Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search

Saehoon Kim*
POSTECH

Yuxiong He
Microsoft Research

Seung-won Hwang
POSTECH

Sameh Elnikety
Microsoft Research

Seungjin Choi
POSTECH

ABSTRACT

A commercial web search engine shards its index among many servers, and therefore the response time of a search query is dominated by the slowest server that processes the query. Prior approaches target improving responsiveness by reducing the tail latency of an individual search server. They predict query execution time, and if a query is predicted to be long-running, it runs in parallel, otherwise it runs sequentially. These approaches are, however, not accurate enough for reducing a high tail latency when responses are aggregated from many servers because this requires each server to reduce a substantially higher tail latency (e.g., the 99.99th-percentile), which we call extreme tail latency.

We propose a prediction framework to reduce the extreme tail latency of search servers. The framework has a unique set of characteristics to predict long-running queries with high recall and improved precision. Specifically, prediction is delayed by a short duration to allow many short-running queries to complete without parallelization, and to allow the predictor to collect a set of dynamic features using runtime information. These features estimate query execution time with high accuracy. We also use them to estimate the prediction errors to override an uncertain prediction by selectively accelerating the query for a higher recall.

We evaluate the proposed prediction framework to improve search engine performance in two scenarios using a simulation study: (1) query parallelization on a multicore processor, and (2) query scheduling on a heterogeneous processor. The results show that, for both scenarios, the proposed framework is effective in reducing the extreme tail latency compared to a start-of-the-art predictor because of its higher recall, and it improves server throughput by more than 70% because of its improved precision.

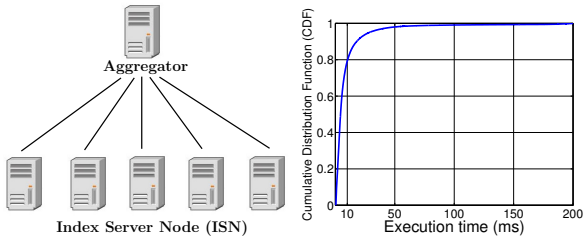
Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Search process

*This work was done during Microsoft Research internship.

WSDM'15, February 2–6, 2015, Shanghai, China.

http://dx.doi.org/10.1145/2684822.2685289.



(a) Search architecture (b) Execution time CDF

Figure 1: (a) Web search engine architecture. (b) Execution time distribution on Bing ISN.

Keywords

Tail Latency; Prediction; Parallelization; Search Engine

1. INTRODUCTION

The primary objective for a web search engine is to deliver relevant web documents to its users with consistently low response times. A long response time degrades user satisfaction and reduces revenues [26]. In particular, commercial search engines optimize for reducing the high-percentile (*e.g.,* the 99th-percentile) response time, which is called the tail latency. A short tail latency offers consistently fast responses to users, which is more important than the mean response time in this context [9].

A web search engine partitions its large index across many index server nodes (ISN). A search query runs concurrently on the ISNs: Each ISN searches for best matches on its local partition, and the results are aggregated, forming the query results. In such a partition-aggregate scheme as depicted in Figure 1(a), the response time at the aggregator is determined by the slowest ISN machine: A long latency at any ISN manifests as a slow query response [9].

To reduce the aggregator tail latency, we need to reduce a much higher tail at each ISN. For example, assuming the aggregator has 40 ISNs, if we want to process 99% of user queries within 120 ms, then each ISN needs to reply within 120 ms with probability around 0.9999. In other words, for the aggregator to have 120 ms at the 99th-percentile response time, the response time of each ISN must be at most 120 ms at the 99.99th-percentile. This motivates our goal of taming the extreme tail latency, namely very high-percentile (e.g., 99.99th-percentile) latency, at ISNs.

Reducing the extreme tail latency is challenging because the workload exhibits high variability. In Figure 1(b), we

measure query execution time using a production log of user queries played on an ISN running a commercial web search engine, Microsoft Bing. We find that most queries are short-running: More than 80% take less than 10 ms. However, few queries are long-running taking more than 200 ms. In particular, the average execution time is 9.1 ms while the 99.99th-percentile execution time is 202 ms, which is 22 times the average. Therefore, to reduce extreme tail latency, it is important to accelerate almost all long-running queries. A long-running query that is not accelerated appears in that tail latency. For a target of 120 ms at the 99.99th-percentile response time, for every 10,000 queries we can miss only one long-running query!

There are a few proposals to accelerate query processing by parallelization [15]. However, it is too expensive to parallelize all queries: Executing a query in parallel consumes more resources than sequential execution, motivating the need to predict the long-running queries. Recent approaches [16, 20] predict query execution time using query and index features. The prediction is used to process long-running queries in parallel and to run short-running queries sequentially [16]. However, these approaches cannot meet our target accuracy — according to our workload, to reduce the 99.99th-percentile latency, we need a predictor with target recall of 98.9% to correctly identify most of the long-running queries, and with good precision to reduce the overhead of accelerating the short-running queries misidentified as long-running.

In this paper we develop and evaluate a prediction framework for reducing the extreme tail latency. The prediction framework identifies the long-running queries, and achieves high recall with good precision by delaying prediction to exploit runtime information. First, since most queries are short-running, delaying prediction (e.g., by 10 ms) enables many short-running queries complete without ineffective acceleration. Second, and more importantly, as the query executes during this short duration, we observe the runtime behavior and identify a new set of features, which we call dynamic features, to improve prediction accuracy of estimated time. Finally, we predict estimated prediction error [4] to override an uncertain prediction by selectively accelerating the query for a higher recall.

We call our predictor *DDS*, which stands for **D**elayed, **D**ynamic, and **S**elective prediction. The framework has three characteristics: (1) Delaying the prediction, (2) exploiting Dynamic features, and (3) Selectively accelerating a query based on its predicted execution time and estimated prediction error. DDS achieves our target recall of 98.9%, while improving precision by 957% compared to a state-of-the-art predictor [16] that uses static features only.

We evaluate *DDS* in two scenarios and use simulation to assess its benefits:

- *Query parallelization on a multicore server.* We parallelize long-running queries to reduce tail latency and run short-running queries sequentially to avoid their parallelization overhead.

- *Query scheduling on a heterogeneous server.* We exploit the emerging heterogeneous processor design which combines energy efficient slow cores with high-performance fast cores. We execute the long-running queries on fast cores to reduce the tail latency and execute short-running queries on slow cores to improve throughput.

