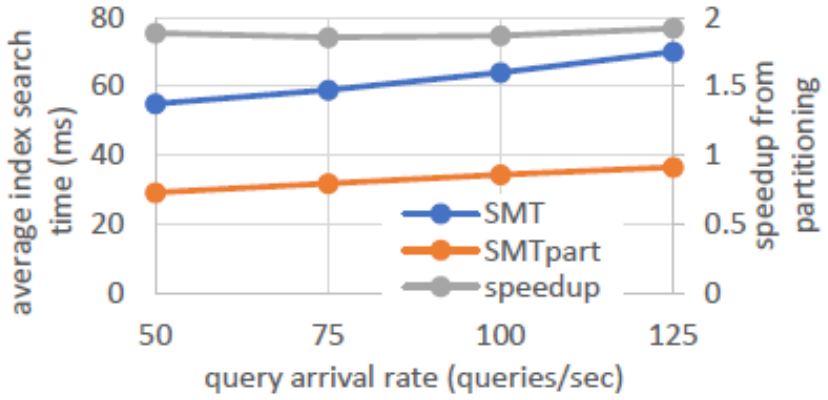


Figure 6.9 Single-socket index search time and speedup from partitioning.

Figure 6.9, SMTpart compared to SMT provides nearly 2X faster average search time (pure index-search time without queueing time included).

6.3 Dual Socket Latency Results

Figure 6.10 compares the average latency and Figure 6.11 the 99th percentile for dual-socket NoSMTnoPart, Part, SMT and SMTpart configurations. Similar to single-socket runs, the SMT outperforms NoSMTnoPart at higher traffic rates while providing equal latencies at lower traffic rates. Again, the clear winner for the dual-socket experiments is SMTpart both for average and 99th percentile but Part configuration also provides better latencies across all utilization levels compared to NoSMTnoPart.

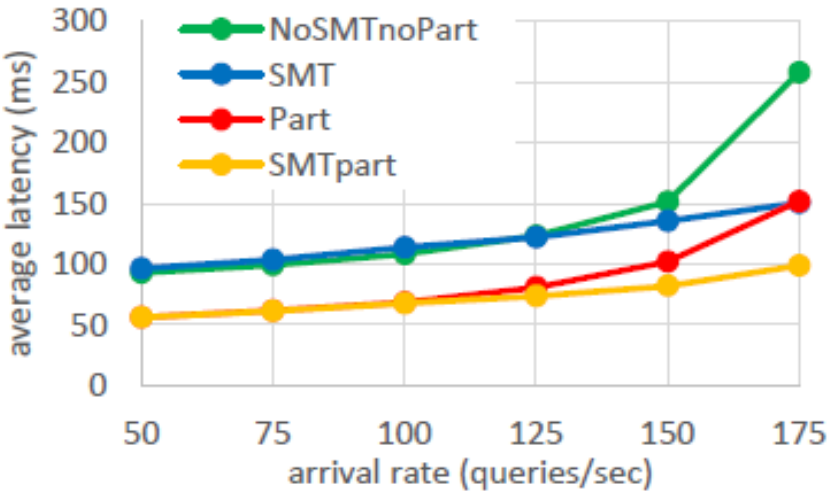


Figure 6.10 Dual-socket average latency.

