Adaptive Parallelism for Web Search

Myeongjae Jeon[†], Yuxiong He[*], Sameh Elnikety[*], Alan L. Cox[†], Scott Rixner[†]

[*]Microsoft Research
Redmond, WA, USA

[†]Rice University
Houston, TX, USA

Abstract

A web search query made to Microsoft Bing is currently parallelized by distributing the query processing across many servers. Within each of these servers, the query is, however, processed sequentially. Although each server may be processing multiple queries concurrently, with modern multi-core servers, parallelizing the processing of an individual query within the server may nonetheless improve the user's experience by reducing the response time. In this paper, we describe the issues that make the parallelization of an individual query within a server challenging, and we present a parallelization approach that effectively addresses these challenges. Since each server may be processing multiple queries concurrently, we also present a adaptive resource management algorithm that chooses the degree of parallelism at run-time for each query, taking into account system load and parallelization efficiency. As a result, the servers now execute queries with a high degree of parallelism at low loads, gracefully reduce the degree of parallelism with increased load, and choose sequential execution under high load. We have implemented our parallelization approach and adaptive resource management algorithm in Bing servers and evaluated them experimentally with production workloads. The experimental results show that the mean and 95th-percentile response times for queries are reduced by more than 50% under light or moderate load. Moreover, under high load where parallelization adversely degrades the system performance, the response times are kept the same as when queries are executed sequentially. In all cases, we observe no degradation in the relevance of the search results.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Threads; H.3.3 [*Information Storage and Retrieval*]: Information Search and Retrieval—Search process

EuroSys'13 April 15-17, 2013, Prague, Czech Republic

General Terms Algorithms, Design, Performance

Keywords web search, parallelism, response time

1. Introduction

We have become dependent on web search in our everyday lives. Moreover, we have come to expect that the search results will be returned quickly to us and will be highly relevant to our search query. More formally, web search operates under an SLA requiring short response times (*e.g.*, 300 ms) and high result relevance. In order to meet this SLA, web search engines maintain massive indices of documents, which are partitioned across hundreds or thousands of servers. Quickly finding relevant documents in such a massive index relies on the use of large-scale parallelism. Nonetheless, in practice, achieving both high responsiveness and high quality is still challenging because these two requirements are often at odds with each other.

Today, web search engines commonly achieve large-scale parallelism in two complementary ways. They process multiple search queries concurrently, and they distribute the processing of each query over hundreds or thousands of servers. In this paper, we explore a third complementary way of achieving parallelism. We study intra-query parallelization of index searching within a multi-core server. Specifically, we focus on how to parallelize a single query within one server that hosts a fragment of the web index, such that multiple processor cores cooperate to service the query. This is motivated by having processors with more cores rather than faster clock speeds, making intra-query parallelization an effective technique to reduce query response time.

At the query level, web search is embarrassingly parallel. Each server searches its fragment of the index. This fragment is effectively a sorted list of documents in order of decreasing "importance". Documents that are more important are given a higher static priority. When performing a web search, these documents are searched first for relevant results. The relevance of a result is a combination of the static priority, indicating document importance, and the dynamic priority, indicating relevance to the search terms. A web search can terminate early if it discovers that the static priority of the remaining pages is such that further work is unlikely to yield any better results for the current query. Therefore, a sequen-

tial search of the ordered index almost never scans the entire index. In fact, about 30% of queries need only search 10% of the index.

Unfortunately, this early termination of web searches makes intra-query parallelization challenging. It is not obvious how to partition tasks within a single query in order to effectively search the index without performing large amounts of unnecessary work. Since a sequential search can terminate the search early, a parallel search will almost always scan more of the index. Documents with a higher static priority are being scanned concurrently with documents with a lower static priority. So, the time spent looking at the lower priority documents is wasted if enough relevant results are found in the higher priority documents.

To reduce such wasted work, this paper introduces a dynamic fine-grain sharing technique that parallelizes each individual request while preserving the sequential order of execution. This technique limits wasted work and achieves good load balancing.

In addition, any intra-query parallelization strategy has to be sensitive to both system load and parallelization overhead. Blindly using high degrees of parallelism pays off under low load, dramatically reducing response latency. However, under high load, the unnecessary work delays waiting queries.

To address these additional problems, this paper introduces an *adaptive* parallelization strategy that dynamically selects the degree of parallelism on a query-by-query basis. The selection is made based upon the current load and the estimated cost of parallelizing the query. The cost amounts to a combination of the estimated benefits to the parallelized query and the estimated delays on subsequent waiting queries.

We show that the adaptive strategy outperforms the baseline sequential execution that is performed today on production servers. The delay is decreased or at least the same and the relevance of the results is always the same or slightly better.

When system load is light, using fixed degrees of parallelism for all queries results in latency reductions of up to 50%, when compared to sequential execution. However, as load increases, the system becomes saturated and latency increases rapidly. This motivates both the need for parallelism to reduce latency and the need for adaptation to prevent system saturation, depending on the system load. However, existing adaptive techniques [26] decide the request parallelism degree using only system load without considering request parallelization efficiency. This results in improvements over using a fixed degree of parallelism. However, it is hard to decide how to decrease the degree of parallelism with increased load without considering the efficiency with which an individual request can be parallelized. Either the scheme will be too conservative and not reduce latency as much as possible, or the scheme will be too aggressive and latency will increase beyond sequential at higher loads.

Our adaptive parallelization strategy achieves the best of both worlds. It dynamically adjusts the parallelism on a query-by-query basis considering both system load and parallelization efficiency. Finally, it selects sequential execution automatically under high load, preventing saturation and elevated latencies. We implement and evaluate these techniques in the context of a commercial search engine, Microsoft Bing. Our experimental results using workloads from a production environment show a reduction in both the mean and 95th-percentile response time by more than 50% under light or moderate load at which web search servers would typically operate. Moreover, the results show no increase in the response times under very high load. In all cases, no degradation in the relevance of the search results is observed.

The contributions of our work are the following:

- We develop a dynamic fine-grain sharing technique that partitions the indices and parallelizes query processing with little wasted work and good load balancing.
- We develop an adaptive algorithm that decides the degree of parallelism for each request at run-time using system load and parallelization efficiency.
- We implement our parallelization techniques in a production environment, Microsoft Bing, and evaluate it with production workloads.

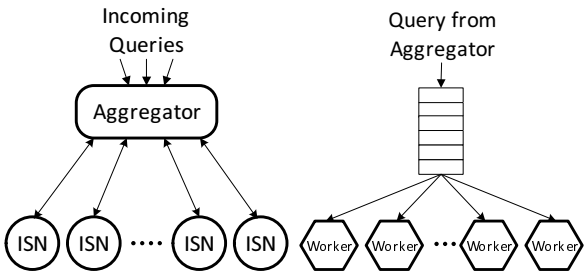
This paper is organized as follows. We first present background on web search in Section 2. Then, we introduce opportunities and challenges of query parallelization in Section 3. Section 4 and 5 address two challenges respectively: (1) how to partition and process index data, and (2) how to determine request parallelism at runtime. Section 6 presents our experimental results. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2. Web Search Engines

We focus our discussion here on the index serving part of the web search engine that processes user search queries (interactive processing), rather than on the web crawler and indexer (batch processing). In this section, we present the architecture of the index serving system in the Bing search engine, its data layout and query processing.

2.1 Requirements

Response time. Achieving consistently low response times is a primary design requirement for web search engines. A web search service is required to respond to a user query within a bounded amount of time. Since users are highly sensitive to the server's response time, a small increase in query response time can significantly degrade user satisfaction and reduce revenue [16, 28]. Therefore, it is important for service providers to lower the mean and high-percentile response times for their services.



