TPC: Target-Driven Parallelism Combining Prediction and Correction to Reduce Tail Latency in Interactive Services

Myeongjae Jeon[†], Yuxiong He[†], Hwanju Kim[*1], Sameh Elnikety[†], Scott Rixner[‡], Alan L. Cox[‡]

[†]Microsoft Research
Redmond, WA, USA

[*]University of Cambridge
Cambridge, UK

[‡]Rice University
Houston, TX, USA

Abstract

In interactive services such as web search, recommendations, games and finance, reducing the tail latency is crucial to provide fast response to every user. Using web search as a driving example, we systematically characterize interactive workload to identify the opportunities and challenges for reducing tail latency. We find that the workload consists of mainly short requests that do not benefit from parallelism, and a few long requests which significantly impact the tail but exhibit high parallelism speedup. This motivates estimating request execution time, using a predictor, to identify long requests and to parallelize them. Prediction, however, is not perfect; a long request mispredicted as short is likely to contribute to the server tail latency, setting a ceiling on the achievable tail latency.

We propose TPC, an approach that combines prediction information judiciously with dynamic correction for inaccurate prediction. Dynamic correction increases parallelism to accelerate a long request that is mispredicted as short. TPC carefully selects the appropriate target latencies based on system load and parallelism efficiency to reduce tail latency.

We implement TPC and several prior approaches to compare them experimentally on a single search server and on a cluster of 40 search servers. The experimental results show that TPC reduces the 99th- and 99.9th-percentile latency by up to 40% compared with the best prior work. Moreover, we evaluate TPC on a finance server, demonstrating its effectiveness on reducing tail latency of interactive services beyond web search.

[1] Currently with EMC

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481. Copyright 2016 held by Owner/Author. Publication Rights Licensed to ACM.

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.
Copyright © 2016 ACM 978-1-4503-4091-5/16/04…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2872362.2872370

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Threads

General Terms Algorithm, Design, Performance

Keywords Interactive Service, Tail Latency, Parallelism, Thread Scheduling, Machine Learning, Web Search

1. Introduction

Interactive online services such as web search, financial trading, and online games provide consistently fast responses to every user request, which translates to requirements on the tail for request response times [9, 19]. Reducing response latency is important because it generates extra revenues: The saved latency can be used for adding extra capabilities and new features. It can also be used for cost savings as the system capacity increases: Under the same latency SLA and input workload, fewer servers are deployed when servers employ the latency reduction techniques to support the higher load. This paper presents an approach to reduce the tail latency of interactive services, where we use web search as a driving example.

For large-scale commercial search engines, the web index is typically partitioned among many servers, and therefore a user request (or query) is processed on several servers [3, 8, 20]. The slowest server determines the request latency. On a single server, it is important to reduce the tail latency rather than the average. We illustrate this point with a simple example of a cluster containing 40 search servers. Here, the request response is aggregated from the 40 servers, and to obtain a 99th-percentile latency of X time units, each server must achieve a substantially stricter tail latency [9, 19], specifically $\sqrt[40]{99/100} \times 100 = 99.98$ th-percentile latency of X.

We characterize the server workload to investigate how to reduce tail latency. The study shows that computation time dominates disk and network IO as well as queueing latencies. CPU is the bottleneck resource, making parallel processing on a multi-core server a promising approach. Moreover, data center environments collect extensive telemetry data, including system load, query logs, and workload in-

formation, which can be exploited to provide better performance.

However, workload characterization reveals important challenges. Queries exhibit large latency variability, where the service demand of long queries is orders of magnitude higher than the median. Moreover, not all queries benefit equally from parallelization: Parallelization has a non-negligible overhead, particularly for short queries. This motivates estimating request execution time, using a predictor, to identify which queries are long. Prediction, however, is not perfect; a long request mispredicted as short is likely to contribute to the server tail latency. In effect, prediction inaccuracy sets a ceiling on the achievable tail latency reduction.

Prior work on using parallel processing to reduce query latency falls into two categories according to whether prediction information is used. (1) Techniques in the first category ignore per-query prediction information completely. For example, recent work [20, 33] determines the parallelism degree before running the query according to system load without differentiating short and long queries, resulting in substantial overhead as short requests are parallelized. Another approach [15] increases the parallelism degree during query execution, but misses the opportunity to parallelize long queries early, leading to longer completion time and impacting the tail latency. (2) Recent work in the second category relies primarily on per-query prediction information [21]. The parallelism degree can be decided according to the predicted query execution time. The latency reduction is, however, limited because the mispredicted long queries impact the tail substantially.

To overcome the limitations of the prior work, we propose a novel approach that exploits prediction information judiciously with dynamic correction for inaccurate prediction. Dynamic correction ramps up the degree of parallelism to accelerate long queries mispredicted as short. Ramping up parallelism must be done carefully: If it is invoked too early, it wastes resources that could have been used to reduce the tail latency. If it is invoked too late, the request completes late directly impacting the tail latency.

Our main contribution is TPC, standing for Target-driven parallelism combining Prediction and Correction, which is an algorithm to reduce the very high tail latency. TPC exploits several factors jointly, including per-query prediction information, instantaneous system load information, statistical workload information (*i.e.*, average parallelism speedup). It has three key insights:

First, TPC determines a common *target* completion time to determine how aggressively to parallelize requests and when to invoke dynamic correction. The target is used for two purposes: (1) to allocate the least amount of resources so that majority of requests complete before the target, and (2) to determine which requests take longer than expected, e.g., due to under-estimating their service demands. The requests taking longer than the target are likely to impact the tail. If additional resources are available, they are allocated to accelerate those requests. The target value depends on the current system load and overall parallelism efficiency of requests.

Second, TPC applies *predictive parallelism*: it predicts the execution time of each request to compute its degree of parallelism. More precisely, TPC selects the smallest parallelism degree so that the request meets its target completion time while accounting for the request parallelism efficiency. As a result, short requests complete within the target using sequential execution and long requests are parallelized to meet the target using minimum resources.

Third, TPC introduces *dynamic correction* to handle mispredictions — long requests that are mispredicted as short and execute sequentially or with a low parallelism degree. TPC increases parallelism for requests that exceed the target completion time while considering the available spare resources, reducing latency for the very high-percentile.

To assess the benefits of TPC, we implement it in the context of Microsoft Bing search engine and evaluate it experimentally using a production workload. The empirical results show that TPC consistently outperforms the prior work [20, 21, 33]. TPC effectively reduces the very high-percentile latency. The server 99.9th-percentile latency is reduced by up to 40% compared with the best prior work. Moreover, as each server completes requests by meeting the computed target resulting in lower latency variance and lower tail latency, at the cluster level tail latency is reduced. Our results show that the 99th-percentile tail latency of a 40-server cluster is reduced by 66% compared with sequential and 29% compared with the best prior work.

Furthermore, we generalize the TPC approach to a wider class of CPU-intensive interactive services exhibiting the properties of highly-variant request demand, parallelizable request execution and predictable sequential execution time. As an example, we implement TPC on a finance server and show its effectiveness on reducing tail latency.

