How to make SMT Tail Latency Friendly

ABSTRACT

In this work we investigate how to run a latency sensitive workload on all simultaneous multithreading (SMT) contexts of a core while improving both latency and throughput. Our study focuses on the SMT ramifications on a web search application. One way to run web search with SMT is to execute one independent web search query per SMT context. This approach improves web search throughput, because more queries can be served in parallel, but unfortunately it is detrimental for QoS. The culprit is single thread performance degradation due to resource contention across SMT contexts. As a possible remedy for this degradation, we examine the benefits of partitioning the dataset and serving each web search query with multiple threads across SMT contexts. Our results show this approach to improve the tail latency of web search as compared to a baseline that does not use SMT. Since each query is served faster, throughput is improved as well. These findings indicate that web search is amenable to parallelization and dataset partitioning. Our results on a real platform show that the synergy of SMT with workload partitioning and parallel execution improves web search throughput and tail latencies by 1.3X.

CCS Concepts

• Computer systems organization -> Architectures

Keywords

SMT, hyper threading, web search, partitioning, online, latency sensitive, datacenters, simultaneous multi-threading, tail, latency

1. INTRODUCTION

Simultaneous multithreading [30] is a performance feature found often in server processors [1,2,3,27]. Datacenter and HPC operators can choose whether to actually employ SMT or disable it. Broadly speaking, SMT increases CPU utilization and computational throughput by allowing multiple threads to share a physical core's resources (such as registers, execution units and caches). On the flip side, SMT can have harmful effect on single thread performance because of the contention for shared resources. Particularly in the case of latency sensitive workloads (online workloads), SMT can have detrimental effects on response latencies and QoS (Quality of Service) requirements [2,3] such as the tail latency.

A clear testament of the SMT trade-off between throughput and response latency is presented in Figure 1. The figure analyzes the performance implications of SMT on a web search service by comparing a run which utilizes 8 physical cores versus a run that utilizes 16 SMT contexts on the same 8 physical cores. More details about this experiment and the experimental setup are given in Sections 3 and 4. Throughput above 1 indicates that SMT is beneficial, whereas average and 99th percentile above 1 show that SMT is harmful to response latencies. As it is expected throughput is increased because more threads are available to serve independent query requests. But the average and 99th response latency increase as well. Therefore, using SMT improves throughput but single thread performance is degraded. Therefore, the safest choice by a datacenter architect, that cares about response latency of this service, will be to not use SMT.

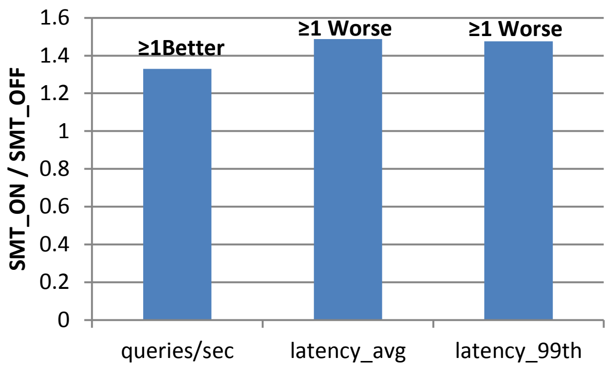

Figure 1. Relative comparison of throughput and latencies with and without SMT

Indeed, related work as well as anecdotal sources suggests that a often datacenter and HPC operators prefer to not utilize SMT [1,2,3]. Not using SMT means inefficient use of available resources that translates to need for additional servers, higher power and cost. Naturally, previous work attempted to identify safe ways to collocate batch and online workloads on SMT contexts to improve server utilization without violating QoS requirements [2,3]. But, as far as we know, no previous published work proposed means to leverage SMT for both throughput and latency reduction.

This works builds on the observations of an earlier study [9] that shows, for a web search benchmark, dataset partitioning and parallel execution - on lower performing cores - can match both the average and tail latency of a more powerful core without dataset partitioning.

In particular, in this work we examine how the above observations can enable the efficient use of SMT for servers running online services. We show that SMT can be useful for improving both throughput and latency of interactive services. To achieve this, we combine dataset partitioning and parallel execution of each web query across SMT contexts. This idea is appealing for two reasons: a) It enables latency reduction of each web request, b) throughput wise, it has the capacity to serve the same number of independent web requests as a server that is deployed with SMT disabled; furthermore, since each request is served faster, throughput is improved as well. The proposal is evaluated using the Cloudsuite's web search benchmark [4] which is a representative online service workload [4,5,6].