(a) Partition-aggregate structure (b) Query processing in ISN

Figure 1. Search engine architecture.

Relevance. A web search engine must provide results that are relevant to the search keywords. Relevance is a complex metric to which many factors contribute. In general, the relevance of results improves with a larger web index containing more documents, with more features extracted from the documents, and with more sophisticated ranking functions. However, all of these factors contribute to greater resource demands and longer query processing times.

2.2 Search Architecture

The index serving system is a distributed service consisting of aggregators (also known as brokers) and index serving nodes (ISNs). An entire web index, consisting of billions of web documents, is partitioned and stored in hundreds of ISNs. When the result of a request is not cached, the index serving system processes and answers it. Aggregators propagate the query to all ISNs hosting the web index, ISNs find matching documents for the query in their part, and aggregators collect the results.

Figure 1(a) illustrates the *partition-aggregate* architecture, consisting of an aggregator and its ISNs. When a query arrives, an aggregator broadcasts it to its ISNs. Each ISN searches a disjoint partition of the web index and returns the most relevant results (currently 4). Each ISN returns these results to the aggregator within a time bound (*e.g.*, 200 ms). Late responses are dropped by the aggregator. After aggregating and sorting the results, the aggregator returns the top N documents to the user. Due to the large number of ISNs, several levels of aggregators may be used.

ISNs are the workhorse of the index serving system. They constitute over 90% of the total hardware resources. Moreover, they are along the critical path for query processing and account for more than two-thirds of the total processing time. Any ISN that fails to meet the time bound cannot contribute to the final response, resulting in degraded response quality and wasted resources.

2.3 Query Processing in the ISN

Workflow. Figure 1(b) shows the workflow for query processing in an ISN. Currently, each query is processed sequentially by the ISNs, but each ISN processes multiple queries concurrently. Newly arrived requests join the waiting queue. When the waiting queue becomes full, new requests will be dropped. An ISN manages a number of *worker* threads. Each worker processes a single request at a time. When a worker completes a request, it gets a new query from the head of the waiting queue and starts to process it.

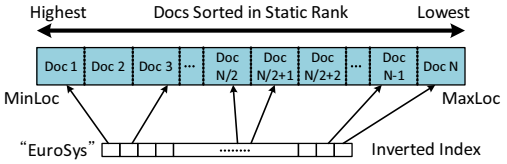


Figure 2. Sorted web documents and inverted index.

The number of workers is at least equal to the number of cores in the system. Typically, there are twice as many workers as cores, to account for workers who block on I/O. However, blocking on I/O is rare because the indices are partitioned, which promotes locality at the ISNs. Moreover, most of the remaining I/O can be performed asynchronously.

Data layout and query processing. When an ISN worker thread processes a query, it searches its web index to produce a list of documents matching the keywords in the query. It then ranks the matching documents. This is the most time consuming part of the ISN's work because the ranking function extracts and computes many features of the documents.

The ISN uses an inverted index (also called posting list) to store information about documents. Figure 2 shows an example of the data layout of the inverted index, where the documents are sorted based on their *static rank*. The static rank of a document depends only on its own features such as its popularity; it does not depend on the query. When a worker matches and ranks the documents by following the inverted indices, it processes the documents with higher static ranking first as they are more likely to contribute to the top matching results of the query. During the ranking process, each match is also "dynamically" scored to compute a *relevance score*. The relevance score of a document depends on its static rank as well as on many other features, including for example, where and how many times the query keywords appear in the document, the distance among multiple keywords, user location and context, etc.

When the search query has multiple keywords, the ISN finds matches with all/many keywords appearing in the same document. As queries are normally processed in the conjunctive mode, this is done by intersecting inverted indices for all keywords. The inverted indices are sorted based on document static rank, so this is a simple merge-join process on sorted data. The following explains a common algorithm for performing the intersection. For each element in the shortest index, a search method (*e.g.*, binary or interpolation search) is performed in the other indices. A matching document is found if the element belongs to all/many indices. Otherwise, the searching immediately stops, ignoring remaining

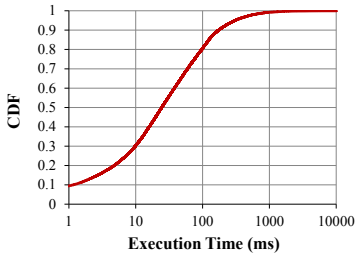


Figure 3. Semi-log CDF of query execution time.

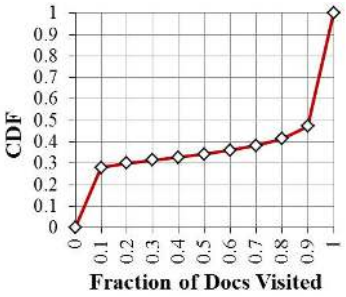


Figure 4. CDF of the fraction of index data processed.

indices, and moves to the next element in the shortest index. After iterating over all elements in the shortest index, the intersection terminates.

The inverted index is not implemented as a conventional array. Its implementation is similar to that of a skip list. The seek cost from one chunk to another is not constant but logarithmic on the size of the inverted index for a particular keyword. Moreover, the inverted indices are compressed, exploiting more advanced data structures that trade-off among space, decompression cost, and sequential and random access efficiency.

Query termination. Once an ISN finds results for a query that are good enough, it terminates the search and returns the results. This is called *early termination* [1, 7, 24, 29]. The ISN predicts how likely the results will improve from scoring the remaining documents and terminates the execution when the probability is low. Early termination is effective because the inverted indices are sorted based on the static rank of the web pages, which lists important and popular web pages first. Web pages processed earlier are more likely to rank higher and contribute to the top results of the query. The ISN has an algorithm that takes the current top documents and the static ranking of the later documents as inputs, and it decides if the termination condition is satisfied. The specific details of this algorithm are beyond the scope of this paper. With early termination, there is a small, but non-zero, probability that more relevant results might be missed.

3. Opportunities and Challenges on Parallelization

We introduce intra-query parallelism into the ISN. In this section, we discuss important workload characteristics of web search that create both opportunities and challenges for such query parallelization. The methods for collecting the data are the same as described in Section 6. For brevity, we henceforth use the term query parallelization to mean intra-query parallelization, unless specified otherwise.

3.1 Opportunities

Long requests require parallelization. Search requests have varying computational demands. Figure 3 shows the cumulative distribution of request service demands from Bing; where the X-axis is in log scale. We observe a large gap between short and long requests[1]. In particular, the 95th-percentile service demand is 3.68 times the average. The gap between the median and the 95th-percentile is even larger at 12.37 times. These results show that there are a large number of short requests but long requests dominate the total work: the longest 25% of the requests represent more than 81% of the total work. These long requests greatly affect the user experience: users can hardly tell if a query takes 20 ms or 80 ms to complete, but the difference between 200 ms and 800 ms matters a lot, as 200–300 ms is a critical threshold for query processing latency in many online services [28].

Two factors are typically responsible for these long queries. First, these queries tend to have more documents to match and score. This requires frequent invocations of ranking inference engines which usually consume a significant number of processor cycles for scoring each match. Second, these queries involve the intersection of a larger number of inverted indices. It is known that the average latency of queries with ten keywords is approximately an order of magnitude greater than that of queries with only two keywords [31].

Computationally intensive workload allows parallelization. The ISN performs complex calculations, such as computing the relevance of matches. Furthermore, the indices are partitioned across nodes so as to create locality and reduce I/O accesses. These factors make the ISNs computationally intensive. In fact, web search exhibits higher instructions per cycle (IPC) than traditional enterprise workloads [17]. This also means that processor cache misses are relatively infrequent. Therefore, we would expect queries running in parallel on an enterprise-class server to exhibit little slowdown due to memory contention.