We structure the paper around our contributions: First, we conduct a comprehensive workload characterization that shows the opportunities and challenges of using parallelization to reduce request tail latency (Section 2). Second, we develop a new policy, TPC, that applies prediction information judiciously with dynamic correction under inaccurate prediction (Section 3). Third, we implement and evaluate TPC experimentally in a commercial search engine. We find that TPC consistently outperforms the prior work (Section 4). Finally, we implement and evaluate TPC on a finance server, demonstrating its effectiveness beyond web search (Section 5).

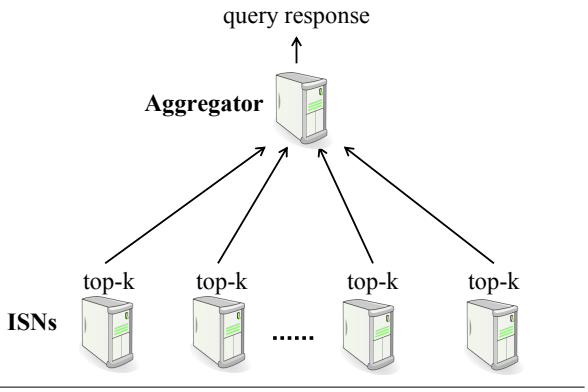


Figure 1. Index serving system architecture.

2. Workload Characterization

The index serving system in a search engine retrieves documents relevant to user queries, generating an interactive workload that we characterize carefully to motivate TPC design.

2.1 System Overview

System architecture. Figure 1 illustrates the *partition-aggregate* architecture of an index serving system. It consists of an aggregator (also known as a broker) and search servers called index serving nodes (ISNs). The index contains information about web documents, is document-sharded [3] and distributed among the ISNs. When a user sends a query and the query response is not cached, the aggregator propagates the query to all ISNs hosting the web index. Each ISN searches its fragment of the web index to return the top-k most relevant results to the aggregator. The aggregator receives the results from the ISNs, and merges them to compute the response to the user query. The aggregator waits for all of its ISNs to respond so it will not miss any search results. ISNs are the workhorse of the index serving system. They constitute over 90% of the total hardware resources and account for the majority of the query processing time. This architecture is similar to others discussed in prior work [3, 8, 20].

Query processing. The ISN manages a number of worker threads that can process several queries concurrently on multiple cores. Newly arrived queries first join the waiting queue of the ISN. When a worker thread is idle, it dequeues a query from the waiting queue and starts to process it. The worker thread searches its web index fragment to produce a list of documents matching the keywords in the query. As there are a fixed number of worker threads, some queries may experience a delay in the waiting queue. Thus, a query's response time consists of both its queueing delay and execution time.

2.2 Computationally Bound Workload

Several factors make web search a computationally bound workload. First, the inverted indices are sharded across a large number of ISNs, each of which has 10s of GBs of DRAM for caching its part of these indices. This caching substantially reduces the amount of disk I/O performed by the ISNs [20]. For example, under production workloads, the average amount of disk I/O by each ISN is only 0.3 KB/s. Consequently, the CPU spends little or no time blocked on disk I/O. Second, a search engine is an interactive application. Therefore, it is not operated at extremely high loads to avoid queueing delay and quality degradation [20]. Our measurements show that even under a relatively high CPU utilization of 73%, the average queueing delay at the ISN is 0.35 ms. Third, the time for network I/O is a small fraction of the overall query latency. The average time for network I/O per query is only 2.13 ms, while the computation by the CPU represents the largest fraction, with a service demand of 13.47 ms on average or up to more than 200 ms. We have observed this small impact of the non-computation parts regardless of the query length. Therefore, query latency reduction is best achieved by speeding up the computation.

Queries running concurrently exhibit little interference or memory contention. [20] reports that a query suffers less than a 5% slowdown while running together with other queries at 50% CPU utilization, as compared to running alone. We have observed similar behavior under our workloads. In summary, we have a computationally bound workload with little interference among queries. So, we believe that *query parallelization has a good chance of reducing latency.*

2.3 Latency Variability

Server requests have varying computational demands. Most queries are short, with more than 85% taking below 15 ms. A few queries are, however, very long, taking up to 200 ms. In particular, the average service demand is 13.47 ms, while the 99th-percentile service demand is 200 ms, which is 15 times the average. The gap between the median and the 99th-percentile is even larger at 56 times. We observe that this variability is fairly consistent across a few hundred ISNs in the index serving cluster.

Many factors affect query execution time, and we discuss two typical factors responsible for long execution. First, long queries tend to have more documents to match and score, which consumes more processor cycles. Second, long queries involve the intersection of inverted indices for a larger number of keywords. It is known that the average latency of queries with ten keywords can be approximately an order of magnitude greater than that of queries with only two keywords [36].

The distributed query processing further increases latency variability. In a partition-aggregate cluster where a search query runs in parallel and its results are aggregated over a large number of ISNs (in Figure 1), speeding up the long queries at each ISN is necessary to reduce the cluster tail latency, which directly affects end-to-end response time. Our experimental evaluation on a Bing search cluster (in Section 4.5) also shows that reducing tail latency of the clus-

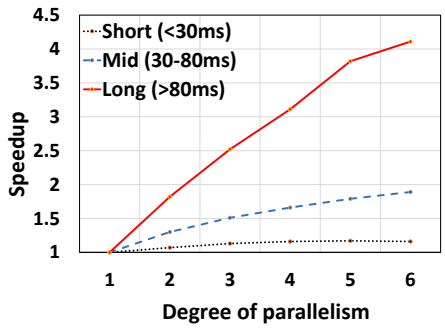


Figure 2. Average speedup for processing queries in parallel grouped by their execution time.

ter requires reducing a much higher percentile at each ISN. Thus, each server should be optimized for both moderate (*e.g.*, 95th- to 99th-percentile) and even high (*e.g.*, higher than 99th-percentile) tail latency. We conclude that *to deliver low-latency responses to users, it is crucial to speed up long queries at each ISN.*

2.4 Query Parallelization Efficiency

Multiple threads can process a single query to exploit the modern multi-core processors. We adopt a state-of-the-art query parallelization approach from prior work [20], where the index data is partitioned into small tasks, forming a task pool, and the query threads retrieve tasks from the pool and process them. A query can be assigned a number of threads upon the start of query execution [20], and later on the scheduler can add additional threads to a query during its execution [15]. As observed in many applications [1, 11, 23, 33, 38], parallelization incurs overhead due to various factors, such as task creation, synchronization and speculated execution [15, 20]. The parallelization efficiency varies among queries for two reasons, which impact on short queries: (1) Some steps are not parallelized, *e.g.*, query parsing and rescoring of the top results. (2) Load imbalance among tasks adversely affects parallelism speedup [12].