Our experiments compare *DDS* with two baselines: (1) a no-prediction baseline, which accelerates all queries in order to meet the desired latency, and (2) a state-of-the-art predictor, which we call *S*, that uses static features [16].

Our results show that, for both scenarios, *DDS* reduces tail latency for a wide range of load at the ISN and at the aggregator, while the baselines fail to meet the latency target at moderate and heavy load. To meet the desired 99.99th-percentile response time of 120 ms, the no-prediction approach needs to accelerate all queries, and *S* accelerates 83.5% of queries. In contrast, *DDS* accelerates only 7.9% of all queries because of its improved precision. As *DDS* significantly reduces the number of queries to accelerate, sparing their corresponding overhead, it increases system throughput by 74% from less than 390 QPS (queries per second) of baselines to more than 680 QPS, potentially saving more than 43% of servers required to process the same workload. As commercial search engines use thousands of servers, *DDS* presents an opportunity for significant server cost savings.

The main contributions of this paper are the following:

- We propose *DDS*, a prediction framework that exploits delayed prediction, dynamic query features, and selective acceleration (Section 3).

- We train and evaluate *DDS* using production query log and web index. At the target 98.9% recall, it shows 957% improvement in precision compared with the prior work using static features (Section 4).

- We assess the benefits of using *DDS* in a simulation study of two query acceleration techniques: query parallelization on a multicore server and query scheduling on a heterogeneous server. *DDS* effectively reduces extreme tail latency at moderate and heavy server load, and significantly increases throughput by 74% compared to prior work (Section 5).

2. BACKGROUND

In this section, we briefly describe the architecture and query processing of a web search engine.

Modern search engines, including Bing, consist of aggregators and multiple index server nodes (ISNs) as shown in Figure 1(a). Billions of documents are partitioned into hundreds of disjoint subsets, each of which is served by a single ISN. For higher efficiency, the subset of documents are constructed as an inverted index, where documents are sorted using a static score such as PageRank. When query is received, an aggregator broadcasts it to multiple ISNs if the results are not cached. A query traverses posting lists of each ISN to match documents, and computes the relevance scores of the matched documents. The aggregator collects the top-k documents for each ISN, and re-ranks the returned documents for the final list [6]. For the entire query processing, the main index of Bing follows a Document-at-a-time (DAAT) [5] strategy, where all posting lists are intersected for a multi-term query and a matched document is scored before exploring the next matched document.

The ISN is a multi-threaded server, capable of processing several queries concurrently for higher efficiency. Queries sent from the aggregator first join the waiting queue of the

Table 1: Static features in previous work [16].

Category	Feature	Description
term feature	IDF	inverse document frequency
query feature	English	Query in English or not (binary)
	NumAugTerm	# augmented requirements
	Complexity	Degree of query complexity
	RelaxCount	Relax count applied or not (binary)
	NumBefore	# terms in the original query
	NumAfter	# terms after query rewriting

Table 2: Dynamic features.

Feature name	Short Description
MaxDynScore	Maximum of dynamic scores
MinDynScore	Minimum of dynamic scores
SumDynScore	Summation of dynamic scores
AvgDynScore	Average of dynamic scores
VarDynScore	Variance of dynamic scores
NumCurrentMatchDoc	# matched documents
NumEstMatchDoc	Estimated # matched documents
FirstMatchingTime	First match time
AvgMatchingTime	Averaging match time

ISN. When a worker thread becomes available, it starts executing a query from the head of the waiting queue. The query response time consists of waiting time and execution time. The waiting time is the amount of time a query spends in the waiting queue before query processing, which is highly related to system load (i.e., the number of queries per seconds). Execution time is the amount of time to find the top-k relevant documents for a query, which is a major portion of response time unless system load is very high.

Dynamic pruning strategies [5, 29] have been proposed to reduce query processing time. While traversing the posting lists, the matched documents are dynamically scored by a complex scoring function with inputs such as the document static score, the number of query terms occurred in a document, location of query, among others. Since this score is obtainable only when a query is executed, we call it *dynamic score*. To terminate query processing, the dynamic score of future matches is estimated. If the estimated dynamic score is sufficiently low compared to the current best documents, the future matches are unlikely to supersede the current matches. Query processing is early-terminated and the top-k matched documents are sent to the aggregator.

3. PREDICTION FRAMEWORK

This section develops a prediction framework, designed to accurately identify long-running queries to reduce tail latency. To meet the goal, we first derive the accuracy requirements of the predictor. We next introduce the *DDS* prediction framework, with delayed, dynamic, and selective prediction, and discuss the design choices.

3.1 Requirements

We derive the accuracy and performance requirements to meet the 99.99th target percentile response time, in terms of precision and recall defined as follows:

$$\text{precision} = \frac{|A \cap P|}{|P|} \quad \text{recall} \quad = \quad \frac{|A \cap P|}{|A|}, \qquad (1)$$

where A is a set of true long-running queries, P is a set of predicted long-running queries. Precision means how many queries are predicted as long-running and recall means how many long-running queries are correctly identified. We define a query as true long-running if its execution time is greater than a given value T (e.g., $T = 100$ ms), which is determined by SLA of the web search engine. To meet a tail latency of 120 ms, we choose $T = 100$ ms, i.e., we want to accelerate queries with execution time greater than 100 ms. We do not choose $T = 120$ ms because queries can incur waiting times, which also increase query latency.

We illustrate how to compute the recall target by analyzing the query workload. Using Bing query log, we find that 0.92% of queries are long-running (their execution time is greater than $T = 100$). Therefore, recall should be at least 98.9%, because the number of the remaining long-running queries (misidentified as short-running) is $0.0092 * (1 - 0.989) = 0.0001$ of all queries, which is acceptable to optimize the 99.99th target percentile latency.

At this recall target, precision should stay high because the falsely predicted long-running queries consume unnecessary resources for acceleration. Improving precision is important: We show that under heavy load, we cannot reduce the tail latency without improved precision in Section 4.2.3 and 5.

Finally, the overhead involved in performing prediction must be small. Prediction adds additional work to query execution, increasing query response time. Since the average execution time of queries whose execution time is larger than 10 ms is 28 ms, adding 5% of it for prediction is a relatively small cost for the potential benefits. Here, we set the goal of less than 1.4 ms to predict query execution time.