This is, in fact, the case. At 10 queries per second (QPS), only one core is active on average, and the total CPU utilization across 12 cores is 7%. At 70 QPS, only 6 cores out of 12 are active on average, and the total CPU utilization is 50%. The difference in average response latency between these two cases is only 5%, confirming that the interference among concurrently running queries is negligible.

[1] In this paper, we use "long" or "slow" to indicate queries with high computational cost and long processing time. The term "short" or "fast" thus has the opposite meaning.

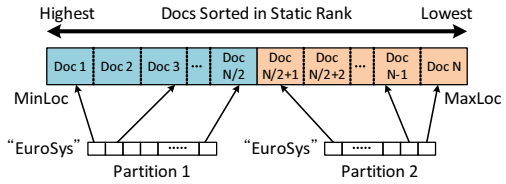


Figure 5. An example of static coarse-grain processing with 2 threads. Both web documents and inverted index are accordingly bipartite.

3.2 Challenges

Parallelization based on static data partitioning and index processing is ineffective at achieving early termination. As a result of early termination, it is common that documents with low importance are rarely visited during sequential execution. Figure 4 shows the cumulative distribution of the fraction of inverted indices visited by Bing ISNs. The measurements indicate that about 30% of the queries use only 10% of the documents. Moreover, more than 38% of the queries never need to score matches on the second half of the documents. This poses challenges to query parallelization. We present two basic approaches and discuss their performance limitations.

One possible approach, *static coarse-grain processing*, partitions the data into the same number of chunks as there are threads assigned to the query. This approach may, however, introduce a large amount of speculative execution because all matching documents will be processed. Consider a simple example: we partition the inverted index of a query into two chunks and process each chunk with a different thread, as shown in Figure 5. Much of the work performed by the second thread is likely to be useless, since 38% of the queries (Figure 4) are unlikely to yield any relevant results from the second half of the index.

Another approach, *static fine-grain alternating*, partitions the data into many, smaller chunks, and assigns these chunks to the threads in an alternating fashion. For example, we label the partitioned chunks of the inverted index based on their static order so we have chunks 1 to N. When we use two threads, one thread processes odd chunks while the other thread processes even chunks, and each thread processes their chunks based on their static order so more important chunks are processed first. While processing the chunks, the threads communicate with each other to merge the top results they have found so far. Thus, a thread can terminate early without processing all assigned chunks once the current top results are good enough. When concurrent threads of a query advance at a similar rate, static fine-grain alternating behaves similarly to the sequential order of execution and it helps to reduce the amount of speculative execution.

The static fine-grain alternating approach still has two issues: (1) The amount of computation per chunk and per thread may be uneven, resulting in unbalanced load among threads. (2) Multiple threads of a query may not always be co-scheduled since there are more threads than processor cores. This can lead to different completion times for the threads. Moreover, when one thread is delayed and does not process its most important chunks, other threads may perform a larger amount of speculative work because early termination does not occur promptly.

Therefore, to parallelize web search, our first challenge is to partition and process the index data efficiently to reduce additional work.

Fixed parallelism is inefficient. Load on search servers varies over time [23], so does the availability of idle resources. Having fixed parallelism for queries cannot produce low response times under all loads. We need a robust adaptive parallelization scheme that decides the degree of parallelism at run-time based on system load and request characteristics. This is our second challenge.

We address these two challenges in Sections 4 and 5.

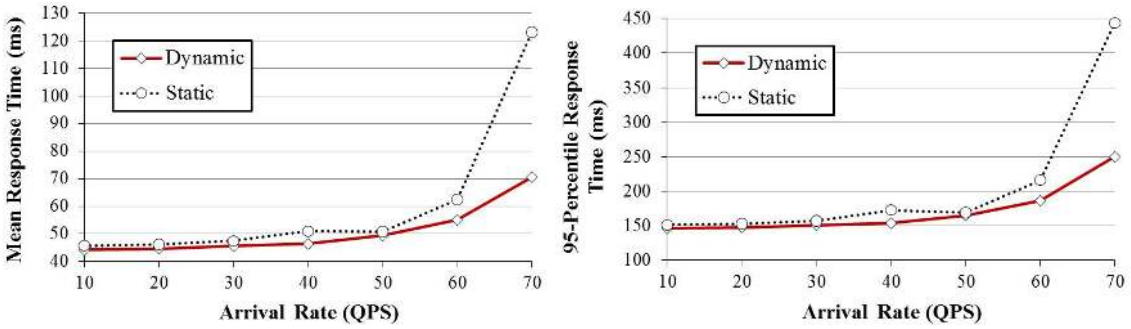
4. Query Parallelization

This section presents how to partition and process the index data in parallel to reduce wasted, speculative work given a fixed parallelization degree. We propose a parallelization technique — *dynamic fine-grain sharing* — that mimics the sequential order of execution with small synchronization overhead and good load balancing among threads. This keeps the overhead of parallelization low while producing results with comparable quality to sequential execution. In this section, we first describe how we partition and process the index data using concurrent threads. Then, we discuss the communication among threads and how to reduce the overhead. Finally, we show the speedup results of using our parallelization technique.

4.1 Index Data Partitioning and Processing

Dynamic fine-grain sharing partitions the index into small chunks. When a thread becomes idle, it grabs the most important unprocessed data chunk. To implement this technique, we need only use a counter representing the ID of the next most important unprocessed chunk. An idle thread uses an atomic fetch-and-increment operation to claim this chunk and update the counter to reference the next most important chunk. The threads communicate with each other to merge the top results they have found so far. When a thread finds that the current top results are good enough and the later chunks are unlikely to produce better results, the thread stops processing the query, reducing computation overhead.

Compared to the static fine-grain alternating approach in Section 3, dynamic sharing more closely mimics the sequential order of execution regardless of thread scheduling. It also achieves better load balancing. Even when threads of a query are not co-scheduled, the active threads of the query still process the chunks based on their sequential order. Moreover,



(a) Mean response time

(b) 95th-percentile response time

Figure 6. Comparison between dynamic fine-grain sharing and static fine-grain alternating using degree-3 parallelism.

as the dynamic sharing is similar to having a shared global queue of tasks, good load balancing is attained. Nonetheless, the synchronization overhead is quite small, because obtaining a task only requires a single atomic fetch-and-increment operation.

Figure 6 compares the performance between the static alternating and dynamic sharing techniques. The experimental setup is the same as that in Section 6. The figure shows that dynamic sharing always outperforms static alternating with lower mean and 95th-percentile response time. Under light load from 10 to 50 QPS, the performance gap between the static and dynamic approaches is smaller: the static approach takes 3% to 12% longer than dynamic sharing for both the mean and the 95th-percentile response time. The inefficiency of static alternating is mostly caused by the uneven work distribution among chunks. Under heavy loads, however, both thread scheduling and uneven work distribution among chunks affect response time. Specifically, due to the imbalance of OS thread scheduling, when a delayed thread did not get a chance to process its most important chunks, other threads may incur a larger amount of wasted, speculative work. These threads evaluate lower static-rank documents that are unlikely to be the top results, increasing response time. As shown in the figure, under 70 QPS, the mean and 95th-percentile response times for static alternating are 75% and 78% higher respectively than dynamic sharing.

4.2 Thread Communication

Threads must communicate to determine when to terminate the search. There are different approaches for threads to communicate and share information. A simple way is for all threads to maintain one global data structure and update this data structure whenever a thread finds new results. The problem with this simple approach is the potential for contention among threads. Processing of a query often evaluates thousands of matches, and thus synchronizing for every new match is expensive.