We study query parallelization efficiency empirically using a production index and query log. We classify the queries into three classes based on execution time and show the speedup with different parallelism degrees in Figure 2. The long queries that run more than 80 ms achieve more than 4 times speedup on 6 threads, reducing their mean execution time from 168 ms to 41 ms. In contrast, using 6 threads, the short queries that complete in under 30 ms achieve only 1.16 times speedup. In addition, medium queries that run between 30 and 80 ms have modest speedup, around 2 times with parallelism degree 6. The parallelization efficiency thus depends on the query service demand, as has been observed in other search workloads [12]. We conclude that *parallelization is more effective for long queries than short queries.*

2.5 Impact of Prediction Accuracy

Prior work uses machine learning techniques to predict (sequential) execution time of a query before running it to identify long queries [21, 26]. We adopt a predictor [21] that uses both term and query features such as IDF (inverted document frequency) and the number of augmented keywords, and uses a boosted-tree regressor to predict query execution time. Regressor accuracy is commonly measured using L1 error, representing the difference of the predicted and actual query execution time. The L1 error of our predictor is 14 ms using production indices and query logs. To illustrate the impact of prediction accuracy on tail latency, we present a more intuitive view from a classifier's perspective as we show in the experimental evaluation: The predictor identifies long queries, classifying each query as either long or short. Classifier accuracy is measured using precision (the fraction of detections which are truly long queries) and recall (the fraction of truly long queries which are detected).

High recall is essential; We should identify the majority of long queries and subsequently process them faster to reduce their response time. A misprediction due to imperfect recall means that a long query is processed as a short query and therefore is likely to increase tail latency. In contrast, misprediction due to imperfect precision is directly related to how many short queries are mispredicted as long, resulting in higher overhead: Resources are wasted to parallelize short queries with no benefit for reducing tail latency.

Across the 40 ISNs, the average recall is 0.86 and precision is 0.91 for identifying queries running longer than a latency threshold of 80 ms. Since the workload has 4% of long queries ($>$ 80 ms) and the predictor correctly identifies 86% of them, the remaining long queries, which are 0.56% of all queries, are misidentified as short. These mispredicted long queries will not affect the 99th-percentile tail latency (as they are less than 1%) but they significantly affect the higher tail such as 99.9th-percentile (as they constitute more than 0.1% of all queries). Prediction alone cannot effectively reduce any tail latency beyond the 99.44th-percentile.

2.6 Implications for Tail Latency

This workload characterization shows both opportunities and challenges for reducing tail latency. On one hand, parallelization can effectively reduce query execution time particularly for long queries. On the other hand, queries exhibit different parallelization efficiency, and impact the tail differently according to their service demands. Execution time prediction identifies the majority of long queries (accelerating them to reduce tail latency) and identifies the majority of short queries (avoiding their parallelism overhead). However, it is not sufficient to reduce latency in very high percentiles because of the mispredicted long queries.

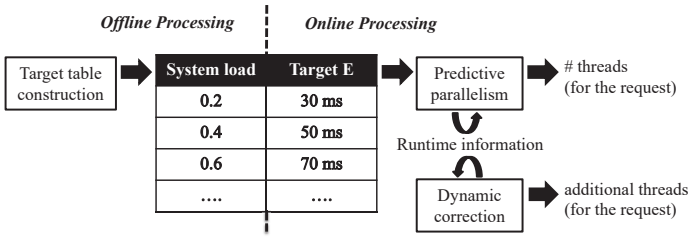


Figure 3. Online/Offline processing workflow in TPC.

3. TPC Algorithm

We propose TPC — Target-driven parallelism combining Prediction and Correction — which exploits the benefits of prediction and uses dynamic parallelism to handle the issues of mispredicted long queries. Ramping up parallelism on mispredicted queries can further reduce tail latencies beyond using prediction alone. However, the challenge is to determine "when" to start ramping up the parallelism dynamically. If we do this too early, we end up wasting resources of parallelizing queries that will not impact the tail; if we do it too late, we end up increasing latency. TPC carefully selects the appropriate *target* latencies based on system load to leverage prediction and dynamic correction to reduce tail latency.

TPC computes a target completion time E for all requests based on current system load. Resources are allocated to a request such that it completes within the target E. Short requests can complete within E using sequential execution, but long requests must be parallelized.

If the predicted sequential execution time is longer than E, parallelism is carefully used to shorten the request completion time to be within E using the minimum resources. Providing extra resources to complete a request earlier than E is not beneficial to reducing the tail, and it takes resources that could have been used by other requests to meet E.

Since prediction is not 100% accurate, long requests mispredicted as short would affect the tail. We develop a dynamic correction mechanism, to ramp up request parallelism and use available idle cores to speed up the requests that have not completed as expected, mitigating the tail.

Figure 3 presents an overview of TPC. As scheduling decisions must be made quickly, we split the scheduling algorithm into two parts: online and offline. The *offline* component constructs a table, called the *target table*, that maps the system load to target completion time E. We develop an efficient algorithm using gradient descent method to search for the appropriate targets. During *online* processing, TPC utilizes the target E from the table according to the current system load, and it determines the parallelism degree of requests through two steps: predictive parallelism and dynamic correction. *Predictive parallelism* decides the parallelism degree before executing the request. It exploits the predicted request execution time and request parallelism efficiency to meet E by using minimum resources. When a request does not complete by the target time E, TPC employs *dynamic correction*, increasing its parallelism at runtime to complete it faster. We next focus on the online component of TPC: predictive parallelism and dynamic correction.

3.1 Predictive Parallelism

We use the predicted request execution time to parallelize the long requests, reducing tail, and execute the short requests sequentially, saving resources.

Inputs. Predictive parallelism takes three inputs to decide request parallelism degree: *(1) Target completion time.* TPC reads the current system load to retrieve the target completion time E from the target table. *(2) Predicted request execution time.* To identify long requests, TPC uses a predictor from prior work [21] to estimate request execution time L before running a request. *(3) Parallelization efficiency.* The degree of parallelism should depend on the parallelization efficiency. To illustrate, consider the case of perfect parallelization efficiency, in which the response time decreases linearly with increased parallelism. Here, we should use as many cores as possible. In contrast, if parallelization efficiency is too low, response time does not decrease with increased parallelism, making sequential execution the best option. Realistic requests fit between these extremes, and better efficiency allows a higher degree of parallelism.

We use a speedup profile to model request parallelization efficiency. The speedup profile maps request parallelism degree to its speedup. We denote it by $\{S_i | i = 1, 2, ...P\}$, where S_i is the request speedup with parallelism degree i and P is the maximum degree. In practice, request speedup is hard to predict accurately. However, since long requests exhibit better speedup than short ones, we classify requests into groups based on their sequential execution time and measure the average speedup in each group, *e.g.*, as shown in Figure 2. Given a request, we use its predicted sequential execution time to determine its group and retrieve the speedup profile.

Algorithm. Upon request arrival, TPC retrieves the target completion time E based on system load, predicts the execution time L of the request, and identifies its speedup profile $\{S_i\}$. With a speedup S_i, the request execution time using parallelism degree i is estimated as $T_i = L/S_i$. *TPC finds the smallest parallelism degree d to meet E*, i.e., $d = \arg\min_{1 \le i \le P}\{T_i | T_i \le \mathrm{E}\}$. Putting extra resources to complete a request much earlier than E is not beneficial to reducing the tail, as it consumes resources that could have been used by other requests to meet their target completion time.

3.2 Dynamic Correction

If a request has not completed within the target E, TPC increases the request's parallelism degree dynamically at runtime to complete the request quickly.