Our experimental results on a real platform show that the combination of (i) dataset partitioning and (ii) parallel execution of web search query across SMT contexts, offers significant performance gains. Particularly, the results show approximately 1.3X improvement in throughput and tail latencies over a baseline configuration that does not use SMT.

The rest of the paper is organized as follows. Section 2 provides background on SMT and Web Search. Section 3 examines the impact of SMT on Web Search. Section 4 presents the experimental setup. Section 5 presents the experimental results.

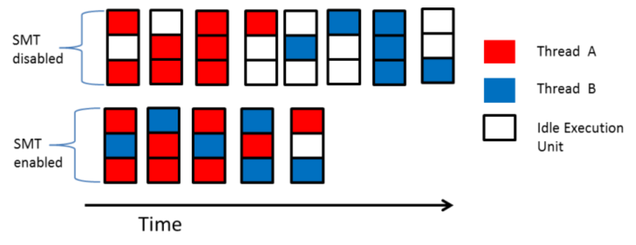

Figure 2. Illustration of pipeline behavior without and with SMT.

Section 6 presents the related work and explains how this paper differs from it. The paper concludes in Section 7.

2. Background

This section provides some background on SMT and web search.

2.1 SMT

Figure 2 illustrates how two threads A and B run on a core without SMT support, or SMT disabled, and when using SMT. Without SMT the threads run sequentially and the pipeline is underutilized when hazards prevent instructions of one thread from executing (empty squares in Figure 1 signify idle execution). When SMT is enabled we can observe that pipeline utilization is increased because the two threads can execute simultaneously and use different processor resources. In general, with SMT the instructions per cycle (IPC) executed by the core is increased but single-thread performance, IPC per thread, gets worse due to resource contention.

Most modern CPUs offer two-way SMT [11], this means that up to two threads can simultaneously execute and share a core's resources. However, 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 [1,12,13]. Regarding the SMT effects on online services, the throughput may improve, but latencies will worsen due to resource contention [2,3].

2.2 Web Search

Web search is one of the most widely used online services [14]. Web search refers to services that respond to user queries with relevant web documents. For example, a query with the following keywords "how smt works" should return web pages that explain SMT functionality. For our evaluation we use the Nutch web search benchmark from cloudsuite [4]. Nutch benchmark is based on Lucene [15] a widely deployed open source search engine.

The Nutch Web Search benchmark consists of four main components that are illustrated in Figure 3 that 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. The index partitioning enables parallel search across index servers. Typically index search is performed in parallel across multiple server machines [5,7]. Document servers hold the actual documents and they are used for fetching the summaries of the search results. A query is executed with the following sequence: 1) The client sends a query to the frontend server. 2) The frontend server receives the query and asks from each index server to return the most relevant to the query documents. 3) The

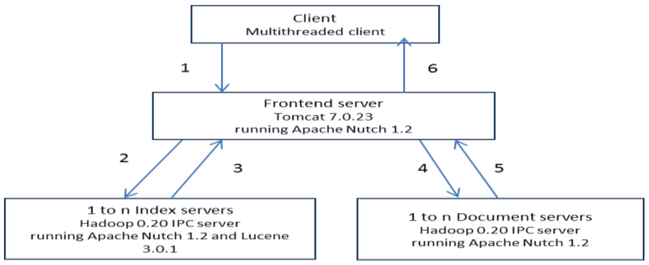

Figure 3. The basic components of the Nutch Benchmark. The arrows show the information flow and interactions between components during a query execution. The numbers show the order of the execution flow.

index 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 server collects the results and sorts the documents according to their relevance score. 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 titles 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 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.

To summarize, a query end to end latency can be viewed as the sum of 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 3) the detail requests which are also performed on the index server, and 4) the summary requests which are performed on the document server. Figure 4 shows on average how much time is spent at each query phase 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 the figure is that the most time is spent on index search. This conclusion is in line with what previous work has reported, that index search is the most important part of a web search engine [5,8,16]. Therefore unless noted otherwise, this paper focus is on optimizing index search.