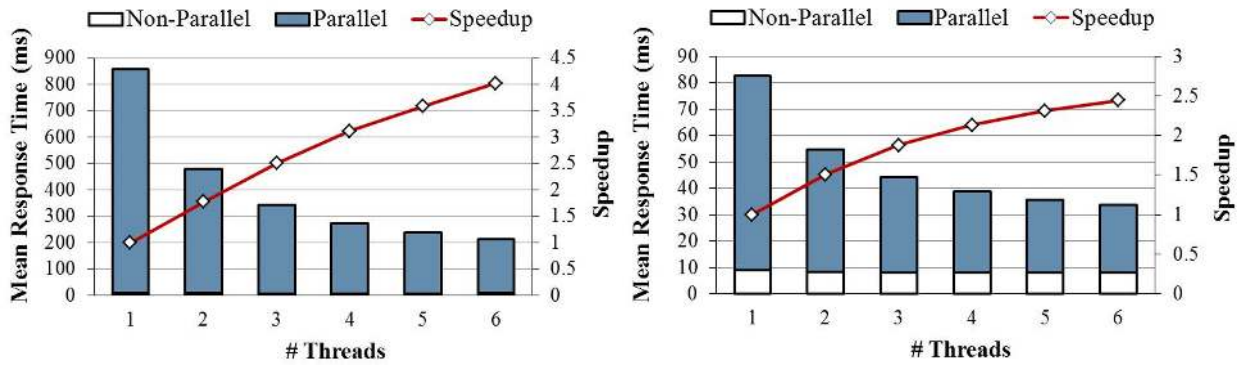
Our approach uses batched updates. We maintain a global heap for each query and a local heap for each thread executing the query. A thread periodically synchronizes the information in its local heap with the global heap, not directly with other threads. Specifically, we synchronize the global heap and a thread's local heap only when a thread completes its computation for a chunk. Batch updates allow threads to see the results of other threads soon enough to facilitate effective early termination without introducing much synchronization overhead. The chunk size is tunable to tradeoff between the synchronization overhead and the amount of speculative execution, but data compression limits the smallest chunk size. We determine the appropriate chunk size empirically.

4.3 Speedup with Query Parallelization

Figure 7(a) shows that for the 5% of the queries that run longest, we achieve more than 4 times speedup by using 6 cores. This reduces their mean response time from 856 ms to 212 ms. This is an important improvement: to provide a pleasing user experience, the service's response to a user request must occur typically within a couple hundred milliseconds, around the limits of human perception.

Long requests achieve good speedup for two reasons. First, the dynamic data partitioning and processing scheme effectively controls the amount of speculative execution. As compared to a sequential execution, a parallel execution with degree p is unlikely to visit more than $p - 1$ extra chunks. Since sequential executions of the long requests visit a few hundred chunks on average, the additional $p - 1$ chunks constitute a small percentage of the overhead. Second, our parallelization focuses on the matching and ranking of the inverted indices to find the top documents of a query, however, there are parts of the query processing that are not parallelized, e.g., query parsing and rescoring of the top results. For the longest queries, the non-parallelized part constitutes a very small part of the total execution time.

Figure 7(b) shows the execution time reduction for all queries. Using 6 threads, we achieve about 2.5 times speedup. Short queries benefit less from parallelization for two reasons. First, the non-parallelized part of a short query is a larger fraction of the total response time, reducing the ben-



(a) Top 5% longest queries

(b) All queries

Figure 7. Breakdown of execution time and speedup in parallelization. The execution time is divided into non-parallel and parallel parts. There is more than 4 times speedup when using 6 threads for parallelizing long queries.

efits of parallelization. Second, the time spent speculatively processing the additional chunks constitutes a larger fraction of the total response time.

Although reducing the chunk size can reduce the amount of speculative work, it may add to other overheads. For example, reducing the chunk size will increase the frequency of moving between non-adjacent chunks. This has a non-trivial cost because the inverted index cannot be accessed like an array. Also, it will increase the synchronization overhead.

5. Adaptive Parallelism

A fixed degree of parallelism cannot reduce response times for all loads. For example, with only one query in the system, it is better to use as many cores as possible to process the query, rather than leaving the cores idle. In contrast at heavy load, high degrees of parallelism may introduce substantial overhead: as there are little spare resources, the parallelization overhead of a request takes resources away from waiting requests. We propose an adaptive parallelization algorithm that decides the query parallelization degree at runtime on a per query basis to reduce the response time of requests over different loads. We first elaborate on the factors that affect the desired degree of parallelism for a request; then we describe the adaptive parallelism algorithm.

5.1 Factors Impacting Parallelism Degree

The adaptive parallelization algorithm considers two factors: system load and parallelization efficiency.

System load. Our algorithm uses the number of waiting jobs (*i.e.*, queries) in the queue to model the instantaneous system load. Compared to query arrival rate, queue length captures fine-grain changes on system load. For example, even at the same average query arrival rate, we directly observe the change of queue length due to transient overload or underload, where different parallelism degrees may be used for better performance.

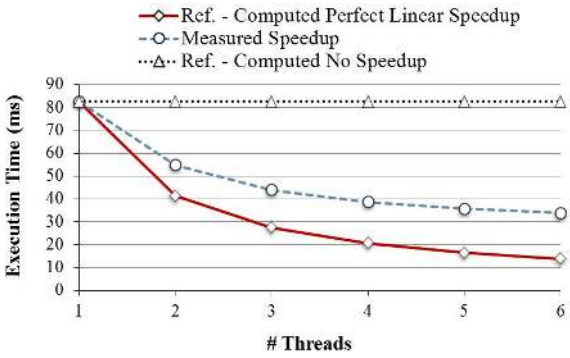


Figure 8. Request execution time profile.

Parallelization efficiency. Our algorithm uses a *request execution time profile* to model request parallelization efficiency, capturing parallelization overhead. The request execution time profile maps the degree of parallelism to an expected execution time. As the exact execution time of a query is not known in advance, we use the average execution time as measured across all requests as an estimate. Figure 8 presents the measured average execution time profile of Bing requests, executed one at a time, using production index and query log based on our parallel implementation. For reference, Figure 8 also presents the execution time profile assuming ideal speedup or no speedup.

The parallelization efficiency directly impacts the desired degree of parallelism. Requests whose response time decreases linearly with increased parallelism should utilize as many cores as possible. In contrast, requests whose response time would not decrease with increased parallelism should run sequentially. However, in reality, most requests fit neither of these extremes. So, requests fall along a continuum of desired parallelism.

More generally, the type and service demand of a request can also influence parallel execution time. However, in Bing, requests have the same type, and the scheduler has no prior

knowledge of the computational cost of a request. Therefore, we do not use these factors to improve the decision.

5.2 Determining Parallelism Degree

We admit a request from the waiting queue to the processor when the total number of active threads in an ISN is less than $2P$ where P is the total number of cores. Before executing a request, we decide its degree of parallelism using Algorithm 1. The algorithm takes system load and the request execution time profile as inputs and returns the parallelism degree for the current job. The algorithm uses a greedy approach to choose the parallelism degree that increases the total response time the least. More specifically, the $\arg\min$ operator at Line 5 searches for the parallelism degree from 1 to P that minimizes the estimated increase on the total response time and returns that parallelism degree.[2] The algorithm is computationally efficient with complexity $O(P)$, where P is the total number of cores in the server.

The expression (in Line 5) estimates the increase in total response time from executing a job J with parallelism degree i. The expression consists of two parts: (1) *self-increase*: the execution time of the job J, and (2) *peer-increase*: the additional time that other jobs in the system have to wait because of the execution of job J. As the precise execution time of J is unknown in advance, we estimate the self-increase of J using average job execution time T_i when using parallelism degree i.