Motivation. In Section 2.5, we demonstrate that using prediction alone cannot reduce latency beyond 99.44th-percentile due to the limited predictor accuracy: There are

0.56% of requests are long but misidentified as short requests. For reducing even higher percentile latency, *e.g.*, 99.9th or higher, dynamic correction is needed to recover prediction errors.

Technique. When a request has not finished its execution by the target E, TPC dynamically increases the request's parallelism degree at runtime to complete the request quickly. In particular, TPC increases the request parallelism degree up to using all the available resources or reaching the maximum parallelism degree of the request. The available resources can be measured in several ways, including number of idle cores (or hardware contexts with SMT), and the number of idle worker threads. In the experiments, TPC uses the number of idle worker threads. Dynamic correction accelerates long requests mispredicted as short, and therefore reduces the very high percentile tail latency that predictive parallelism cannot optimize effectively.

Synergy between Predictive Parallelism and Dynamic Correction. When request execution time is predictable, dynamic correction may not be as efficient as predictive parallelism. First, prediction allows us to parallelize those really long requests at the very beginning of query execution with high parallelism while dynamic correction would defer the decision and prolong their execution time. Moreover, when we identify a request as long, we can parallelize the request early using smaller parallelism degree to meet its target E, whereas dynamic correction parallelizes the request late using a higher parallelism degree. Since the parallelism efficiency decreases with increasing parallelism degree, predictive parallelism completes the request while using fewer resources.

Therefore, TPC uses predictive parallelism to parallelize the majority of long requests effectively using fewer resources, and it applies dynamic correction to only mispredicted long requests when necessary. The two techniques are synergistic in reducing the tail latency.

3.3 Computing the Target Table

Instantaneous load on a server varies over time [29], impacting the availability of idle resources. This motivates the need for adaptation to exploit the currently available idle resources. TPC defines target table to map the current load value to its corresponding target completion time. When load increases, TPC chooses a larger target because the system has fewer spare sources. It parallelizes requests less aggressively and uses spare resources for really long requests only. This section describes a systematic way to construct the target table, which searches the desired targets across loads using gradient descent method. Our algorithm is applicable to a variety of system load metrics, such as request arrival rate, number of active threads, and processor utilization.

We formally represent the target table Φ as a list of (load, target) pairs with m entries, *i.e.*, $\Phi = [(d_0, e_0), ..., (d_i, e_i), ..., (d_{m-1}, e_{m-1})]$. Here (d_i, e_i) denotes a (load, target) pair and the loads are sorted in ascending order, *i.e.*, $d_{i-1} < d_i$. For a given instantaneous load d, TPC chooses a target value $E = e_i$ such that $d_{i-1} < d \leq d_i$.

Algorithm 1 describes the procedure — BUILDTARGETTABLE — that searches for the desired target completion times across loads and builds the target table. Instead of using exhaustive search, which is costly, we develop an efficient heuristic algorithm using greedy gradient descent method. The algorithm takes two inputs: an initial target table and the search step size. For example, when using the number of active threads as load metric, the initial table consists of load values $\{d_i\}$ ranging from $0, 1, 2, ..., \hat{d}, \infty$. Here $\hat{d}$ is the maximum number of active threads observed under the production setting. The entry ∞ maps to any load $d > \hat{d}$. Moreover, we initialize the targets $\{e_i\}$ in the table to a set of small values, *e.g.*, the latency of an unloaded system where every single request is parallelized using all cores, which gives the smallest target we may ever achieve.

BUILDTARGETTABLE searches for the desired target values iteratively using gradient descent method. Starting from the initial table, we increase the target value of each load entry one by one with step size δ (Line 7) and measure the latency impact of a set of new tables experimentally using MEASURETAIL (Line 8). MEASURETAIL is an experimental procedure that takes a target table as input, runs a predefined experiment to cover all production load ranges, and returns a weighted sum of their tail latencies across the load ranges. Among all the new tables $\{tmpTable_i\}$, we identify the one $tmpTable_{i^*}$ that gives the smallest tail latency (Line 10). If this latency is smaller than that of the current table, we update the target table to $tmpTable_{i^*}$ and continue the search (Line 12). Otherwise, if the current table has smaller latency, we stop the search and return the current table as our final target table (Line 15).

The complexity of BUILDTARGETTABLE is measured using the number of times MEASURETAIL is invoked. It is upper bounded by mE_{max}/δ, where E_{max} denotes the maximum latency target. In an interactive system like web search, the value of E_{max} is within a few hundred milliseconds. The step size δ presents a tradeoff between search time and search quality. We use 1ms in our experiments because it is the smallest unit for tail latency measurements. The result of E_{max}/δ bounds the number of iterations we take to complete the search in the while loop (Line 5). Comparing with exhaustive search incurring cost of $(E_{max}/\delta)^m$, our search algorithm is much more efficient. We demonstrate its effectiveness empirically in Section 4.

There are a few remarks related to target table construction. (1) Section 4.6 compares several system load metrics and shows the number of active threads of long queries is a good metric, which we use as default. (2) The target table can be computed periodically as system parameters change. (3) At web search clusters, the workloads are fairly evenly partitioned and well balanced across ISNs. Our experimen-

Algorithm 1 Algorithm for Target Table Construction

procedure BUILDTARGETTABLE(initial target table Φ_0, step size δ)
  $\Phi = \Phi_0$ ▷ initialize target table Φ
  $m = |\Phi|$ ▷ get the number of entries of Φ
  $curLatency$ = MEASURETAIL(Φ)
  while true **do**
    for $i = 0; i < m; i++$ **do**
      $tmpTable_i = [(\Phi.d_0, \Phi.e_0), ..., (\Phi.d_i, \Phi.e_i + \delta), ..., (\Phi.d_{m-1}, \Phi.e_{m-1})]$
      $newLatency_i$ = MEASURETAIL($tmpTable_i$)
    end for
    $i^* = argmin_{0 \le i < m}(newLatency_i)$
    if $newLatency_{i^*} < curLatency$ **then**
      $\Phi = tmpTable_{i^*}$ ▷ update the table
      $curLatency = newLatency_{i^*}$
    else
      return Φ; ▷ found the final target table
    end if
  end while
end procedure

tal results on a cluster of 40 ISNs show that these ISNs have the same target table.

Summary. TPC reduces server tail latency using three insights. (1) It coordinates request completion time using a common target, which is used to identify long requests that are likely to impact the tail, and thus accelerates them through parallelization. (2) It applies predictive parallelism to parallelize the majority of long requests from the beginning of request execution using minimum resources, and to execute short requests sequentially. (3) It exploits dynamic correction to speed up long requests mispredicted as short, further reducing the tail.

4. Experimental Evaluation

We implement TPC along with competing techniques in the index serving nodes (ISNs) of Microsoft Bing, and conduct evaluation using production workload. We show that TPC consistently outperforms prior work across loads in reducing tail latency. In a nutshell, TPC reduces both the 99th-percentile and 99.9th-percentile latency in the ISN by up to 40% compared to the best performing policy (Figure 4 and 5). Moreover, TPC reduces the 99th-percentile latency in a cluster of 40 ISNs by 29% from the best prior work (Figure 8(a)). The saved latency improves user experiences on multiple folds, including providing faster response, enabling extra capabilities such as better ranking, or using a larger web index size.