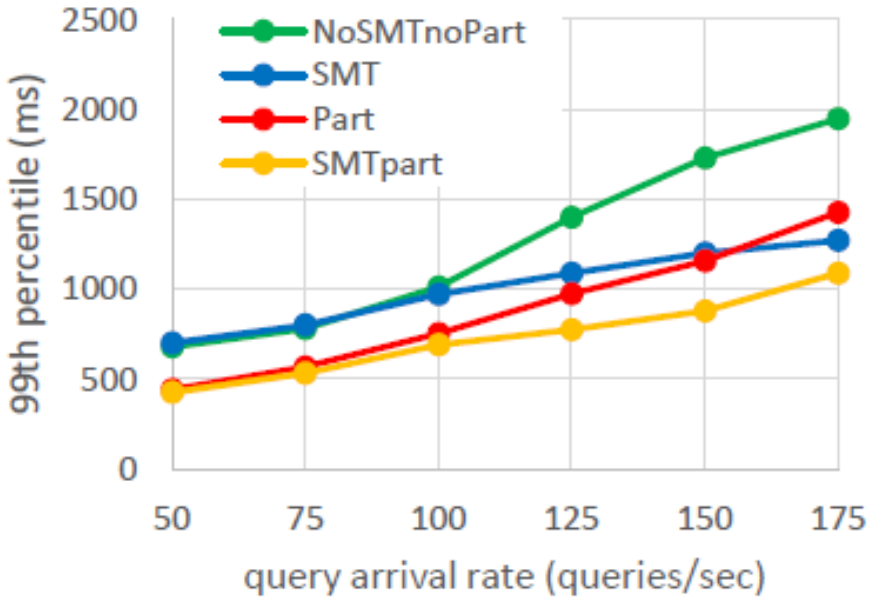


Figure 6.11 Dual Socket 99th percentile.

This behavior of Part stems from a reduction in RMA. In particular, as explained previously we dedicate one socket and its local memory node to each index server and its corresponding index partition. This eliminates the remote memory accesses (RMA) and results in the improved speedup. Figure 6.12 compares the RMA for SMT and SMTpart configurations. It clearly shows that SMT incurs RMA accesses while

SMTpart effectively does not. Because of this, SMTpart has 10% higher single-thread IPC than SMT as shown in Figure 6.13. Moreover, the same figure shows other micro-architectural metrics. None of them appears substantial to justify the IPC difference. Hence, we attribute the performance difference between the two configurations to the difference in the RMA behavior.

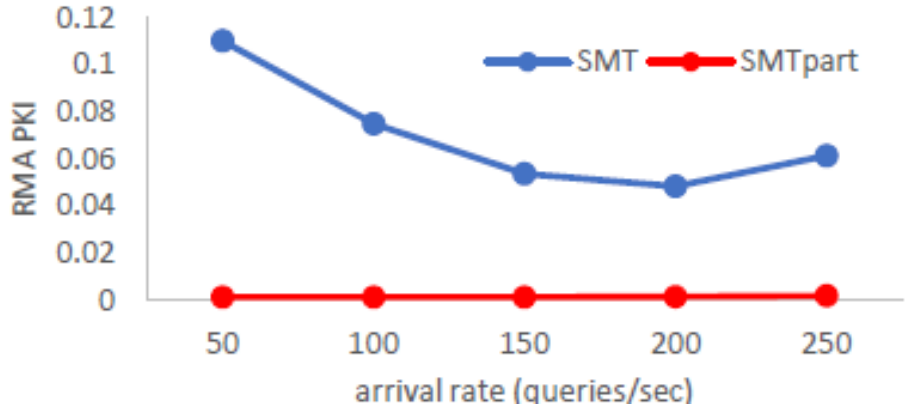


Figure 6.12 RMA accesses per 1K instructions for dual-socket SMT and SMTpart.

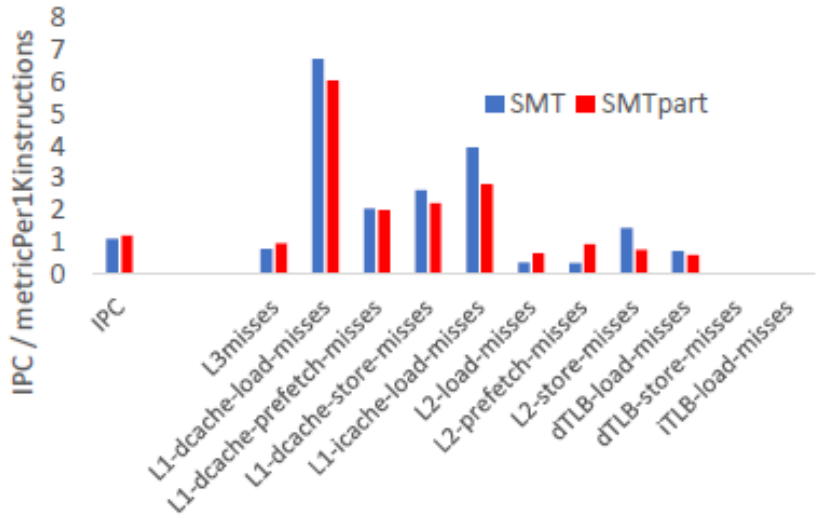


Figure 6.13 Performance counters for dual-socket SMT and SMTpart.