We estimate the amount of peer-increase as the product on the amount of time that J delayed other jobs and the number of jobs that J delayed. The amount of delayed time is T_i. We estimate the number of jobs J delayed as $(K-1)\times i/P$ where K is the total number of jobs waiting in the queue (including job J). Here the number of jobs affected by J is proportional to how many other jobs waiting in the queue, which is $K-1$. When J uses i cores, it only delays the jobs using these i cores, which we estimate as an i/P proportion of the total jobs. So the expected number of jobs J delays is $(K-1)\times i/P$, which gives peer-increase $T_i\times(K-1)\times i/P$. The expression in Line 5 adds the self-increase and peer-increase of J, producing the increase on total response time.

Impact of load. The following examples illustrate how the algorithm works in response to system loads. First, at very light load with only job J in the system, the self-increase of J dominates its increase on total response time, so small T_i is better. As larger parallelism degree often reduces execution time, our algorithm chooses a high parallelism degree at light load. Second, at heavy load, the peer increases dominates due to large K. To minimize total response time increase, we need to minimize the value of $T_i \times i$, which is the total amount of work job J incurs. Sequential execution often gives the minimum because of parallelization overhead. Therefore, our algorithm chooses a low parallelism degree at heavy load.

[2] The operator $\arg\min f(i)$ returns parameter i that minimizes function $f(i)$. Here the argument i denotes the parallelism degree and $f(i) = T_i + (K-1)\times T_i \times i/P$ represents the estimated response time increase using parallelism degree i.

Algorithm 1 Deciding the parallelism degree of a job

Input:
1: P: total number of cores in the server
2: Instantaneous system load K: the total number of jobs in the waiting queue
3: Request execution time profile $\{T_i | i = 1...P\}$: T_i is the average job execution time with parallelism degree i

4: **procedure** ADAPTIVE
5: **return** $\arg\min_{i=1...P}(T_i + (K-1)\times T_i \times i/P)$
6: **end procedure**

Impact of parallelization efficiency. We first consider two extreme cases. (1) For a job with perfect linear speedup, the values of $T_i \times i$ are the same for any parallelism degree $i \leq P$. To minimize the total increase on response time, peer-increase does not change with i value, thus the algorithm chooses the i value minimizing the self-increase, *i.e.*, a fully parallelized execution with $i = P$. (2) For a job without any speedup, the values of the execution time T_i are the same for any $i \leq P$. To minimize the total increase on response time, self-increase does not change with i value, thus the algorithm chooses the i value minimizing the peer-increase, *i.e.*, a sequential execution with $i = 1$.

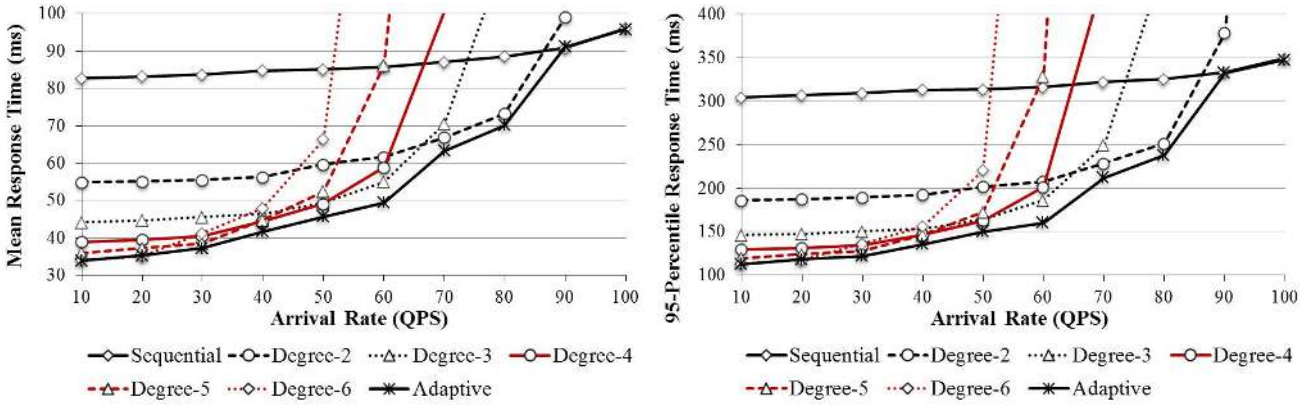
More generally, when a request has high parallelization efficiency and low overhead, the adaptive algorithm is more aggressive at using higher degrees of parallelism: higher degrees reduce the self-increase significantly while increasing the peer-increase slightly. On the other hand, when a request has low parallelization efficiency, our decision is more conservative: higher degrees significantly increase the peer-increase as the total work of the parallel execution is much larger than sequential.

6. Evaluation

6.1 Experimental Setup

Machine setup and workload. The index serving node (ISN) for our evaluation has two 6-core Intel 64-bit Xeon processors (2.27 GHz) and 32 GB of main memory and runs Windows Server 2012. We use 22 GB of memory to cache recently accessed web index data. The ISN manages a 90 GB web index, which is stored on an SSD to enable fast index data fetching and to avoid performance degradation from interleaved I/O accesses. The number of worker threads is set to twice the number of cores, as workers sometimes block on I/O accesses. The OS scheduler assigns the threads of a query to the available cores.

Our evaluation includes an ISN that answers queries and a client that replays queries from a production trace of Bing user queries. The trace contains 100K queries and was obtained in June 2012. We run the system by issuing queries



(a) Mean response time

(b) 95th-percentile response time

Figure 9. Response time of sequential execution, fixed parallelism, and adaptive parallelism. The adaptive parallelism performs better than any of fixed parallelism configurations. It also achieves significant reductions in both mean and 95th-percentile latencies than sequential execution.

following a Poisson distribution in an open-loop system. We vary system load by changing the arrival rate expressed in QPS (queries per second). The ISN searches its local index and returns the top 4 matching results to the client with relevance scores. This is a standard configuration at Bing. However, with more results returned from each ISN, the processing time of queries may increase. Our parallelism techniques work in all cases.

Index chunk size. We empirically explore various chunk sizes and use the one resulting in the smallest average query execution time. In our experiments, the index space in the ISN is partitioned into 200 chunks. A query processes 117 chunks on average as some queries terminate early. We observe that the performance is good overall and stable if the average number of chunks processed per query is between 100 and 200. With larger chunk sizes, the overhead due to speculative execution is too high and the load among the threads is not balanced. These are no longer a problem with smaller chunk sizes, but the benefits are offset by the synchronization cost and the seek cost across chunks.

Performance metrics. We compare adaptive parallelism to both sequential execution and fixed parallelism. We use the following metrics:

- *Response time*. We measure query latency at the ISN from the time that it receives the query to the time that it responds to the client. Both mean response time (*Mean*) and 95th-percentile response time (*95th-Percentile*) are presented. Our measurements show that the improvement in 99th-percentile response time is similar to 95th-Percentile.
- *Response quality*. To evaluate the quality of a response to a search query under adaptive parallelism, we compare the relevance scores in the response to the relevance scores in sequential execution. We will explain the details of this methodology next.
- *Cost of parallelism*. We report average CPU utilization of the ISN and the I/O and network bandwidth overheads.

We collect the performance metrics after the ISN index cache is warmed to obtain steady state measurements.

Response quality metric. The quality of the response is an important measure for search engines and we want adaptive parallelism to provide very similar response quality to sequential execution. The quality metric we use is a relative comparison of the quality of the results from a *baseline* run (sequential) to a *test* run (parallelized).

To compare the quality of search results from two different runs, we do the following. First, we perform a pairwise comparison between relevance scores on the sorted results from the two runs. So, the "best" result from the baseline run will be compared to the "best" result from the test run, and so on. Then, for each pairwise comparison, the quality metric is incremented, decremented, or unchanged by the given weight (explained below) based on whether the test run has higher, lower, or the same relevance. This will produce a final "relative quality" metric that tells us how much better (or worse) the results from the test run are.