4.1 Experimental Setup and Methodology

Machine setup and workload. Each ISN machine has two 2.27GHz 6-core Intel Xeon processors with hyperthreading enabled for a total of 24 concurrent hardware threads. The ISN manages a 160 GB web index partition on SSD and uses 17 GB of memory to cache recently accessed web index data. The number of worker threads is set to 28 as a worker thread may occasionally block for disk or network I/O. The Windows OS scheduler dynamically schedules worker threads on the available cores.

Algorithm	Information use		
	Predicted exec. time	System load	Para. efficiency
TPC	✓	✓	✓
AP	✗	✓	✓
Pred	✓	✗	✗
WQ-Linear	✗	✓	✗

Table 1. Information used in parallelism policies.

We conduct experiments both on a single ISN and across a cluster of 40 ISNs. We employ a client that plays queries from a trace of 100K user queries using a Poisson process in an open loop. We vary the load by changing the query arrival rate, *i.e.*, queries per second (QPS). The cluster setup includes an aggregator as illustrated in Figure 1, and Section 4.5 discusses the cluster experiments. All other experiments are conducted using the single-ISN setup.

Policies for comparison. We compare TPC to the following parallelization strategies from prior work. Table 1 compares the information used in choosing the parallelism degree for a query.

- **AP**. Adaptive Parallelism (AP) approach [20] takes into account the average parallelism speedup of all queries and the waiting queue length. The algorithm chooses the parallelism degree for a query such that the total response time of all queries in the system is minimized.
- **Pred**. Pred [21] predicts query execution time using machine learning to parallelize long queries using a *fixed* parallelism degree. All other queries are executed sequentially.
- **WQ-Linear**. Work Queue Linear (WQ-Linear) [33] considers only system load, which is the number of queries waiting in the queue. Queries are parallelized with a degree inversely proportional to the system load.
- **Sequential**. Baseline system with sequential execution.

TPC and Pred use a state-of-art predictor [21], which is memory efficient and achieves high accuracy as explained in Section 2.5. As queries executed across ISNs share common characteristics and features, we build the predictor from one selected ISN and distribute it to all ISNs. Therefore, the cost of building the predictor is constant and not affected by the size of cluster.

We limit the maximum parallelism degree to six in TPC and AP. TPC measures system load using the number of active threads running long queries, and it uses the number of idle worker threads as the available resources for dynamic

correction. To model parallelism efficiency, queries are classified into three groups, short (<30 ms), mid (30–80 ms), and long (>80 ms) as depicted in Figure 2. We discuss other groupings and load metrics in the sensitivity study in Section 4.6.

Performance metrics. We use the *99th-percentile* (P99) and *99.9th-percentile* (P99.9) of query response time as the tail latency metrics. Query response time of the ISN or the cluster is measured from the time it receives the query to the time it responds to the client. We do not report response quality (*i.e.*, relevance scores) because parallelism decisions do not affect the response quality [20].

4.2 Comparison to Competing Techniques

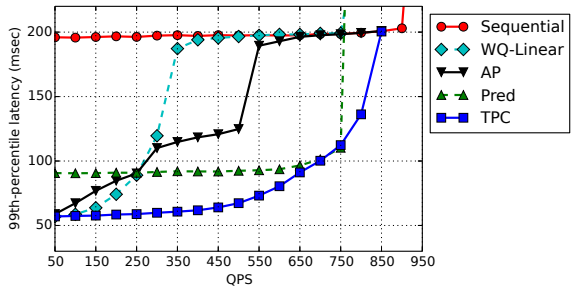


Figure 4. 99th-percentile latency for parallelism policies.

We compare TPC with AP, Pred and WQ-Linear to show the importance of using both query and system load information in TPC. Figure 4 presents P99 latencies of the policies. We vary the load over a wide range from 50 to 900 QPS over the x-axis. For Pred, queries predicted to execute longer than 80 ms run with 3-way parallelism following the reported guidelines [21].

Exploiting predicted query execution time. Figure 4 shows that using per-query information, both TPC and Pred significantly reduce tail latency even at moderate and heavy load. For example, in the range of 500–700 QPS, TPC and Pred achieve approximately 100 ms at P99, while all other approaches have 200 ms or higher P99 latency. TPC and Pred achieve such low tail latencies by parallelizing long queries only; avoiding parallelizing short queries gives more resources to long queries. In contrast, WQ-Linear and AP have high tail latencies since they do not differentiate long and short queries that have different parallelization efficiency and different impact on tail latency. Thus, as load increases, they reduce the parallelism degree for all queries, but short queries may still get parallelized unnecessarily, wasting resources, and long queries are not sufficiently parallelized.

Table 2 shows the query parallelism degree distribution for TPC and AP at low (150 QPS) and high (600 QPS) load. Due to high prediction accuracy, TPC is able to parallelize long queries with high degrees in both loads. At 150 QPS, 98.1% of long queries exploit the maximum degree 6. While

QPS	Policy	Group	1T	2T	3T	4T	5T	6T
150	TPC	Short	93.7	0.8	0.5	0.5	0.3	4.2
		Long	0	0	0.4	0.5	1.0	98.1
	AP	Short	0	1.5	30.5	43.9	15.1	9.0
		Long	0	1.5	29.8	44.9	14.4	9.4
	Pred	Short	98.1	0	1.9	0	0	0
		Long	18.6	0	81.4	0	0	0
600	TPC	Short	96.8	0.6	0.4	0.3	0.2	1.7
		Long	0.3	2.5	5.5	8.4	10.3	73.0
	AP	Short	57.0	42.1	0.9	0	0	0
		Long	55.5	43.3	1.2	0	0	0
	Pred	Short	98.1	0	1.9	0	0	0
		Long	18.6	0	81.4	0	0	0

Table 2. Parallelism degree distribution by percentages.

the ratio decreases to 73% at high load, up to 91.7% of long queries are parallelized with fairly high degrees (>3). Moreover, short queries are rarely parallelized. In contrast, AP gives short and long queries the same parallelism. Thus, long queries receive fewer parallelism degree and less resources compared with TPC. The situation gets worse at higher load: At 600 QPS, 98.8% of long queries are executed with 1 or 2 threads.

Exploiting system load. TPC and Pred have similar tail latency under heavy load, but TPC reduces tail latency from 100 ms of Pred to 60 ms under low to moderate load. The reason is that Pred does not adapt to varying system load and it always parallelizes long queries using the same parallelism degree. In comparison, TPC has more flexibility; it decides which queries to parallelize and by how much according to system load.

We use Table 2 to compare TPC to Pred in detail. At 150 QPS, TPC selects a higher degree for long queries than Pred. Interestingly, even those long queries that are mispredicted as medium or short are often given high degree if idle cores are available. Moreover, unlike Pred, TPC can select any degree among the possible options for better performance. For example, at 600 QPS, although TPC still prefers high parallelism degree for long queries, it exploits all parallel degrees to execute long queries, adapting to transient overload or underload. This adaptation allows TPC to perform better than Pred, which uses fixed parallelism for long queries.