2.2.1 Parallelization benefits for Index Search

Figure 4 shows that the index search time increases linearly with dataset size. Therefore, partitioning the dataset among multiple index servers and performing parallel search can potentially help decrease the search time. To confirm that indeed index search time scales well with parallelization we split the index to 2 and 4 parts and run parallel index search. Figure 5 shows the effect of partitioning on average and 99th percentile latencies as perceived by the frontend server. In other words, the query latency is measured from the time the frontend server issues search requests to all index servers until the time the search results arrive back to frontend. Also, queries are executed sequentially in isolation. The figure shows that the speedup for average index search time and 99th percentile scales nearly linearly for 2 index servers. Also, the speedup in 99th percentile is better than the speedup of the average

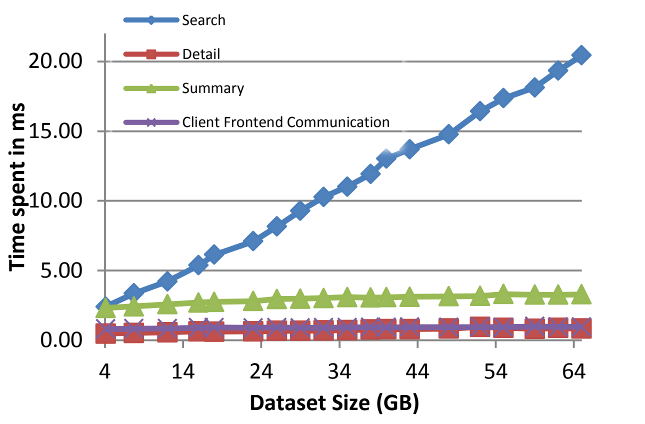

Figure 4. Time spent at each phase of the benchmark in relation to dataset size. The index search is the most demanding phase and scales with dataset size.

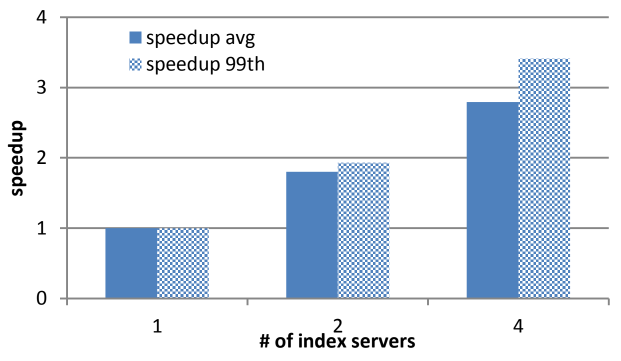

Figure 5. Speedup in average index search time and 99th percentile vs the number of index servers

response time both for 2 and 4 index servers. Queries that dictate the 99th percentile usually have much higher processing times than the average query [3,6,9,10]. It is generally accepted that the bigger a problem is in terms of amount of work that has to be performed, the more it will benefit from parallelism [10]. Therefore, it is reasonable to see higher speedup for 99th percentile. To conclude, the observed speedup both for average and 99th percentile encourages index dataset partitioning and parallel index search across SMT contexts.

3. SMT impact on web search

This Section presents different ways of using SMT to run an interactive service and discusses the SMT implications on an individual server's performance as well as on the overall datacenter architecture running an interactive service. The following three possibilities are examined: a) Use one SMT context per physical core (same effect as disabling SMT), b) Enable SMT and execute one independent query per SMT context, c) Enable SMT, partition the workload across SMT contexts and execute each query in parallel.

3.1 SMT implications on server performance

Figure 6 presents the three options for how to use SMT resources to serve web search queries. The first option is to not use SMT (NO-SMT), the single thread performance and response latency is best but the SMT capability remains unused. The second option is to use SMT ("SMT" configuration) to increase the logic core count by two; therefore, each core can serve up to two independent queries at the same time and increase throughput. Unfortunately, the cores are slower than a core with "No-SMT" because of the single thread performance degradation due to contention (see Figure 1). One way to recover the performance loss is to reduce the dataset per server and employ more servers each with smaller dataset. This has ramification on the datacenter scale that is discussed subsequently.