In the evaluation, the weights for the increment or decrement are assigned in two ways. We use *proportional weights*, which gives each of the top $N = 4$ documents the same weight. We also use another quality metric, *exponential weights*, to assign higher weights to higher ranking documents, as higher ranking documents are more important so they are more likely selected by the top-level aggregator to return to the user. We assign 8/15, 4/15, 2/15, and 1/15 for the highest to lowest results.

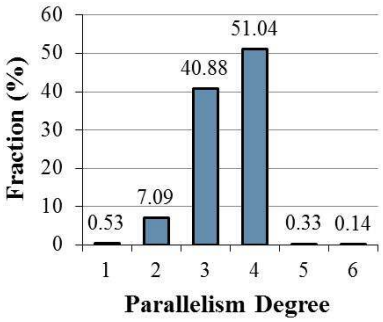
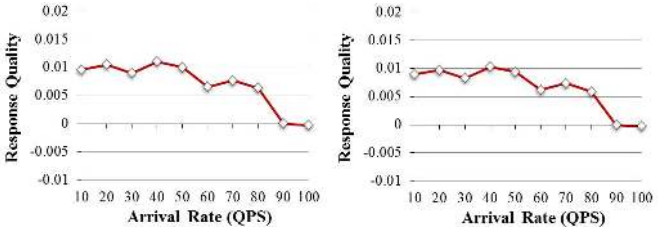


Figure 10. Distribution of selected degree in adaptive parallelism at 60 QPS.



(a) Proportional weights (b) Exponential weights

Figure 11. Response quality of adaptive parallelism over sequential execution. Non-negative quality values indicate Adaptive achieves slightly better or equivalent qualities compared with Sequential.

6.2 Adaptive Parallelism

6.2.1 Response Time

Figure 9 shows the *Mean* and *95th-Percentile* response time for all competing policies. For fixed parallelism, we evaluate 5 different degrees of parallelism (from *Degree-2* to *Degree-6*). We limit parallelism degree to 6 (number of cores on 1 CPU) because higher degrees do not perform better than degree 6 — we have already observed diminishing returns from degree 5 to 6 in Figure 7(b). Note that sequential execution (*Sequential*) is based on unmodified production code.

Figure 9 shows comparisons in two dimensions. First, it presents the performance of fixed parallelism over a variety of query arrival rates. The figure clearly shows that there is no fixed degree of parallelism that performs best across all the arrival rates. Second, the figure shows that our adaptive algorithm (*Adaptive*) performs the same or better than all other policies under all the arrival rates. In particular, for the arrival rates less than 90 QPS, adaptive parallelism outperforms all fixed parallelism solutions. For high arrival rate, 90 QPS and above, adaptive parallelism does not achieve any benefits, but it also does not incur any penalty.

As the figure shows, our adaptive approach dynamically selects the best degree of parallelism given the current arrival rate. As the arrival rate increases, the degree of parallelism for each query is reduced appropriately until the system is saturated at 90 QPS. This allows the adaptive strategy to achieve lower mean and 95th-percentile latencies under all arrival rates. For example, at 60 QPS, sequential execution has mean and 95th-percentile latencies of 86 ms and 316 ms, respectively. The best fixed parallelization strategy (Degree-3) lowers the mean and 95th-percentile latencies to 55 ms and 186 ms, whereas adaptive parallelism further reduces these latencies to 49 ms and 160 ms.

Note that Adaptive even outperforms the best fixed strategy at each arrival rate. That is because even then, there are queries being satisfied with different degrees of parallelism, more effectively utilizing the system. Figure 10 shows the distribution of the selected degree in the adaptive strategy at 60 QPS. The figure illustrates that 1) unlike fixed parallelism, Adaptive is able to select any degree among all possible options, and 2) these degrees are utilized unevenly to produce better performance. Adaptive parallelizes queries using a degree of 3 or 4 in most cases, with an average degree of 3.44. This behavior cannot be achieved by any fixed parallelism degree, and it enables Adaptive to perform better than any of the fixed parallelism configurations.

Many commercial data centers have server utilization less than 50% for certain production workloads [2, 5, 14], where intra-query parallelism yields significant latency reductions. For example, Bing search servers operate at between 30% and 50% of the maximum level for most of the time. Search engines have low CPU utilization by design, to avoid queueing and to provide consistently high quality responses; the servers are also over-provisioned for the events like a failure of a cluster or an entire data center. At the same time, if load spikes, Adaptive runs all queries sequentially so that average latency is no worse than the sequential case and system throughput is not compromised.

6.2.2 Response Quality

Figure 11 presents the response quality of adaptive parallelism (*Adaptive*) against sequential execution (*Sequential*) over varying arrival rates. For each QPS, Sequential becomes a baseline run to compute the response quality of Adaptive (which is the test run). Therefore, if Adaptive returns better relevance scores of search query responses than Sequential under a QPS, the figure will report a positive value at that QPS.

We see in Figure 11 that Adaptive produces better or equivalent response quality than Sequential over all arrival rates tested. Adaptive returns positive qualities for many cases (10 – 80 QPS), meaning that its relevance scores for search queries are overall higher. This is because parallel execution may cover a (slightly) longer prefix of the inverted indices due to speculative execution, having more documents scored for a search query than when it is executed sequentially. In 90 or 100 QPS, Adaptive executes each query sequentially and thus achieves an equivalent quality to Sequential.

Policy	10 QPS	30 QPS	50 QPS	70 QPS	90 QPS
Sequential	7%	22%	33%	50%	63%
Degree-2	8%	23%	38%	57%	**77%**
Degree-3	9%	26%	44%	69%	**87%**
Degree-4	9%	29%	50%	**76%**	**95%**
Degree-5	9%	30%	55%	**85%**	**≈100%**
Degree-6	10%	36%	62%	**91%**	**≈100%**
Adaptive	10%	31%	51%	65%	62%

Table 1. Comparison of CPU utilization. A policy & QPS with a **bold** value indicates worse response times than Sequential.

6.2.3 Cost of Parallelism

Table 1 compares the CPU utilization of the different policies. CPU utilization is periodically sampled using the performance counters in Windows. The fixed parallelization strategies incur increasing CPU utilization as more threads per query are used. This is expected, due to increasing parallelization overheads and additional speculative work. We observe that on the test system, response times increase at high rate when CPU utilization goes above 70%. The significant CPU contention that results under high arrival rates for these fixed parallelization strategies therefore increases both queue waiting time and query response time.

While the adaptive strategy (the last row) also consumes additional CPU cycles when it parallelizes queries, CPU utilization is always below 70% across a wide range of arrival rates. Therefore, we see that the adaptive strategy is able to balance CPU utilization and query response time effectively.

The adaptive strategy also incurs minimal overheads in terms of I/O and network bandwidth. While using the adaptive strategy, I/O bandwidth increases from 9 MB/s to 14 MB/s at 60 QPS. The increased I/O bandwidth is due to the increase in speculation placing additional pressure on the cache of the index. This increase, however, is marginal considering the bandwidth supported by SSDs and thus is not a limiting factor in the ISN. There is no observable network overhead; this is expected, as the network is primarily used to simply transfer the top matching results.

6.3 Dynamic Changes in Query Arrival Rates

We so far have presented results when the arrival rate is fixed. Here, we vary the arrival rate over time to show that the adaptive strategy can react to such changes. For this experiment, we begin with a query arrival rate of 30 QPS, increase it to 60 QPS, and then reduce it back to 30 QPS. Figure 12 shows the moving average of the response time against time in this experiment.