In summary, these results show that it is essential to use both predicted query execution time and adapting to system load to reduce tail latency.

4.3 Using Predictive Parallelism Alone

We discuss if predictive parallelism alone is adequate. Figure 5 compares TPC with the prior policies with respect to a higher-percentile latency, *i.e.*, P99.9. Optimizing for the higher tail at each ISN is important because to reduce tail latency at the aggregator, we need to reduce a much higher percentile at each ISN (further details in Section 4.5). Figure 5 shows that TPC provides the lowest P99.9 latency, outperforming all other policies. Under moderate and high

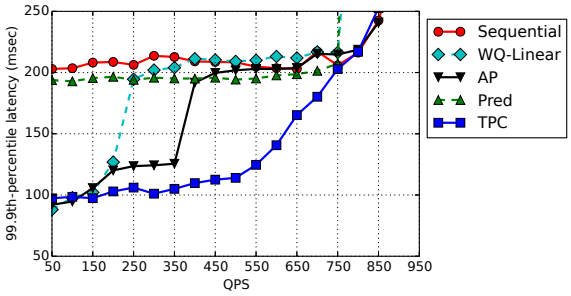
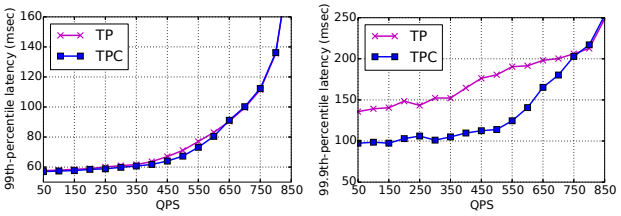


Figure 5. 99.9th-percentile latency for parallelism policies.

load (>400 QPS), TPC reduces the P99.9 latency by up to 40% compared with the best results from prior work. More specifically, the relative latency gap from AP and WQ-linear to TPC is similar to that for the P99 latency (Figure 4). Pred, however, performs poorly at P99.9, almost as high as Sequential, for all the loads, although Pred performs reasonably well for P99 latency.

The primary reason for a big difference on the performance of Pred with respect to the P99 and P99.9 latency is the limited accuracy of the predictor. As discussed in Section 2.5, assuming a tail latency target of 80 ms, which is set for Pred, some long queries are misidentified as short, and they constitute 0.56% of total queries. These queries do not affect P99 (as they are less than 1%) but P99.9 (as they are more than 0.1%). This is why Pred, which uses prediction only, performs poorly for the higher percentile latency. In contrast, TPC uses dynamic correction to recover from prediction errors, effectively reducing the higher tail latency.



(a) 99th-percentile latency (b) 99.9th-percentile latency

Figure 6. Tail latency for TP and TPC.

To measure the contribution of dynamic correction at TPC, we compare it with TP, which is the same as TPC but does not employ dynamic correction. Figure 6(a) and 6(b) show the P99 and P99.9 latency, respectively, TPC and TP. As expected, for P99 latency, they perform almost the same since prediction is accurate enough to reduce P99 range. However, TPC is better in P99.9 latency. As Figure 6(b) shows, dynamic correction reduces P99.9 by 40–65 ms more than TP. The analysis on the query parallelism degree distribution shows that adding dynamic correction increases the percentage of the long queries parallelized with high degrees (> 3) from 95% to 99.5% at 150 QPS and from 79.2% to 91.7% at 600 QPS.

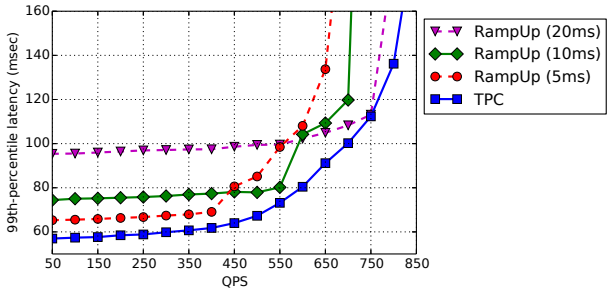


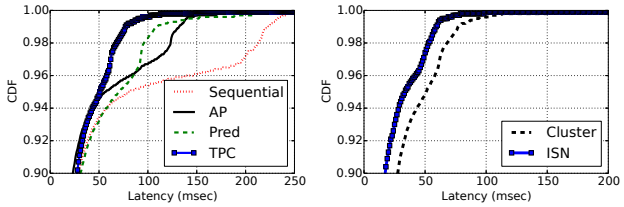
Figure 7. 99th-percentile latency of TPC and RampUp.

4.4 Using Dynamic Correction Alone

This section compares TPC with the policies that employ dynamic correction without prediction, in order to study how dynamic correction alone performs. We introduce policies that increase the query parallelism degree dynamically during its execution, which we call RampUp. RampUp starts a query with sequential execution. Then, if the query does not complete in a predefined time interval, it increases the query parallelism degree by 1 until the query either completes or reaches its maximum parallelism degree (which is set to six). Using RampUp, short queries complete sequentially, while long queries get higher parallelism and more resources. It achieves the goal of parallelizing long queries without knowing query execution time. We use different thread ramp-up intervals (5, 10, and 20 ms) to evaluate RampUp. For example, an interval of 5 ms means that we add a thread to the query at each 5 ms of execution up to 6 threads per query. The smaller the interval is, the higher the parallelism degree the query receives.

Figure 7 compares TPC with RampUp for P99 and shows that TPC achieves lower latency than RampUp for a large range of loads. TPC outperforms RampUp because it accurately predicts query execution time, making parallelism decision early before executing a long query, whereas RampUp intrinsically has delay in increasing parallelism, resulting in higher latency. To reduce the delay, RampUp has to add up parallelism quickly (*e.g.*, at every 5 ms interval). It effectively reduces tail latency at light load, but this comes with high parallelism overhead at high load.

Even when the RampUp policy takes load into account, *i.e.*, using the best RampUp interval at any given load, the latency is still higher than TPC. The reason is that as long as the RampUp interval is not 0 (which is the case except very light load), it cannot parallelize the long queries from the very beginning and it takes longer to complete those queries than TPC. Compared to using RampUp with wide range of intervals, TPC outperforms the best RampUp strategy at every arrival rate across a large load range. For extremely lightly (or heavily) loaded systems, both RampUp and TPC execute queries with maximum parallelism (or sequentially).



(a) Aggregator latency (b) ISN v.s. aggregator for TPC

Figure 8. CDF of latency for a cluster of 40 ISNs at 300 QPS (zoomed to high cumulative fractions).

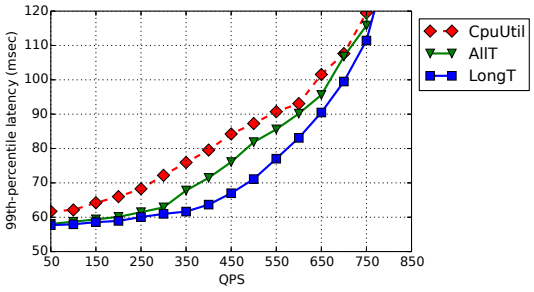


Figure 9. 99th-percentile latency of TPC for load metrics.