3.2 Prior Work and its Limitation

We first review state-of-the-art predictor [16], which we call S as it uses **S**tatic features for prediction.

Static features are essentially statistics on terms and queries, based on which we can estimate the number of documents to be matched while applying early termination algorithms [5, 29]. Table 1 lists "memory-efficient" features (called "cheap features") to meet accuracy requirement without requiring a large memory footprint. Specifically, term feature, such as IDF score, is computed per query term, which is then combined into a single feature. Query features represent the complexity of queries, incurred from query personalization, relaxation, and rewriting.

We argue that S cannot meet the desired accuracy of reducing the extreme tail latency. In particular, S achieves a recall of 69.5% and precision of 80.3%. This recall is not high enough to reduce the 99.99th-percentile latency, which demands a recall of 98.9%.

It is possible for S to trade precision for higher recall. More precisely, S can predict a query as long-running if its predicted time is larger than a time threshold α. Lowering the time threshold α allows more queries to be identified as long-running queries, improving recall but reducing precision. When the time threshold is small enough (i.e., $\alpha = 2.5$ ms) for S to meet a recall of 98.9%, its precision drops to 1.1% only, which means we need to accelerate 83.5% of total queries, introducing significant overhead with much higher query latencies.

Our work investigates three techniques to improve the accuracy of the predictor significantly to reduce the 99.99th-percentile latency.

3.3 Delayed Prediction

We first propose *delayed prediction*, in which a query is executed for a short period of time D (which we call D delay threshold), then the prediction is made.

Delayed prediction exploits the workload characteristics and introduces three advantages: (1) The majority of the short-running queries are terminated within a very short time. From our query log, 80% of queries take less than 10 ms. These short-running queries would not be falsely identified as long-running queries, avoiding the overhead of accelerating them. (2) It reduces the runtime overhead of predicting those short-running queries, since these queries would terminate before making prediction. (3) With the short execution period, we can extract important runtime information on query processing, allowing us to employ dynamic features.

We determine the delay threshold D empirically. Longer delay allows most of the short-running queries to complete without prediction overhead and permits more time to collect dynamic signals, but prevents accelerating long-running queries early on. We evaluate several choices for D in a sensitivity study (Section 5.1.5), and choose $D = 10$ ms, which gives the best performance.

3.4 Dynamic Features

As for the second technique, we propose *DD* (for **D**elayed **Dynamic** prediction), employing dynamic features to enhance prediction accuracy. These dynamic features represent the runtime information of a query, which are collected after we run a query for a little while.

These dynamic features capture the following crucial information for prediction, unavailable from static features.

- First, some information is unknown at indexing time but affects query execution. Examples include the dynamic context of queries, including dynamic scores for pruning, location or language preference.

- Second, some information is too large to be indexed and stored in main memory. For example, "French fries recipe" appear frequently together such that aggregating the individual term features is inaccurate. Indexing such correlation using a trigram index would improve accuracy, their size is larger by orders of magnitude than individual term index and typically remains unindexed. Such correlation can be captured at runtime after processing the query for a short duration.

We focus on two categories of runtime information: matched documents and dynamic scores. Intuitively, when the number of matched documents is high on an early segment of the inverted index, we expect many matches in the entire index, indicating a long query. However, this indicator alone is not sufficient because a query with potentially many matches may terminate early without ranking all matches [5, 29] (or the queries without many matches may run long without such termination). We thus employ the information on dynamic scores: if scoring distribution shows high dynamic scores and small variance, the query is likely to terminate early, leading to shorter execution time. Inspired by these two complementary factors, we identify the following nine novel dynamic features as shown in Table 2.

- *DynScore*: statistics of the dynamic score distribution observed runtime (e.g., minimum, maximum, average, variance, and summation.) For example, *MaxDynScore* is the maximum of scores of the ranked documents at duration D.

- *NumCurrentMatchDoc*: the number of matched documents after D.

- *NumEstMatchDoc*: the estimation of total number of the matched documents when query processing is done. We adopt a simple linear estimate using:

$$\frac{\#\text{current matched documents}}{\#\text{processed documents}} \times \#\text{ total documents},$$

  where *# total documents* is the total number of documents managed by the ISN, which is in the scale of hundreds of million in our evaluation.

- *MatchingTime*: the time to find the next matched document. We use two features *FirstMatchingTime* and *AvgMatchingTime* to represent *MatchingTime* of the first document and its average respectively.

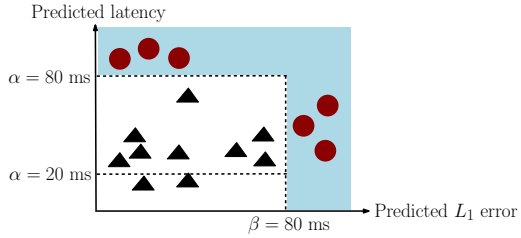


Figure 2: Illustration of selective prediction, where a red circle means a long-running query and a black triangle means a short-running query.

While dynamic features bring significant accuracy improvements, they are also inexpensive to obtain. Unlike some static features that may increase server memory footprint and index size, the dynamic features are byproducts of query processing and do not require main memory, which is scarce resource at index servers.

3.5 Selective Prediction

As for the third technique, we propose *DDS* (for **D**elayed, **D**ynamic, and **S**elective prediction), which exploits both execution time prediction and error estimation to selectively accelerate queries.

To motivate selective prediction, Figure 2 illustrates the problem of existing prediction. In this figure, existing approach estimates predicted latency (y-axis) then accelerates queries predicted to take longer than time threshold α. To identify all long-running queries (marked by circles), α should be set to 20 ms, which mis-predicts many short-running queries (marked by triangles) as long-running, causing low precision. In particular, for the queries with small prediction error, reducing time threshold can effectively improve recall with little trade-off on precision. However, we find that some mispredicted true long-running queries (with large prediction error) compromise precision by introducing many false positives.

For this reason, we extend our prediction framework to estimate not only latency but prediction error. More precisely, we introduce an error threshold β and accelerate the queries with predicted error greater than β. Back in our running

example in Figure 2, this enables us to accelerate two types of queries: (1) the queries with estimated latency greater than time threshold $\alpha = 80$ ms and (2) the queries with estimated error greater than error threshold $\beta = 80$ ms. The insight of selective prediction is that we keep time threshold high based on prediction with low error, and accelerate the queries with high predicted errors, so we will not miss those long-running queries with large uncertainty on its predicted execution time.