Figure 12 shows that the adaptive strategy is responsive to run-time query arrival rate changes with response times consistent with what we observed in Figure 9. Specifically, at 30 QPS, the adaptive strategy executes queries mainly using 5 or 6 threads per query to aggressively make use of idle cores. When the arrival rate goes up to 60 QPS, however, it more conservatively uses 3 or 4 threads per query. As a result, the response times observed under these arrival rates are consistent to those in Figure 9. The sequential response times remain relatively constant because the machine is not saturated at these arrival rates.

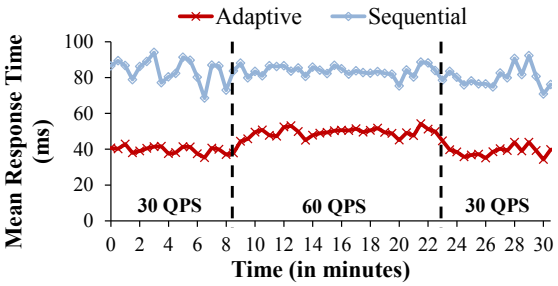


Figure 12. Mean response time with changing arrival rates between adaptive parallelism and sequential execution. Moving average of 30 sec is used to compute mean.

6.4 Comparison to Other Algorithms

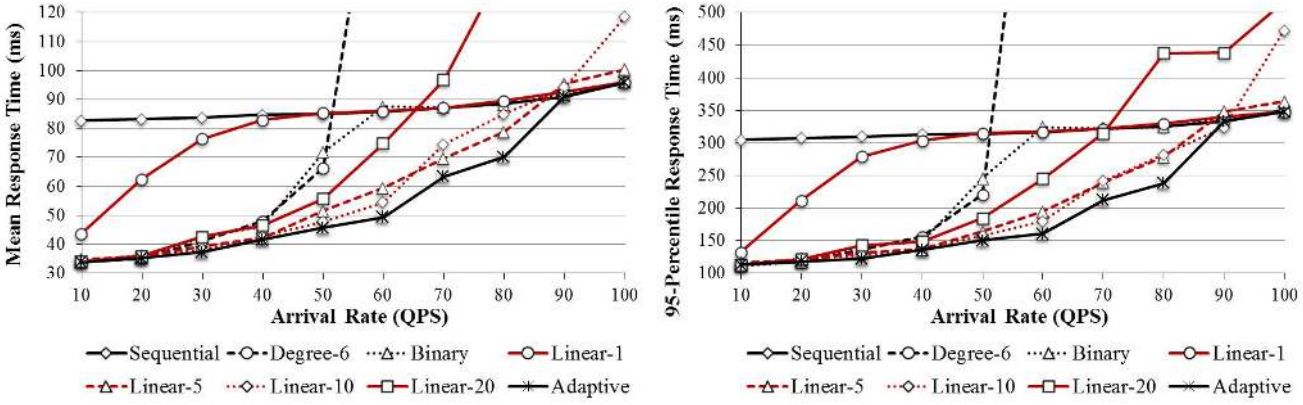
There are other possible adaptive parallelization strategies that could be used to parallelize queries. Here, we consider two schemes: *binary* and *linear* [26]. These schemes consider only instantaneous system load, which is the number of queries waiting in the queue as discussed in Section 5.

The binary scheme selects either no parallelism (sequential execution) or maximum parallelism (degree 6). It begins with sequential execution. If the system load remains below a threshold T for N consecutive queries, it starts parallelizing queries with degree 6. If the system load then remains above the threshold T for N consecutive queries, it returns to sequential execution. In all experiments, we set N to 50 and T to be the average load observed in Sequential at 50 QPS.

The linear scheme utilizes all degrees of parallelization between 1 and 6. If the system load goes above some maximum, T, then queries are not parallelized. Below T, queries are parallelized inversely proportional to the system load. The degree of parallelism is computed using the formula $max(1, 6 \times (T - K) \div T)$, where K is the instantaneous system load. The result is rounded to yield an integral degree of parallelism. Small values of T result in a more conservative approach where lower degrees of parallelism are used for even moderate system loads. Higher values of T result in a more aggressive approach that selects higher degrees of parallelism.

Figure 13 compares these schemes with various values of T. For comparison, we repeat the results of Sequential (sequential execution), Degree-6 (the highest degree of fixed parallelism), and Adaptive (our algorithm) in the figure.

Given the limited choices available to the binary strategy, one would expect it to vary in performance between sequential and Degree-6. As the figure shows, this is exactly what happens. As the threshold T is reached (around 50 QPS given the parameters we used), it transitions from



(a) Mean response time

(b) 95th-percentile response time

Figure 13. Response time comparison with other schemes, *Binary* and *Linear*, from prior work [26]. Adaptive performs better than both of them.

being close to the Degree-6 performance to the sequential performance. While the transition point can be tuned, this strategy fundamentally follows these two curves.

The linear strategy provides more freedom to utilize the system and balance performance. If the value of T is set to be too low (*i.e.*, $T = 1$), then the strategy will be too conservative and will largely track sequential performance. As Figure 13 shows, however, Linear-1 is still able to achieve gains over Sequential at low arrival rates. If the value of T is set to be too high (*i.e.*, $T = 20$), then the strategy will be too aggressive and will blow up. As the figure shows, it still outperforms a fixed degree of 6, but has simply moved the saturation point where response times increase dramatically from around 50 QPS to 70 QPS.

With moderate values of T, such as 5 or 10, performance is more balanced. Figure 13 shows that Linear-5 and Linear-10 perform well across the spectrum of arrival rates. However, even Linear-10 is too aggressive under high arrival rate, which results in significantly increased response times over sequential for 100 QPS. Our application has fairly high parallelization overheads as shown in Figure 8. Consequently, the more conservative linear approach ($T = 5$) is better than linear with other T parameters.

Adaptive performs better than linear because Adaptive exploits both parallelization efficiency and system load when selecting the degree of parallelism. For example, in 95th-percentile latencies, Adaptive performs 8.5%, 17.5%, 11.5%, and 14.6% better than Linear-5 under 50, 60, 70, and 80 QPS, respectively. The performance gap is wider for higher-percentile values; in 99th-percentile latencies, Adaptive performs 21%, 27.5%, 27.2%, 23% better than Linear-5 under the same arrival rates.

The parallelization efficiency is a good indicator on how aggressively the parallelism degree should decrease with increased load. In contrast, linear uses only system load, and therefore it may be either too conservative or too aggressive. Also, as its performance is sensitive to the selection of the threshold value, if workload changes, one should make it clear how to change the threshold value, which is not a robust way.

7. Related Work

Adaptive parallelism. Many interfaces and associated runtime systems have been proposed that adapt parallel program execution to run-time variability and hardware characteristics [6, 9, 19–21, 25, 30, 35]. They focus on improving the execution of a single job with respect to various performance goals, such as reducing job execution time and improving job energy-efficiency. However, they do not consider a server system running concurrent jobs. Simply applying parallelism to minimize the execution time of every single job will not minimize the mean response time across all jobs. Our work focuses on such an environment with many jobs where the parallelization of one job may affect others. We decide the degree of parallelism for a job based on the impact both on the job itself and on the other jobs.

Sharing resources adaptively among parallel jobs, which is often referred to as adaptive job scheduling, has been studied both empirically [11, 22] and theoretically [4, 15]. However, they focus on a multiprogrammed environment instead of an interactive server with latency constraints. In a multiprogrammed environment, it is common for jobs to have different characteristics that the scheduler does not know *a priori*. Thus, work in this area uses non-clairvoyant scheduling wherein nothing is assumed or known about the job before executing it. The scheduler learns the job's characteristics and adjusts the degree of parallelism as the job executes. In contrast, in the web search server that we study, requests share a lot of similarities because they process user queries using the same procedure. So the scheduler can exploit more information, such as the average execution profile for the requests, to improve scheduling decisions. Moreover, as inter-

active requests often complete quickly, there is limited time for the scheduler to learn and adapt to the characteristics of an individual request. Instead, we use a predictive model that makes the decision before executing a job using the available job information.