Both Section 4.3 and 4.4 show that using either predictive parallelism or dynamic parallelism alone is not sufficient. TPC integrates them to effectively lower tail latency.

4.5 Tail Latency in Cluster of ISN machines

This section discusses how TPC improves query responsiveness in a cluster of 40 ISNs. We compare four strategies (Sequential, AP, Pred, and TPC) and measure response times at an aggregator. For each query, the aggregator waits for all ISNs to collect and merge the results. The aggregator latency is thus decided by the slowest response from the ISNs. Latency at the aggregator includes network, I/O, and queueing delay as well as query processing at the ISNs.

Figure 8(a) shows the cumulative distribution of query response times from the aggregator at 300 QPS, where Y-axis depicts high cumulative probability fractions to focus on slowest query responses. The results show that TPC outperforms all other policies consistently over a wide range of tail latency targets, from P95 onwards, leading to much fewer slow responses at the aggregator. In particular, TPC has less than 0.4% of queries taking longer than 100 ms, whereas AP and Pred have 3.3% and 1.7%. Therefore TPC is the only policy that is able to lower P99 aggregator latency below 100 ms — P99 latency of AP, Pred, and TPC are 132.2, 108.9, and 77.7 ms, respectively. TPC reduces P99 aggregator latency by 29% compared with the best performing approach from prior work.

Next, Figure 8(b) compares cumulative distributions for the latency of TPC at the aggregator and an individual ISN. As the figure shows, in order to reduce tail latency at the aggregator, we need to reduce latency of higher tail at the ISN. For example, the latency of 77.7 ms, which we observe in P99 at the aggregator, appears at the P99.8 at the ISN. This indicates that taming P99 aggregator latency requires reducing P99.8 ISN latency, which is higher tail.

4.6 Sensitivity Study

System load metric. Figure 9 compares various metrics for identifying instantaneous system load. We show the results of our baseline (using the number of active long threads (LongT)), and two alternatives (using CPU utilization (CpuUtil) and the total number of active threads (AllT)). The server retrieves CpuUtil by sending a query to Windows performance counters periodically, for which we use Performance Data Helper (PDH) APIs [30]. The intervals are set to 25 ms in our system considering a tradeoff between more accurate information and lower overhead.

Because CpuUtil is a moving average heavily weighted from the past, Figure 9 shows that it performs worse than other two metrics in capturing instantaneous system load. The performance of CpuUtil becomes worse with increased load, as more queries are aggregated in a sampling interval. For example, under 100 QPS, the utilization for 3 past queries may be aggregated. However, under 500 QPS, that for 13 past queries may be aggregated, more likely misrepresenting the instantaneous load.

In contrast, accounting for the number of active threads is a good proxy to measure instantaneous system load. In particular, LongT is the best because threads running long queries are more likely to stay longer in the system, affecting the resource availability of the newly scheduled query, while short queries are transient and could have completed right after the new queries start.

Prediction accuracy. Dynamic correction in TPC recovers prediction errors and helps cover the performance gap compared with a perfect predictor. We mimic a perfect predictor by using the sequential execution time collected in advance for each input query. We compare TPC using its current predictor with using the perfect predictor, and the results show that they perform rather similarly. More specifically, at P99, the performance difference between using the current predictor and the perfect predictor is only 4.0% on average across all loads. For much higher percentiles, dynamic correction helps to recover from prediction errors and reduce the performance gap. For example, their performance difference in P99.9 is 7.8% on average across the loads. This is a significant improvement over TP (*i.e.*, TPC without dynamic correction): TP has 44.1% higher latency than using the perfect predictor. These results show that dynamic correction effectively compensates prediction error, making TPC more resilient to inaccurate predictors.

The number of parallelism efficiency groups. With more groups, predictive parallelism can potentially apply more accurate speedup efficiency information given query execution time. For example, by grouping queries into long,

mid, and short separately, we obtain more latency reduction than treating all queries indifferently. However, when we increase from 3 groups to 6 groups (we obtain 6 groups by evenly dividing each of the 3 groups into 2 subgroups), no further improvement exists — the most improvement we observe across loads is 0.65%. This is because the speedup profiles of queries among neighboring groups have become similar when we divide the groups even further. It supports why our algorithms use parallelism efficiency profiles of 3 groups.

5. Applicability Beyond Web Search

This section summarizes the properties of the interactive workloads where TPC can effectively reduce tail latency. We show that another application — a finance server — satisfies these properties, and TPC reduces its P99 by 52% compared with sequential execution and by 20% compared to the best prior work.

TPC preferred workload properties. (1) CPU processing dominates the latency, (2) Request service demands exhibit high variability, (3) Requests are parallelizable and the parallelism degree can vary at runtime, and (4) Per-request execution time can be estimated before running the request.

Discussion. Several interactive services are often computationally intensive [14, 27, 34]. Increasing the degree of parallelism is supported by several threading frameworks and runtimes, such as Cilk Plus [5], TBB [7], and TPL [25]. If the request service demand is constant, tail latency becomes lower and there is little opportunity to differentiate processing among requests. On the other hand, the higher the variability, the higher the tail latency, leading to bigger benefits from TPC. Request service demand can be known or estimated in several settings [17, 41].

5.1 Evaluation Using a Finance Server

Banks and fund management companies evaluate thousands of financial derivatives everyday, submitting requests that value derivatives and use the results to make immediate trading decisions. Finance servers are interactive and reducing their tail latency is crucial because slower responses are lost investment opportunities. We use Intel TBB runtime to parallelize request execution and evaluate TPC on finance servers.

Workload. We implement an option pricing server that uses Monte Carlo methods for complex path-dependent Asian options. Request processing is CPU-bound, has a regular structure, and consists of iterations. Request sequential execution time can be estimated accurately (as a function of the structure size and number of iterations). To parallelize a request, we fork a number of tasks at each iteration matching the parallelism degree and join them after the computation.

Experimental setup. For evaluation, input requests contain 10% long requests, for which the service demand is 9 times that of a short request, and requests are issued following a Poisson distribution. We vary the load by controlling the mean inter-arrival rate of requests, *i.e.*, requests per second (RPS). We compare TPC to two parallelization policies, AP and Pred, and to Sequential for reference. In both TPC and AP, the maximum degree of parallelism is four. Pred does not adjust to the load, and we use its best setting with parallelism degree two for long requests.

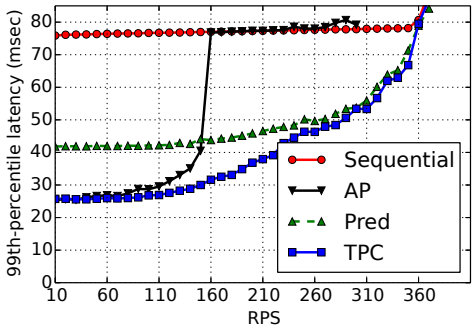


Figure 10. 99th-percentile latency for finance server.

Experimental results. TPC consistently outperforms the three other policies over a wide range of loads as depicted in Figure 10, which shows P99 latency. At light and moderate load, TPC reduces the tail latency over Pred by up to 40% because TPC exploits system load and uses the available resources more aggressively in speeding up long requests. At high load, TPC reduces the tail latency over AP by up to 50% because TPC exploits per-request predicted execution time to parallelize long requests only.