Chapter 7

Related Work

7.1 Related Work

This chapter presents the related work and describes in what ways my thesis differs from it. The impact of SMT on the performance of HPC applications is examined in [29][4]. Also, the following works [22][35][28][26] examined the SMT impact on the performance of operating system, database and network workloads. Authors in [5] examined the impact of SMT on cloud workloads but only from the perspective of throughput not from the perspective of QoS. In [34][33] authors point out that SMT has detrimental effect to QoS of online cloud-workloads. They propose solutions for increasing server utilization with offline jobs to enable the use of SMT contexts without increasing the latencies of online jobs. Our work does not examine SMT colocation of offline and online applications but shows that using all SMT contexts for online search can improve latency. Authors in [27] use SMT to emulate the behavior of a heterogeneous CPU and demonstrate the benefits from heterogeneous aware task-scheduler. The analysis in [27] does not highlight the SMT latency improvement potential. In [17][31][7] the authors examined how web search can benefit from IP and parallel-execution across physical CPU cores. We extend these works by evaluating IP across SMT contexts. The importance of investigating the ramifications of SMT is highlighted by the fact that some previous work on online applications avoids the use of SMT to ease the analysis [6][7].

Work	SMT	Online Workload	Latency Analysis	Index-Partitioning
[29][4][22][35] [28] [26]	X			
[5][27]	X	X		
[34][33]	X		X	
[17][31][7]		X	X	X
This work	X	X	X	X

Table 7.1 Comparison with related work.

As far as we know, this is the only work to examine the ramifications of SMT and IP on the average and tail latency for an online workload. Table 7.1 highlights the differences of our work with prior work.

Chapter 8

Conclusion

8.1 Conclusion

The thesis evaluates the SMT and IP ramifications on the response latency of an online search application. Contrary to popular belief, we find that SMT can be latency-friendly. Generally, if the single-thread degradation due to SMT contention is less than the increase in logical cores SMT provides, SMT can help reduce both average and tail latency. Our study also shows that even at low query traffic rates, when not all SMT contexts are utilized, SMT is not detrimental because of SMT-aware OS scheduling. Furthermore, we observe that given a workload amenable to partitioning, such as the online search we evaluate, SMT combined with IP across SMT contexts can provide the lowest latencies across all inter-arrival rates we have considered. We also demonstrate the generality of our conclusions with the help of a queuing model.

This work points to several directions for future work. For one, there is a need for similar studies, as the one performed in this work, but for other applications. This is useful to investigate the generality of the observations we make. Another direction of work is to study the potential benefits with increasing number of SMT contexts and identify potential microarchitectural features that can moderate the SMT performance degradation with multiple SMT contexts.

A big part of the current thesis is based on the journal “Comprehensive Characterization of an Open Source Document Search Engine” (Xi lab unpublished data) and “Characterization and Analysis of a websearch benchmark”[6].

References

[1] Barroso, Luiz André, Jeffrey Dean, and Urs Holzle. "Web search for a planet: The Google cluster architecture." IEEE micro 23.2 (2003): 22-28.

[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): 1-154.

[3] Christopher, D. Manning, Raghavan Prabhakar, and Schutza Hinrich. "Introduction to information retrieval." An Introduction To Information Retrieval 151.177 (2008): 5.

[4] Esmaeilzadeh, Hadi, et al. "Looking back on the language and hardware revolutions: measured power, performance, and scaling." ACM SIGARCH Computer Architecture News. Vol. 39. No. 1. ACM, 2011.

[5] Ferdman, Michael, et al. "Clearing the clouds: a study of emerging scale-out workloads on modern hardware." ACM SIGPLAN Notices. Vol. 47. No. 4. ACM, 2012.