Adaptive resource allocation for server systems has been explored; most of this work, however, focus on dynamically allocating resources to different components of the server while the individual requests still execute sequentially [33, 36]. In contrast, Raman *et al.* have proposed the Degree of Parallelism Executive (DoPE), an API and run-time system for adaptive parallelism [26]. The API allows developers to express parallelism options and goals such as maximizing throughput and minimizing mean response time. The run-time has mechanisms to dynamically decide the degree of parallelism to meet the goals. Like this paper, one goal of parallelization in their work is to minimize response time of requests in a server system. They present two algorithms for deciding the degree of parallelism. We implement both algorithms, called *binary* and *linear* in Section 6, and we compare experimentally them to our adaptive algorithm.

Reducing response time in web search. To reduce the latency of semantic web search queries, Frachtenberg applies multithreading to achieve intra-query parallelism [12]. To parallelize a query, the ISN partitions the data into equal-sized subsets of document IDs and each thread works on one subset. This is identical to the static coarse-grained processing approach that is discussed in Section 3. A key assumption of that work [12] is the following: Matching documents of the query in web index are evenly distributed and thus load balancing among threads is not of great concern. However, this is not always the case because matching documents of a query may not be evenly distributed along the index space. Techniques that address load imbalance among threads when intra-query parallelism is exploited have been proposed in [31, 32].

Tsirogiannis *et al.* explore efficient algorithms that partition the inverted indices of a search query to balance the load among threads within a guaranteed error bound [29]. The algorithms further exploit good probing orders on the partitioned indices to reduce the intersection overhead. Their techniques can be combined with ours to enhance the load balancing.

Tatikonda *et al.* propose a fine-grained intra-query parallelism approach in web search [31], which has some similarities to our query parallelization technique. Both approaches partition the indices into a number of fine-grained tasks, which enables good load balancing among threads. Moreover, both maintain a shared data structure among all threads of a query to store the top ranked results. The two approaches have several fundamental differences. First, while they propose to use a producer-consumer model to generate and assign tasks, we employ a decentralized method in which threads coordinate and acquire its next index partition using a single atomic fetch-and-increment operation and the top results are kept in each thread's context. This enables each thread to promptly invoke early termination to reduce the overhead of speculative execution with small synchronization overhead. Moreover, another key focus of our work is to decide the parallelism of a query adaptively based on query execution profile and system load, which is not studied in the prior work [31]. Lastly, we implement our approach and evaluate it using more recent data sets.

Graphics processors (GPU) [10] and SIMD instructions [27] have been used to parallelize the processing of an individual search query. Also, there are studies on reducing the response time for web search queries across different system components, for example, optimizing caching [3, 13] and prefetching [18] to mitigate I/O costs and improving network protocols between ISNs and aggregators [34, 37]. Moreover, response quality can be traded off to reduce response time, especially under heavy load or other exceptional situations in which the servers could not process queries fast enough [8]. These studies are complementary to our work.

8. Conclusions

The quality of results of modern web search needs to keep improving without increasing latency, in order to increase user satisfaction and revenue. However, this is challenging because the servers must search a massive number of web documents to produce high quality search results.

This paper explores effective techniques to parallelize the query execution in web search servers to reduce their mean and high-percentile response time while providing the same response quality as in sequential execution. This paper introduces two techniques: (1) a dynamic fine-grain sharing technique that mimics the sequential order of execution to parallelize each individual request with small speculative execution and good load balancing; and (2) an adaptive algorithm that decides the parallelism degree of each request at runtime using system load and parallelization efficiency. The proposed schemes are prototyped in Microsoft Bing and evaluated experimentally with production workloads. The experimental results show that the mean and 95th-percentile response times for queries are reduced by 50% or more under moderate or low system load, with no degradation in the relevance of the search results.

Adaptive parallelism is a promising approach to reduce response time of various interactive workloads besides web search. Future work will be to apply the adaptive parallelism algorithm to different workloads and resource bottlenecks.

Acknowledgments

We thank our shepherd, Flavio Junqueira, and the anonymous reviewers for their valuable comments and suggestions. We also thank Ashot Geodakov, Chenyu Yan, Daniel Yuan, Fang Liu, Jun Zhao, and Junhua Wang from Microsoft Bing for their great help and support, and we thank James

Larus, Kathryn McKinley and Yi-Min Wang from Microsoft Research for the insightful discussions and feedback.

References

[1] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *SIGIR*, 2001.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53 (4):50–58, Apr. 2010.

[3] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *SIGIR*, 2007.

[4] N. Bansal, K. Dhamdhere, and A. Sinha. Non-clairvoyant scheduling for minimizing mean slowdown. *Algorithmica*, 40 (4):305–318, Sept. 2004.

[5] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, Dec. 2007.

[6] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, C. D. Antonopoulos, and M. Curtis-Maury. Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems. *Parallel Comput.*, 33(10-11):700–719, Nov. 2007.

[7] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *SIGIR*, 1985.

[8] B. B. Cambazoglu, F. P. Junqueira, V. Plachouras, S. Banachowski, B. Cui, S. Lim, and B. Bridge. A refreshing perspective of search engine caching. In *WWW*, 2010.

[9] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *ICS*, 2006.

[10] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance ir query processing. In *WWW*, 2009.

[11] D. Feitelson. *A Survey of Scheduling in Multiprogrammed Parallel Systems*. Research report. IBM T.J. Watson Research Center, 1994.

[12] E. Frachtenberg. Reducing query latencies in web search using fine-grained parallelism. In *WWW*, 2009.

[13] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *WWW*, 2009.

[14] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, Dec. 2008.

[15] Y. He, W.-J. Hsu, and C. E. Leiserson. Provably efficient online nonclairvoyant adaptive scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 19(9):1263–1279, Sept. 2008.

[16] 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.

[17] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *ISCA*, 2010.

[18] S. Jonassen, B. B. Cambazoglu, and F. Silvestri. Prefetching query results and its impact on search engines. In *SIGIR*, 2012.

[19] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for smt multiprocessor architectures. In *PPoPP*, 2005.

[20] W. Ko, M. N. Yankelevsky, D. S. Nikolopoulos, and C. D. Polychronopoulos. Effective cross-platform, multilevel parallelism via dynamic adaptive execution. In *IPDPS*, 2002.

[21] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *ISCA*, 2010.

[22] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 11(2):146–178, May 1993.

[23] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *ISCA*, 2011.

[24] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10):749–764, Sept. 1996.

[25] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *IISWC*, 2011.

[26] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using dope: the degree of parallelism executive. In *PLDI*, 2011.

[27] B. Schlegel, T. Willhalm, and W. Lehner. Fast sorted-set intersection using simd instructions. In *ADMS*, 2011.

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

[29] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *SIGIR*, 2005.

[30] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. In *ASPLOS*, 2008.

[31] S. Tatikonda, B. B. Cambazoglu, and F. P. Junqueira. Posting list intersection on multicore architectures. In *SIGIR*, 2011.

[32] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *Proc. VLDB Endow.*, 2(1): 838–849, Aug. 2009.

[33] G. Upadhyaya, V. S. Pai, and S. P. Midkiff. Expressing and exploiting concurrency in networked applications with aspen. In *PPoPP*, 2007.

[34] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *SIGCOMM*, 2012.

[35] Z. Wang and M. F. O'Boyle. Mapping parallelism to multicores: a machine learning based approach. In *PPoPP*, 2009.

[36] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.

[37] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.