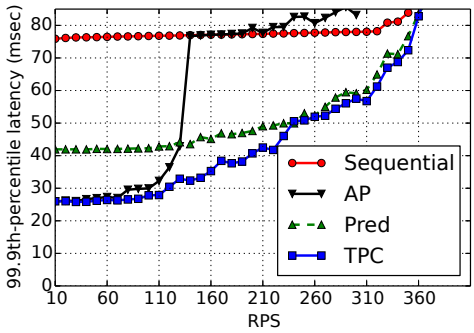


Figure 11. 99.9th-percentile latency for finance server.

Figure 11 depicts P99.9 latency, which has a similar trend as P99. While web search requires dynamic correction to reduce P99.9 to recover from prediction errors, the finance server never calls dynamic correction because the estimated stencil computation time is accurate. At 200 RPS, with TPC, there are on average 3.5 concurrent requests in the system competing for server resources. At this load, TPC runs short requests sequentially and long requests with degree four resulting in short tail latency (P99=37 ms, P99.9=41 ms). In contrast, AP (which does not exploit the request service demand) runs all requests with average degree 3.9, resulting in

high tail latency (P99=77 ms, P99.9=79 ms) as extra CPU resources are needed to parallelize short requests. Pred (which does not use system load to adapt the parallelism degree) runs short requests sequentially but executes long request with degree two rather than a higher degree, contributing to higher tail latency (P99=46 ms, P99.9=48 ms).

6. Related Work

Section 4 compares TPC with several recent algorithms that decide request parallelism degree on interactive server systems, and we do not repeat them here. Prior work shows how to adapt parallel program execution to runtime variability and hardware characteristics [4, 16, 24, 28, 31] either to improve performance of a single program or to reduce average latency on a multiprogrammed environment where job characteristics are unknown a priori. This work addresses a different problem, and we focus on interactive server systems such as web search.

Search query parallelization. Frachtenberg [12] proposes a heuristic to predict which queries to parallelize based on runtime information. Query first runs sequentially for a subset of the index partition, and the ratio of hits to documents is determined. If the ratio is above a threshold, the query is assumed to have good parallelization speedup and is then parallelized; otherwise, the query runs sequentially. Haque *et al.* propose few-to-many incremental parallelism, which dynamically increases parallelism to reduce tail latency [15]. This approach is similar to using load-aware RampUp but without prediction. Compared with TPC, long queries under the above two approaches do not get the resources to speed up their execution at the earliest possible time. It thus has a similar limitation with RampUp policy in Section 4, and we show that TPC provides lower tail latencies by combining prediction with dynamic correction. Moreover, their work uses the total number of instantaneous queries as their load metric, which is coarse-grained. We propose a more general table construction method that allows various (coarse- and fine-grained) load metrics, enabling additional latency savings (Figure 9). Tatikonda *et al.* propose a fine-grained intra-query parallelism approach [36], which has some similarities to how we parallelize a query. However, this paper does not discuss how to decide the parallelism degree, which is the focus of this work.

Predicting query execution time for search engine. Macdonald *et al.* [26] propose a framework to predict execution time of web search queries under early termination. Jeon *et al.* [21] improve upon the prior query predictor [26] by incorporating more features, query rewriting, and boosted-tree regressor, resulting in better accuracy. Our work uses the predictor proposed in [21] to predict query execution time.

Other techniques for reducing response times in interactive services. In web search, graphics processors (GPU) [10], SIMD instructions [35], and core frequency scaling [18] have been used to accelerate the processing of a query. In particular, core frequency scaling offers a complementary acceleration mechanism to reduce tail latency [18]. Parallelism can achieve additional speedup beyond a single core, and this potential speedup comes with a key challenge: competition among concurrent queries for cores, which we address in this work.

There are studies on reducing the response time for web search queries across different system components, for example, optimizing caching [2, 13] and prefetching [22] to mitigate I/O costs and improving network protocols between ISNs and aggregators [37, 40]. 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 [6]. For the case where other sources of variability, such as interference from other workloads and hardware variability, contribute to tail latency, Dean *et al.* [9] propose to submit multiple copies of a query to different replicas, and cancel the slower query. These studies are complementary to our work.

Today's data analytics clusters have been optimized to offer timely responses for parallel jobs [32, 39]. Compared to web search, their framework is fundamentally different: Schedulers are often decentralized and pick a set of machines to serve a scheduled job. They reduce latency by sampling the machine status and improving the assignment of jobs to machines.

7. Conclusions

Interactive services are designed to reduce the tail latency of user requests. We introduce TPC, a new algorithm that determines request parallelism degree by combining predictive parallelism and dynamic correction. We evaluate TPC experimentally on both web search and finance servers. The results show that it significantly reduces the tail latency compared with prior work.

Acknowledgments

We thank anonymous reviewers for their valuable comments and suggestions. We also thank Chenyu Yan, Fang Liu, Jun Zhao, and Junhua Wang from Microsoft Bing for their collaboration and support during this work.

References

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *VLDB*, 5(10):1064–1075, June 2012.

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

[3] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, Mar. 2003.

[4] 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.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPOPP*, 1995.

[6] 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.

[7] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *IISWC*, 2008.

[8] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM*, 2009.

[9] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

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

[11] S. Eyerman and L. Eeckhout. The benefit of smt in the multi-core era: Flexibility towards degrees of thread-level parallelism. In *ASPLOS*, 2014.

[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] R. Guida. Parallelizing a computationally intensive financial r application with zircon technology. In *The R User Conference*, 2010.

[15] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ASPLOS*, 2015.

[16] 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.

[17] Y. He, S. Elnikety, and H. Sun. Tians scheduling: Using partial processing in best-effort applications. In *ICDCS*, 2011.

[18] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *HPCA*, 2015.

[19] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *SIGCOMM '13*, 2013.

[20] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Adaptive parallelism for web search. In *EuroSys '13*, 2013.

[21] M. Jeon, S. Kim, S.-W. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: Taming tail latencies in web search. In *SIGIR*, 2014.

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

[23] J. Kwon, K.-W. Kim, S. Paik, J. Lee, and C.-G. Lee. Multicore scheduling of parallel real-time tasks with multiple parallelization options. In *RTAS*, 2015.

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

[25] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA*, 2009.

[26] C. Macdonald, N. Tonellotto, and I. Ounis. Learning to predict response times for online query scheduling. In *SIGIR*, 2012.

[27] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, 2011.

[28] 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.

[29] 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.

[30] MSDN. Using the pdh functions to consume counter data. `http://msdn.microsoft.com/en-us/library/windows/desktop/aa373214(v=vs.85).aspx`.

[31] S. C. Muller, G. Alonso, A. Amara, and A. Csillaghy. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *OSDI*, 2014.

[32] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[33] 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.

[34] S. Ren, Y. He, S. Elnikety, and K. S. McKinley. Exploiting processor heterogeneity in interactive services. In *ICAC*, 2013.

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

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

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

[38] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *ASPLOS*, 2011.

[39] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *OSDI*, 2014.

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

[41] Y. Zhu and V. J. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *HPCA*, 2013.