More specifically, we build two regressors, one of which is for predicting latency and another for regression error. The training set for the first regressor, f_1, is constructed as $D = \{\boldsymbol{x}_i, y_i\}_{i=1}^{n}$, where $\boldsymbol{x}_i$ is the i-th covariate and y_i is the i-th response variable. We use the covariate as static with dynamic features, and the response variable as execution time. The training set for the second regressor, f_2, is constructed by a validation set consisting of $V = \{\boldsymbol{x}_i, y_i^{err} = |f_1(\boldsymbol{x}_i) - y_i|\}_{i=1}^{m}$.

A query is determined as a long-running query if the predicted execution time is larger than α or the predicted L_1 error is larger than β. As shown in Figure 2, a query is decided as long-running if it is in the blue shaded region.

Note that we adopt boosted regressor tree [12] to identify long-running queries instead of a classifier (such as boosted classification tree [12] or SVM), because the time threshold T of defining long-running queries can be varied and the accuracy of classifier is similar with the one of regressor.

3.6 Summary

To reduce 99.99th-percentile response time of web queries, the predictor needs to achieve very high recall of 98.9% with good precision. To meet these requirements, we propose three important techniques: (1) delayed prediction to remove unnecessary short-running queries, (2) dynamic features to achieve higher accuracy, and (3) selective prediction that combines predicted execution time with the estimated prediction error. We empirically evaluate our framework in Section 4, and we show how to leverage the framework to accelerate long-running queries in Section 5.

4. EXPERIMENTS

This section presents experimental results to validate the accuracy and effectiveness of the proposed framework. We conduct our evaluation using production index and query log. Our results show the benefits of delayed prediction, dynamic features and selective prediction. We observe 957% improvement in precision at the desired 98.9% recall compared with the prior work.

4.1 Experimental Setup

We first describe the setting for evaluation. We play 69,010 queries from a production query log at 100 QPS (queries per second) and measure their execution time. We extract static and dynamic features of these queries to construct a training and test set. In particular, we instrument the code of query processing to log all dynamic scores of the ranked documents, which we use to extract dynamic features. For the static features, we follow the same procedure introduced in [16]. To eliminate outliers, we execute the same query three times and plot its minimum. We compare *DDS* with *S*:

- *S* implements a predictor [16] using the static features discussed in Section 3.2. This is our baseline.

Table 3: Top-10 feature importance of static + dynamic features.

Feature	Importance
NumEstMatchDoc	1
MinDynScore	0.7075
MinIDF	0.2767
VarIDF	0.2730
MaxDynScore	0.2662
SumIDF	0.2076
VarDynScore	0.1982
AvgDynScore	0.1831
Complexity	0.1791
NumBefore	0.1762

- *Dynamic-Delayed-Selective (DDS)* accelerates a query when either its predicted latency $\geq$ the time threshold α or its predicted L_1 error $\geq$ the error threshold β.

We also introduce two intermediate steps from *S* to *DDS*, which are used to illustrate the individual contribution of each technique: delayed prediction, dynamic features, and selective prediction.

- *Static-Delayed (SD)* uses the same static features as in *S*, but makes a decision after $D = 10$ ms.
- *Dynamic-Delayed (DD)* delays prediction and uses both static features and dynamic features that we proposed in Section 3.2.

We use all 69,010 queries to evaluate *S*, and use 14,546 queries to evaluate delayed prediction frameworks including *DDS, SD*, and DD, because only 14,546 queries are not terminated after 10 ms. We use 10-fold cross validation (10 folds are randomly selected from a dataset) and repeat this procedure 5 times and report the average number to prevent biased results. We use boosted regression tree to predict the execution time and regression errors, where we use 100 regression trees.

4.2 Experimental Results

In this section, we identify which dynamic features are important to predict execution time, and compare *DDS* to *S*, showing the contribution of each technique: delayed prediction, dynamic features, and selective prediction.

4.2.1 Importance of Dynamic Features

Table 3 shows the top-10 feature selection results, where the importance is normalized by the highest value. The importance of a feature is evaluated by per-feature gain from the boosted tree, proportional to the total error reduction per split in the tree. The results show that dynamic features provide critical information on predicting query execution time: two dynamic features (NumEstMatchDoc and MinDynScore) outrank most of static features in terms of effectiveness. NumEstMatchDoc is most effective, being a simple but powerful estimator for the number of documents to be matched, which is highly correlated with the query execution time. However, this alone cannot predict time, as the query may terminate early depending on scoring distribution for the query. Various summary of such distribution (min, max, avg, and variance) thus play important and complementary roles in prediction.

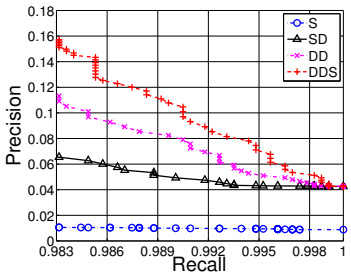


Figure 3: Precision and recall curves of four approaches by varying the time threshold α.

4.2.2 Low Prediction Overhead

As *DDS* delays prediction until a query runs for 10 ms, 80% of queries complete before that, and only 20% of queries use the predictor to estimate their execution time. In addition, the runtime overhead of prediction is less than 1 ms, which is less than 5% of the average execution time of these queries that use prediction.

4.2.3 Improved Precision

Figure 3 compares precision recall curves for *S*, *SD*, *DD* and *DDS*. All curves are obtained by varying the time threshold α (smaller α trades precision for higher recall). We set the error threshold of *DDS* to be $\beta = 10$ ms, which we empirically find to be effective. For the target 98.9% recall (to reduce 99.99th-percentile latency), Figure 3 shows, precision of *S* is about 1.1%, and *DDS* improves the precision to 11.63%, representing a 957% improvement over *S*. The detailed contributions of delayed prediction, dynamic features, and selection prediction are described next.

Contribution of delayed prediction (*S* versus *SD*): Figure 3 shows that *SD* (Static-Delayed) improves the precision from 1.1% to 5.25% compared with *S*. *SD* consistently outperforms *S* because 80% of queries terminate within 10 ms — SD saves prediction time and avoids prediction errors for those queries.