The third configuration, the one that we propose, combines SMT with dataset partitioning (SMT+ partitioning). This configuration does not tradeoff latency for throughput as the "SMT" configuration does and improves both latency and throughput over the "NO-SMT" configuration. This is possible given (i) that dataset partitioning does not lead to imbalance, supported by Figure 5 that shows ~1.9X with two-way partitioninig, and (ii) the performance degradation due to resource contention caused by SMT scales slower than the degree of dataset partitioning (supported by Figure 1 that shows 33% degradation with two way SMT). Simply put, if we partition a dataset by X and use X SMT threads in a core, then if each SMT thread has performance >1/X as compared to NO-SMT then SMT can better both response latency and throughout over NO-SMT. Putting it in another way, the per query latency is improved because the parallelism benefits from partitioning overcome the single thread slowdown due to SMT. Naturally, the throughput of SMT+partitioning is also higher as compared to NO-SMT since each query is executed faster.

A broader implication of SMT+partitioning is that it may be applicable to real life web search deployments for immediate latency, energy and cost reduction. Of course, workload partitioning across SMT contexts might need some development or administrative effort, which varies depending on the particularities of each specific web search engine. Nonetheless, we believe that this effort is worthwhile. First of all, the proposed approach enables the use of SMT, a feature that often remains unexploited [3]. This helps web search providers to use more efficiently the purchased servers. Secondly, the latency and throughput gains observed, enable energy and cost savings. Simply put, SMT+partitioning improves server performance and, thus, for the same amount of work fewer servers can be used which can translate to energy and cost savings.

There are two ways to implement the SMT+partitioning configuration regarding query and data locality. We can either aim for query locality by enforcing SMT contexts of the same physical core to perform search on the same query but different index data; or we can strive for index data locality by enforcing SMT contexts of the same physical core to perform search on the same index data but for different queries. We will evaluate which option is better in the experimental section.

To summarize, Table I qualitatively shows how each configuration affects the latency and throughput. No-SMT is the baseline; therefore, it has a neutral effect on throughput and latency. "SMT" gives better throughput but worse latency compared to No-SMT. And SMT+partitioning has better latency and throughput greater or at least equal to "No-SMT".

3.2 SMT implications on the datacenter architecture

From a single server perspective, the "SMT+partitioning" looks like the winner configuration. It provides both throughput and latency improvement at the server level over the "No-SMT" configuration. But from a datacenter total cost of ownership (TCO) point of view it is not so trivial to draw conclusions.

In particular, assuming that the NO-SMT configuration satisfies QoS goals, then the SMT+partitioning configuration enables to increase of index dataset per server while achieving the same QoS and throughput as the No-SMT configuration. This essentially translates to TCO reduction since fewer servers can be used to achieve the same level of performance.

On the other hand, the "SMT" configuration can match the No-SMT QoS by decreasing the amount of index dataset per server and deploy more servers to accommodate the dataset. Acquisition of more servers increases the TCO but the "SMT" configuration offers 2X more throughput while achieving the same QoS as the other configurations. Therefore, in terms of throughput/$ the SMT might seem winner configuration. Still, though, it is hard to conclude if such deployment would be advantageous because the use of more servers to serve each query (as the dataset needed to be searched is spread in more servers), can have impact on the tail latencies of the whole cluster due to aggregation overheads, load imbalance and other performance variability sources [30]. The implications of scaling the number of servers on the tail latencies is an important and interesting problem but out of the scope of this paper. In future work we will try to evaluate the scaling overheads over multiple servers and evaluate how the two SMT configurations compare in terms of a holistic datacenter point of view.

4. Experimental Setup

For the experimental analysis we use dual socket blade servers based on Intel Xeon E5620@2.4 GHz CPU. Table II 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 1Gb Ethernet. Experiments are performed with CPU constantly running at the max frequency of 2.4GHz. Our blade server has eight physical cores. Each physical core supports two-way SMT. Therefore, the total number of available SMT contexts is equal to 16.

For query stream we use 100K real life queries taken from the AOL query log [26]. Queries are sent in a stress test manner meaning each client thread sends a new query as soon as it receives the response for the previous one. Unless stated otherwise, eight client threads are used in the runs because we find that this number is sufficient to utilize all the 8 physical cores of our blade server. To set up the SMT+partitioning configuration we run two index server processes on the index server machine. Each index server utilizes 8 SMT contexts. We use taskset command to set the affinity of index servers. We have two options for setting the index server affinity. One option enforces query locality and the other option enforces index data locality among SMT contexts that belong to the same physical core. For instance, index server #1 may utilize one SMT context from each physical core and index server #2 may utilize the rest SMT contexts. Essentially, this way we enforce SMT contexts of a physical core to perform search on the same query but on different index data. Basically this option enforces query locality during execution. The other option is to force index server #1 to utilize all SMT contexts of the first four physical cores and index server #2 to utilize all SMT contexts of the last four physical cores. This way index data locality is achieved by enforcing SMT contexts that belong to the same physical core, to perform search on the same index data but for different query. We will examine in the experimental results section which option is better. For the index dataset we used the one which comes with the CloudSuite Simics images [20]. We are using a 5GB index.