[6] Hadjilambrou, Zacharias, Marios Kleanthous, and Yanos Sazeides. "Characterization and analysis of a web search benchmark." Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on. IEEE, 2015.

[7] Haque, M. E., He, Y., Elnikety, S., Bianchini, R., & McKinley, K. S. (2015, March). Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In ACM SIGPLAN Notices (Vol. 50, No. 4, pp. 161-175). ACM.

[8] Haque, M. E., He, Y., Elnikety, S., Nguyen, T. D., Bianchini, R., & McKinley, K. S. (2017, October). Exploiting heterogeneity for tail latency and energy efficiency. In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (pp. 625-638). ACM.

[9] Hölzle, U. (2010). Brawny cores still beat wimpy cores, most of the time.

[10] http://download.microsoft.com/download/5/7/7/577a5684-8a83-43ae-9272-ff260a9c20e2/hyper-thread_windows.doc

[11] http://man7.org/linux/man-pages/man5/tmpfs.5.html

[12] https://01.org/numatop

[13] https://linux.die.net/man/8/numactl

[14] https://perf.wiki.kernel.org/index.php/Tutorial

[15] https://wiki.apache.org/nutch/NutchTutorial

[16] J. Doweck et al., "Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake," in IEEE Micro, vol. 37, no. 2, pp. 52-62, Mar.-Apr. 2017.

[17] Jeon, Myeongjae, et al. "Adaptive parallelism for web search." Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013.

[18] Jeon, Myeongjae, et al. "Predictive parallelization: Taming tail latencies in web search." Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval. ACM, 2014.

[19] Kanev, Svilen, et al. "Profiling a warehouse-scale computer." 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). IEEE, 2015.

[20] L. Kleinrock, "Queueing Systems", Wiley-Interscience, 1975

[21] Li, Jialin, Naveen Kr Sharma, Dan RK Ports, and Steven D.Gribble. "Tales of the tail: Hardware, os, and application-level sources of tail latency." In Proceedings of the ACM Symposium on Cloud Computing, pp. 1-14. ACM, 2014

[22] Lo, Jack L., et al. "An analysis of database workload performance on simultaneous multithreaded processors." ACM SIGARCH Computer Architecture News. Vol. 26. No. 3. IEEE Computer Society, 1998.

[23] Marr, Deborah T., et al. "Hyper-Threading Technology Architecture and Microarchitecture." Intel Technology Journal 6.1 (2002).

[24] Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and James Lin. 2012. Earlybird: Real-time search at twitter. In Data Engineering (ICDE), 2012 IEEE 28th International Conference on. IEEE, 1360–1369.

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

[26] Redstone, Joshua A., Susan J. Eggers, and Henry M. Levy. "An analysis of operating system behavior on a simultaneous multithreaded architecture."ACM SIGPLAN Notices 35.11 (2000): 245-256.

[27] Ren, Shaolei, et al. "Exploiting processor heterogeneity in interactive services." Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13). 2013.

[28] Ruan, Yaoping, et al. "Evaluating the impact of simultaneous multithreading on network servers using real hardware." ACM SIGMETRICS Performance Evaluation Review. Vol. 33. No. 1. ACM, 2005.

[29] Saini, Subhash, et al. "The impact of hyper-threading on processor resource utilization in production applications." High Performance Computing (HiPC), 2011 18th International Conference on. IEEE, 2011.

[30] T. Singh et al., "Zen: An Energy-Efficient High-Performance x86 Core," in IEEE Journal of Solid-State Circuits, vol. 53, no. 1, pp. 102-114, Jan. 2018.

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

[32] Tullsen, Dean M., Susan J. Eggers, and Henry M. Levy. "Simultaneous multithreading: Maximizing on-chip parallelism." ACM SIGARCH Computer Architecture News. Vol. 23. No. 2. ACM, 1995

[33] Yang, Xi, et al. "Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading." 2016 USENIX Annual Technical Conference (USENIX ATC 16). 2016.

[34] Zhang, Yunqi, et al. "Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers." Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2014.

[35] Zhou, Jingren, et al. "Improving database performance on simultaneous multithreading processors." Proceedings of the 31st international conference on Very large data bases. VLDB Endowment, 2005.