Contribution of dynamic features (*SD* versus *DD*): Figure 3 shows that *DD* (Dynamic-Delayed) improves the precision from 5.25% to 9.46%, which confirms the usefulness of dynamic features. To evaluate the effectiveness of dynamic features further, Table 4 shows the precision and recall of *SD* and *DD* while varying α. We observe that dynamic features consistently improve both precision and recall in all settings. In addition, our target recall of 98.9% is reached by *DD* when $\alpha = 20$ ms, while not by SD in the same setting.

Contribution of selective prediction (*DD* versus *DDS*): Figure 3 shows that *DDS* improves the precision from 9.46% to 11.63%, which validates the usefulness of selective prediction. In other words, selective prediction in *DDS* helps to identify long-running queries with large prediction error, and produces less false positives than *DD*.

At the recall target of 98.9%, the difference in precision between *DDS* and *S* is significant with attendant performance impact. For the query workload, we find that there are 0.92% of truly long-running queries. A precision of 11.63% indicates that *DDS* identifies 7.9% (= 0.92%/11.63%) of all queries as long-running, which is an affordable number to accelerate. In comparison, to meet the same high recall, using the state-of-the-art prior work *S* has precision of 1.1%, and thus identifies 83.5% of all queries as long-running, incurring a large cost — 9.57 times higher than *DDS* — in order to accelerate them. We discuss implications of these results in the next section.

Table 4: Precision and recall comparison between SD and DD with respect to varying α. We report each accuracy value as 'average (standard deviation)'.

Category	SD	DD
Precision ($\alpha = 100$ ms)	80.3%(0.44%)	**87.6%**(0.84%)
Recall ($\alpha = 100$ ms)	69.5%(0.44%)	**78.4%**(0.33%)
Precision ($\alpha = 60$ ms)	59.6%(0.57%)	**63.1%**(0.45%)
Recall ($\alpha = 60$ ms)	82.2%(0.33%)	**88.9%**(0.44%)
Precision ($\alpha = 40$ ms)	35.7%(0.3%)	**36.3%**(0.36%)
Recall ($\alpha = 40$ ms)	88.2%(0.3%)	**93.7%**(0.5%)
Precision ($\alpha = 20$ ms)	8.3%(0.05%)	**9.3%**(0.03%)
Recall ($\alpha = 20$ ms)	97.2%(0.4%)	**98.9%**(0.1%)

5. USING DDS TO REDUCE TAIL LATENCY

This section demonstrates how to use the proposed prediction framework *DDS* to reduce the extreme tail latency. The key idea is that *DDS* enables us to accurately accelerate only the long-running queries to reduce tail latency with affordable cost. We simulate the Bing search engine on two scenarios: (1) to parallelize long-running queries on multicore servers and (2) to schedule long-running queries on fast cores of heterogeneous servers. We evaluate the tail latency of each index server node (ISN) as well as at the aggregator for a cluster of ISNs. The simulation results show that, for both scenarios, *DDS* effectively reduces tail latency for a wide range of load, while the existing solutions fail to meet the latency target at medium and high load. At the desired tail latency, *DDS* improves the server throughput by more than 70%, potentially saving 44% search servers.

5.1 Query Parallelization

Parallelizing query processing on a search server is a potential solution to reduce query execution time [27, 15]. This is motivated by current hardware trends. A modern server has several cores, and therefore, parallelization allows multiple threads to execute a query concurrently using the available cores to reduce query execution time.

Choosing a parallelization strategy is challenging. When servers are lightly loaded and there is sufficient number of available cores, parallelizing the execution of all queries reduces their execution time, thereby reducing the response time. However, parallelizing all queries is ineffective under medium and high load because it comes with an overhead that varies among queries. Long-running queries achieve better speedup with lower overhead and higher parallelization efficiency. In contrast, parallelizing short-running queries is ineffective, giving no performance benefit while consuming additional resources taken away from long-running queries.

Figure 4 presents the measurement results on the parallelization speedup of Bing queries with different execution time (Figure 3 of [16]). The implementation details on how to execute a web search query using multiple threads and what causes the speedup/parallelism overhead are beyond the scope of this paper. Interested readers may refer to [16, 15]. The results show the queries that run longer than

80 ms achieve more than 4 times speedup by using 6 threads. In contrast, using 6 threads, the short-running queries that complete within 30 ms just achieve about 1.15 times of speedup because their execution time is dominated by the sequential part (*i.e.,* non-parallelized part) of the query processing. As the load increases, spare processor resources become limited and we can no longer afford to parallelize all queries. We thus parallelize long-running queries to reduce the high-percentile response time, and execute short-running queries sequentially to reduce parallelization overhead. We use *DDS* to decide if a query shall be parallelized.

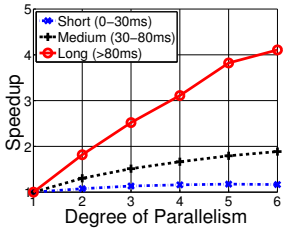


Figure 4: Parallelization efficiency of short, medium and long-running queries.

5.1.1 Experimental setup

Our experiments compare two techniques *DDS* and *S*. *S* uses static features from prior work [16] to predict query execution time before running the query and parallelizes the predicted long-running queries with 4-way parallelism. *DDS* runs a query for 10 ms sequentially and, if the query has not completed by then, it uses dynamic features of the query to predict query execution time and the prediction error. *DDS* parallelizes the query with 4-way parallelism if it is predicted as long running or if the prediction error of the query is larger than the error threshold (as in Section 4.1). Changing parallelism degree of a web search query at runtime is supported by using fine-grained parallelism [10]. In addition, we add the results of two more techniques as references: sequential execution that runs each query sequentially, and no-prediction that parallelizes all queries with 4-way parallelism.

We model a web search engine based on the architecture described in Section 2 and simulate it using a Java-based discrete-event simulator called DESMO-J [23]. The simulation models an aggregator and its 40 ISNs; each ISN has 12 cores. We drive the simulation using production query log, execution time profile (Figure 1(b)) and speedup profile (Figure 4). Queries arrive following a Poisson process, and we vary system load by changing the average arrival rate. We report the average number of 5 runs and use error bars to represent their standard deviation in the figures.

The proposed techniques target reducing the tail latency at medium and high load, rather than across all load regions. Notice that we operate Bing servers only at medium to high loads: Light load means that the system is over provisioned and load can be consolidated to use fewer server clusters; Extremely high load means that system is under provisioned and cannot meet the response time SLA for the tail latency, and load should be balanced among more server clusters.