Table I. Tables shows qualitatively how each configuration affects latency and throughput.

Configuration	Throughput	Latency
No-SMT	++	++
SMT	+++	+
SMT+partitioning	++(+)	+++

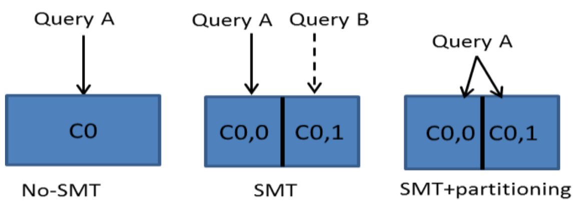

Figure 6. Illustration of how queries are executed without SMT, with SMT and with SMT combined with index partitioning

Table II. Server system parameters

Number of CPUs	2
CPU	Intel Xeon E5620 @ 2.4GHz
Number of physical cores per CPU	4
Number of logical cores (SMT contexts) per CPU	8
DRAM	32GB DDR3 1066MHz
Ethernet speed	1Gb

5. Experimental Results

5.1 SMT Throughput vs Latency tradeoff

Figure 1 shows how SMT impacts the throughput and latency of a server running index search. We evaluate the SMT impact on index search performance by comparing a web search run which utilizes 8 physical cores versus a run that utilizes 16 SMT contexts (our CPU has eight physical cores that support two-way SMT). The No-SMT configuration is being stressed with eight client threads, whereas the SMT configuration is stressed with 16 client threads. The SMT configuration uses more clients to better assess the throughput benefits of SMT; whereas the No-SMT configuration uses eight client threads to prevent latencies from being dominated by queuing effects. The throughput is increased approximately ~1.35X with SMT because more threads are available to serve independent query requests. But the average and 99th latency increase approximately by ~1.5X. With SMT enabled single thread performance is degraded; therefore, each query is served slower compared to the 8 physical cores configuration.

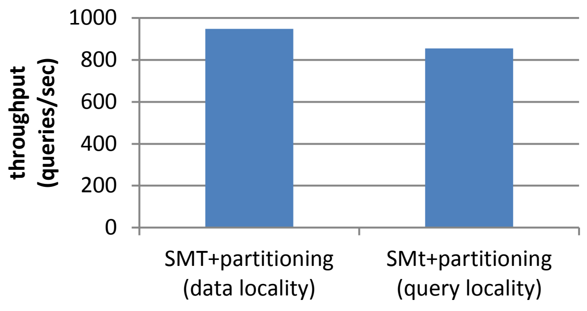

Figure 7. Throughput comparison between two different ways for combining SMT with partitioning

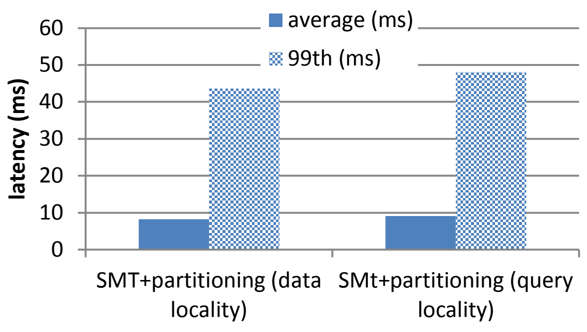

Figure 8. Latency comparison between two different ways for combining SMT with partitioning

5.2 Performance with Different SMT Configurations

The results in Figures 7 and 8 examine how is better to run the SMT+partitioning configuration: with query locality or with index data locality (Section 3,4). Intuitively the index data locality will provide better opportunities for caching and, therefore, better performance. Figure 7 compares the two configurations in terms of throughput and Figure 8 in terms of average latency and 99th percentile. Indeed, data locality provides better throughput and lower latency. Thus, for the rest of this section this configuration will be used for assessing SMT+partitioning.