5.1.2 Response time reduction at ISN

Figure 5 compares *DDS* with sequential execution results and two other parallelization approaches on (a) 99.99th-percentile, (b) 99.9th-percentile, and (c) mean response time. From the results we make three observations. First, parallelization significantly reduces the tail latency. Specifically, at light load (up to 350 QPS), all parallelization approaches reduce the 99.99th-percentile response time from 200 ms using sequential execution to 120 ms. Second, at medium to high load (350–700 QPS), *DDS* still achieves the same level of reduction, reducing the tail latency by 40% over sequential execution. In contrast, no-prediction and *S* have very high tail latency. Third, at extremely high load (>830 QPS), sequential execution has lower response time than the parallelized approaches as there is no free core available to run queries in parallel. However, we do not operate our servers at such very high load which leads to violating the response time SLA of 120 ms at 99.99th-percentile. The results of 99.9th-percentile show similar trend. We also include the mean response time results for completeness.

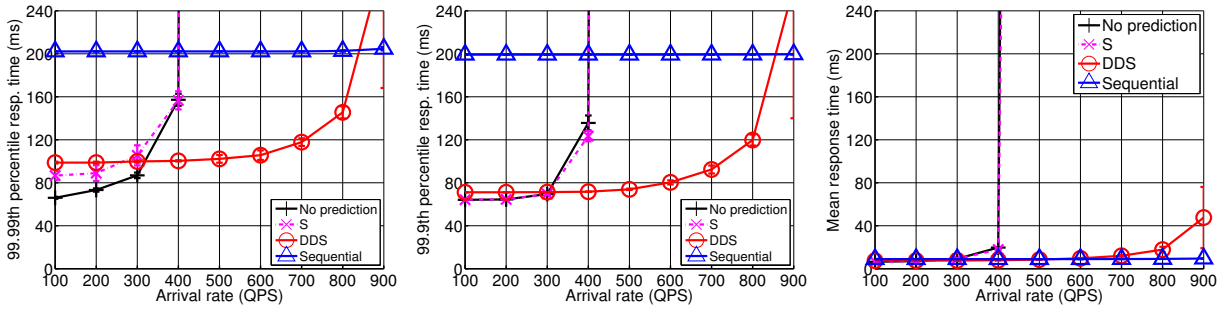
Insights behind performance improvements. We explain why *DDS* outperforms *S* at medium to high load: *DDS* enables judicious utilization of cores to long-running queries because of its higher precision.

In order to reduce 99.99th-percentile response time, we have shown in Section 3.1 that a prediction algorithm needs to achieve high recall of 98.9%. To obtain this recall, different prediction algorithms have different precision, which implies the number of queries the algorithms identify as long running (including both true long-running queries and false positive ones). *S* has precision of 1.1%. As there are 0.92% of truly long-running queries, a precision of 1.1% indicates that *S* identifies about 83.6% (= 0.92%/1.1%) of all queries as long-running, among which, majority are false positives. At light load, the parallelization overhead to accelerate the false positive queries is tolerable as there are available cores. However, when the load increases, these false positive queries take away resources from real long-running queries and their parallelization overhead prolongs the waiting time of all queries, resulting in higher response time. In comparison, *DDS* achieves the precision of 11.63%, parallelizing only 7.93% (= 0.92%/11.63%) of queries, which significantly reduces the number of queries to accelerate and their corresponding overhead. Therefore, even at high load, *DDS* still effectively reduces tail latency.

5.1.3 Contribution of three techniques of DDS

The performance improvement of *DDS* comes from its three techniques — delayed prediction, dynamic features, and selective prediction, and Figure 6 quantifies their individual contributions. In particular, we add two intermediate steps from *S* to *DDS*: *Static-Delayed (SD)*, which delays prediction but only uses static features, and *Dynamic-Delayed (DD)*, which delays prediction and uses both static and dynamic features, but without estimating prediction error.

To meet 120 ms SLA on the 99.99th-percentile response time, Figure 6 shows the following: (1) *SD* outperforms *S*, increasing throughput from 390 QPS to 600 QPS. The reason is that 80% short-running queries complete within 10 ms, and the delayed prediction executes all of these short-running queries sequentially without incurring unnecessary overhead. (2) *DD* increases the throughput further to 650 QPS. The improvement comes from the improved prediction accuracy: dynamic features increase precision to 9.46%, and thus only 9.7% of the total queries need to be parallelized. (3) *DDS* achieves the precision of 11.63% and offers a final



(a) 99.99th-percentile response time (b) 99.9th-percentile response time (c) mean response time

Figure 5: Response time of sequential and three parallel query execution approaches at ISN.

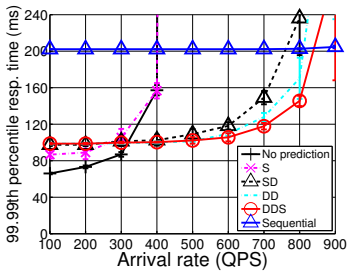
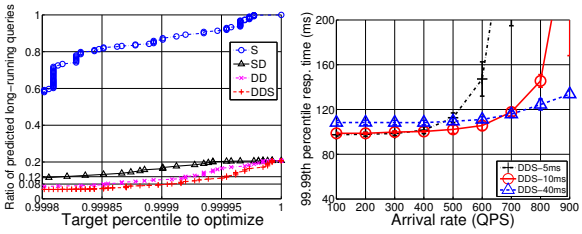


Figure 6: 99.99th-percentile response time for *S*, *SD*, *DD* and *DDS* at ISN.

boost of throughput to 700 QPS by accelerating only 7.9% of queries. It demonstrates the effectiveness of delaying the prediction, using dynamic features, and estimating the prediction error.

5.1.4 *Sensitivity to different target percentiles*

Figure 7(a) shows two observations when we vary the target response percentile: (1) For any given algorithm, in order to optimize for higher percentile, the ratio of the predicted long-running queries increases: trading precision for higher recall. (2) *DDS* consistently outperforms the other schemes for various target percentiles: for any given target, it requires to parallelize the smallest number of queries as it achieves the highest precision.



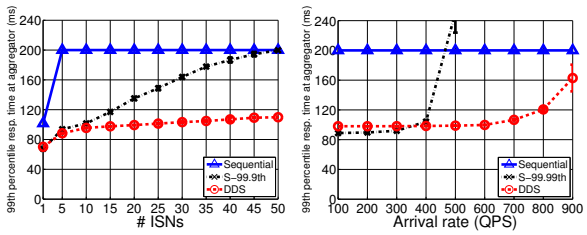
(a) (b)

Figure 7: Sensitivity study: (a) Ratio of the predicted long-running queries with varying target percentile latency; (b) Response time of *DDS* with various delay thresholds: $D = 5, 10, 40$ ms.

5.1.5 *Sensitivity to delay threshold D*

Figure 7(b) shows 99.99th response time for *DDS* with various delay threshold D where *DDS-5ms* means that the prediction is done after 5 ms of sequential execution. We observe that 10 ms is a practically sufficient choice. The 99.99th response time for 40 ms is consistently larger than 10 ms for small-to-mid loads, because the true long-running queries are delayed longer to be accelerated. The 99.99th response time for 5 ms is larger than 10 ms at high load, because earlier prediction does not collect sufficiently accurate dynamic information to achieve precision.

5.2 Latency Reduction at Aggregator



(a) Vary #ISNs, 700 QPS (b) Vary load, 40 ISNs

Figure 8: 99th-percentile latency at an aggregator.

While previous results focus on latency reduction of each ISN, this section evaluates the benefits of *DDS* on reducing aggregator-level latency. An aggregator collects results from all of its ISNs, and thus its latency is determined by the slowest ISN. To reduce tail latency at the aggregator, we need to reduce a much higher tail at each ISN. For example, if the desired $x = 99$th-percentile latency is $Q = 120$ for an aggregator with $k = 40$ ISNs, then each ISN needs to have $\sqrt[k]{x/100} \times 100 = 99.975 \approx 99.99$ percentage of queries with latency Q or lower, i.e., the individual ISN should have a 99.99th-percentile latency of $Q = 120$ ms or lower.[1]

We vary the number of ISNs and measure the 99th-percentile response time at the aggregator. We evaluate two approaches at 700 QPS, labelled *DDS* and *S*-99.9th. First we employ *DDS* to reduce the 99.99th-percentile latency at each ISN. Next, we introduce *S*-99.9th, which uses static features to reduce 99.9th-percentile latency at each ISN. Notice that we do not use *S*-99.99th, the default scheme with static features, which has higher recall but lower precision.

[1]We assume that ISNs operate independently since each ISN works on a random hashed partition of web documents.

At 700 QPS, S-99.99th produces very high parallelism overhead, and its latency substantially exceeds 200 ms (Figure 5). Finally, we plot the sequential results for reference.

Figure 8(a) shows how the aggregator latency of the three schemes rises with increasing number of ISNs. We set the target 99th-percentile of aggregator latency to be 120 ms. At the extreme case of only one ISN for the aggregator, we need to reduce 99th-percentile of that ISN, and the two approaches meet the goal. When the aggregator has 40 ISNs, we need to reduce around 99.99th-percentile at each ISN — S-99.9th cannot meet the goal any more. The results are consistent with our analysis. Since in practice each aggregators has many ISNs, this experiment highlights the importance of reducing the extreme tail at each ISN server.

Figure 8(b) compares the aggregator 99th-percentile latency for *DDS* and S-99.9th with varying load and with $k = 40$ ISNs. The results show *DDS* outperforms the other approaches, reducing the aggregator-level tail latency for moderate and heavy load by 40% from 200 to 120 ms.

Throughput improvement. Using *DDS* to select which queries to parallelize improves system throughput. Figure 8(b) shows that, to meet 120 ms 99th-percentile latency at the aggregator, *DDS* sustains arrival rates up to 700 QPS while the other schemes support up to around 390 QPS. *DDS* improves throughput by more than 70%.

Assume the search engine has a total workload of X QPS, and each aggregator has k ISNs. Using *DDS* requires $X/700 \times k$ servers while using fixed parallelization needs $X/390 \times k$ servers: Here, *DDS* saves $(X/390 - X/700)/(X/390) = 44\%$ of the servers to serve the same workload. As commercial search engine uses thousands of production servers, these savings are significant.

5.3 Scheduling on Heterogenous Server

A heterogeneous multicore processor is an emerging hardware composed of cores with differentiated power and performance characteristics, typically combining energy-efficient slow cores with high-performance fast cores. All cores execute the same instruction set, but they run at different speeds — the faster the core, the more power it consumes. Since power consumption increases faster than speed, a fast core executes a request in less time than a slow core, but consumes more energy. There are several proposals for heterogeneous processors [18, 2, 7], e.g., ARM recently announced their big.LITTLE processor for production [13].

We exploit the prediction results of *DDS* to improve search engine performance on the emerging heterogeneous hardware. Prior work [14] shows that, as ISNs of web search have CPU-intensive workload, a high-performance (fast) core completes a query with less time than an energy-efficient (slow) core but consumes more energy. We accelerate the predicted long-running queries using fast cores and execute short-running queries on slow cores. The high recall achieved by DDS allows us to correctly predict the majority of the long-running queries and execute them using fast cores to reduce tail latency. Good precision is also important here: although faster cores provide higher performance and complete queries faster, they have lower energy efficiency, i.e., to complete the same amount of work, they consume more energy than using slower and energy-efficient cores [24]. The good precision of DDS allows us to execute most of the short-running queries on slow cores to reduce energy consumption and improve system throughput.

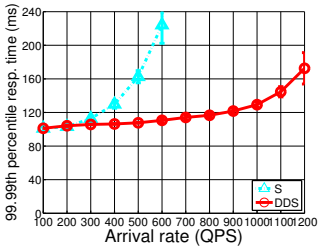


Figure 9: 99.99th-percentile response time on the heterogeneous server.

We simulate a heterogeneous server composed of 3 fast cores (i7 Sandy bridge cores) and 6 slow cores (AtomD cores). We compare DDS with S in our evaluation using the same workload as Section 5.1. Due to space limit, we only summarize the key results. As shown in Figure 9, at moderate to high load, comparing with S, DDS reduces 99.99th-percentile latency of ISN from 200 ms to 120 ms. To meet 99.99th-percentile latency target of 120 ms, *DDS* supports up to 800 QPS, while S only obtains 330 QPS. *DDS* more than doubles the server throughput. In order words, to process the same amount of query workloads, *DDS* saves more than half of the servers.