Figure 9 compares the No-SMT configuration with the SMT+partitioning in terms of throughput and Figure 10 in terms of response latency. SMT+partitioning provides 1.28X better throughput, 1.25X lower average response time and 1.34X lower 99th percentile tail latency. The results show that the latency improvement from index partitioning and parallel execution across SMT contexts, overcomes the single thread performance degradation caused by SMT. This yields latency and throughput improvement over the configuration that does not use SMT.

6. Related work

This section presents the related work and describes how this paper differs from previous work.

The impact of SMT on the performance of HPC application is examined in [1,12,13]. Also, the following works [21,22,23,24] examined the SMT impact on the performance of operating system, database and network workloads. [4] examined the impact

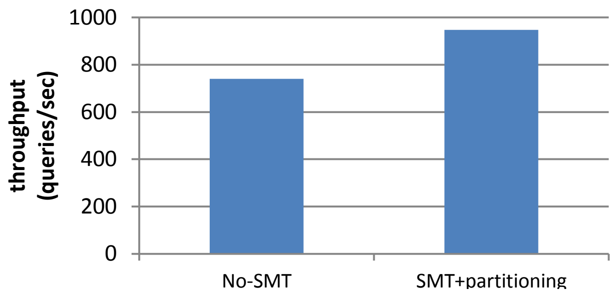

Figure 9. Throughput comparison

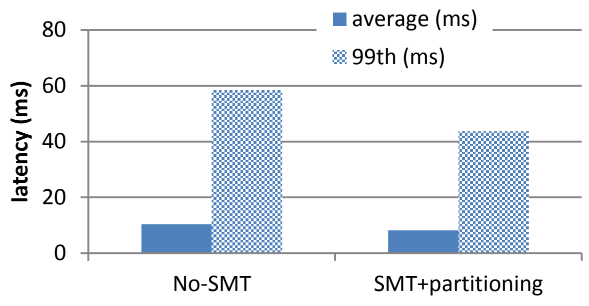

Figure 10. Latency comparison

of SMT on cloud workloads but only from the perspective of throughput not from the perspective of QoS. In [2,3] they identify that SMT has detrimental effect to QoS of online cloud workloads and attempted to provide solutions for increasing server utilization by enabling the use of SMT without increasing the server latencies. On the other hand, our work does not examine SMT colocation but shows how a parallel online workload can exploit SMT for improving latency. The closest related work is [22] which evaluated parallel execution of database queries across SMT contexts.

Also, in [8,9,25] the authors examined how web search can benefit from parallel execution across CPU cores. Again this work is very similar to ours with the key difference been that previous work does not evaluate parallelism across SMT contexts. Actually a lot of previous works does not use SMT for easing the analysis. This shows that SMT is a feature often left unexploited and underlines the importance of understanding how to use SMT more efficiently.

7. Conclusions

The main contribution of this paper is to show that SMT can be latency friendly for interactive services that are amenable to dataset partitioning and parallel execution. The experimental findings indicate that the proposed approach can help servers used for web search to utilize more efficiently and effectively their SMT capable servers.

For future work we plan to evaluate the SMT impact on performance of more cloud workloads. We plan to evaluate partitioning across SMT contexts on other workloads or evaluate other ways to exploit SMT for the favor of latencies. Also, we plan to enrich our work with: a) power measurements, b) analysis of how scaling at multiple servers affects the tail latencies, and c) TCO estimations.

8. REFERENCES

[1] Porter, Leo, et al. "Making the most of smt in hpc: System-and application-level perspectives." ACM Transactions on Architecture and Code Optimization (TACO) 11.4 (2015): 59.

[2] 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.

[3] 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.

[4] 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.

[5] Meisner, David, et al. "Power management of online data-intensive services." Computer Architecture (ISCA), 2011 38th Annual International Symposium on. IEEE, 2011.

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

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

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] http://www.internetlivestats.com/google-search-statistics/

[15] Apache Lucene. http://lucene.apache.org/, 2014.

[16] Reddi, Vijay Janapa, et al. "Web search using mobile cores." Proceedings of the 37th annual international symposium on Computer architecture (ISCA). 2010.

[17] 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

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

[19] Mars, Jason, et al. "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations." Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2011.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

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

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

[28] 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.

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

[30] 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