6. RELATED WORK

Execution time prediction on search queries. Prediction of search query performance can be categorized into quality and execution time prediction. Examples of quality prediction include the prediction of user satisfaction [17], and clicks to mobile advertisements [22].

In contrast, this work focuses on predicting the execution time of web search queries. Moffat *et al.* [21] show that the execution time of a query is related to the posting list lengths of its constituent query terms. However, under dynamic pruning strategies, only a limited portion of such listing is scored [1]. Macdonald *et al.* [20] incorporate term-level statistics to predict the effect of dynamic pruning. In particular, various term-level statistics (static features) are computed for each term offline. When a query arrives, the term-level features are aggregated into query-level statistics, which are used as inputs to a regression model. Meanwhile, [16] proposes query features modelling the complexity from query relaxation, rewriting and personalization, and identifies an effective subset outperforming existing predictors in terms of accuracy with lower memory footprint.

However, these predictors, relying solely on static features, could not achieve our accuracy requirements. We thus propose the use of dynamic features and error prediction for execution time prediction. We empirically show that this combination greatly improves prediction accuracy over state-of-the-art predictors that use only static features. Prediction accuracy gains can be used to optimize search systems in many ways: Macdonald *et al.* [20] uses the predicted execution time for query scheduling, [16] for parallelization, [28] for query pruning, and [11] for reducing power consumption. We use parallelization and scheduling on heterogeneous hardware as our scenarios for demonstration.

Search query parallelization. Parallelizing the processing of search queries is proposed to reduce query execution time [27, 15]. However, the prior studies do not consider

the prediction of query execution time, thus they parallelize all queries, which wastes resources. Recently, a prediction-based parallelization scheme [16] predicts query execution time to parallelize long-running queries. This work improves over prior work by employing delayed, dynamic, and selective prediction. Frachtenberg [10] can be viewed as a simplistic delayed and dynamic prediction using a single feature, which is an estimation on the number of matched documents. This single feature without trained regression model does not capture other important factors contributing to the query execution time such as query complexity and pruning, and it cannot accurately predict query execution time.

Scheduling on heterogeneous servers. Prior work shows that a heterogeneous server is more suitable for applications consisting of tasks of diverse demands to improve performance or save energy [13, 8, 19, 7, 25]. For example, users run delay-sensitive tasks using fast cores, while background services use slow cores [13]. Lakshminarayana et al. schedule the thread in a parallel job with a larger remaining execution time on a fast core [19]. They predict the remaining time of jobs at PARSEC benchmark [3] based on dynamic profiling. Since these applications are not directly related to search engines, the proposed features for predicting job execution time are not appropriate for our purpose. Our work exploits the diversity (short-running versus long-running jobs) in the demand of search queries and shows how to use the predicted execution time to improve search engine performance using heterogeneous hardware.

7. CONCLUSIONS

This paper studies reducing the extreme tail latency of web search server by predicting and accelerating long-running queries. We first show that the predictor must offer high recall and good precision as our prediction requirements. We propose a novel prediction framework, *DDS*, combining delayed prediction, dynamic features and prediction error estimation. In particular, *DDS* delays the prediction until we collect dynamic signals, which are highly effective on improving prediction accuracy. Moreover, *DDS* estimates both latency and error to selectively accelerate queries, achieving high recall target with good precision. Our simulation results show that *DDS* effectively reduces the tail latency and significantly improves server throughput.

Acknowledgements: This work was partially supported by the IT R&D Program of MSIP/IITP (14-824-09-014, Machine Learning Center), National Research Foundation (NRF) of Korea (NRF-2013R1A2A2A01067464), and MSIP/Microsoft Research under ICT/SW Creative research program (NIPA-2014-H0510-14-1018). We appreciate the constructive comments provided by Myeongjae Jeon.

8. REFERENCES

[1] R. Baeza-Yates, A. Gionis, F. P. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. Design trade-offs for search engine caching. *ACM Transactions on Web*, 2008.

[2] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. *ACM Computing Frontiers*, 2006.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. *Technical Report*, 2008.

[4] S. Briesemeister, J. Rahnenfuhrer, and O. Kohlbacher. No longer confidential: Estimating the confidence of individual regression predictions. *PLos ONE*, 2012.

[5] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, 2003.

[6] C. J. Burges, R. Ragno, and Q. V. Le. Learning to rank with nonsmooth cost functions. In *NIPS*, 2006.

[7] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In *DAC*, 2009.

[8] K. V. Craeynest, A. Jalelle, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *ISCA*, 2012.

[9] J. Dean and L. A. Barroso. The tail at scale. In *Communications of the ACM*, 2013.

[10] E. Frachtenberg. Reducing query latencies in web search using fine-grained parallelism. *World Wide Web*, 2009.

[11] A. Freire, C. Macdonald, N. Tonellotto, I. Ounis, and F. Cacheda. A self-adapting latency/power tradeoff model for replicated search engines. In *WSDM*, 2014.

[12] J. Friedman. Greedy function approximation: a gradient boosting machine. In *Annals of Statistics*, 2001.

[13] P. Greenhalgh. Big.little processing with arm cortex-a15 & cortex-a7. *ARM Whitepaper*, 2011.

[14] 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.

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

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

[17] Y. Kim, A. Hassan, R. W. White, and Y.-M. Wang. Playing by the rules: Mining query associations to predict search performance. In *WSDM*, 2013.

[18] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multicore architectures: The potential for processor power reduction. In *MICRO*, 2003.

[19] N. B. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. In *SC*, 2009.

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

[21] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, 2007.

[22] R. J. Oentaryo, E. P. Lim, D. J. W. Low, D. Lo, and M. Finegold. Predicting response in mobile advertising with hierarchical importance-aware factorization machine. In *WSDM*, 2014.

[23] B. Page and T. Lechler. Desmo-J. `http://desmoj.sourceforge.net/overview.html`.

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

[25] J. C. Saez, D. Shelepov, A. Fedorova, and M. Prieto. Leveraging workload diversity through os scheduling to maximize performance on single-isa heterogeneous multicore systems. *JPDC*, 2011.

[26] E. Schurman and J. Brutlag. Performance related changes and their user impact. *Velocity*, 2009.

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

[28] N. Tonellotto, C. Macdonald, and I. Ounis. Efficient and effective retrieval using selective pruning. In *WSDM*, 2013.

[29] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing and Management*, 1995.