Abstract

Interactive web services increasingly drive critical business workloads such as search, advertising, games, shopping, and finance. Whereas optimizing parallel programs and distributed server systems have historically focused on average latency and throughput, the primary metric for interactive applications is instead consistent responsiveness, i.e., minimizing the number of requests that miss a latency target. This paper is the first to show how to generalize work-stealing, which is traditionally used to minimize the makespan of a single parallel job, to optimize for a target latency in interactive services with multiple parallel requests.

We design a new adaptive work stealing policy, called tail-control, that reduces the number of requests that miss a target latency. It uses instantaneous request progress, system load, and a target latency to choose when to parallelize requests with stealing, when to explore new requests, and when to limit parallelism of large requests. We implement this approach in the Intel Thread Building Block (TBB) library and evaluate it on real-world workloads and synthetic workloads. The tail-control policy substantially reduces the number of requests exceeding the desired target latency and delivers up to 58% relative improvement over various baseline policies. This generalization of work stealing for multiple requests effectively optimizes the number of requests that complete within a target latency, a key metric for interactive services.

1. Introduction

Delivering consistent interactive latencies is the key performance metric for interactive cloud services, such as web search, stock trading, ads, and online gaming. The services with the most fluid and seamless responsiveness incur a substantial competitive advantage in attracting and captivating users over less responsive systems (Dean and Barroso 2013; He et al. 2012; Yi et al. 2008). Many such services are deployed on cloud systems that span hundreds or thousands of servers, where achieving interactive latencies is even more challenging (Jeon et al. 2013; Apache Lucene 2014; Mars et al. 2011; Ren et al. 2013; Zircon Computing 2010; Yeh et al. 2006; Raman et al. 2011). The seemingly infrequent occurrence of high latency responses at one server is significantly amplified because services aggregate responses from large number of servers. Therefore, these servers are designed to minimize tail latency — the latency of requests in the high percentiles, e.g., 99th percentile latency (Dean and Barroso 2013; DeCandia et al. 2007; Haque et al. 2015; He et al. 2012; Yi et al. 2008). This paper studies a closely related problem, optimizing for a target latency — given a target latency, the system minimizes the number of requests whose latency exceeds this target.

We explore interactive services on a multicore server with multiple parallelizable requests. Interactive service workloads tend to be computationally intensive with highly variable work demand (Dean and Barroso 2013; He et al. 2012; Jeon et al. 2013; Kanev et al. 2015; Haque et al. 2015; Zircon Computing 2010). A web search engine, for example, represents, partitions, and replicates billions of documents on thousands of servers to meet responsiveness requirements. Request work is unknown when it arrives at a server and prediction is unappealing because it is never perfect or free (Lorch and Smith 2001; Hsu et al. 2015; Jalaparti et al. 2013; Kim et al. 2015b). The amount of work per request varies widely. The work of the median and the 99th percentile requests often differ by orders of magnitude, ranging from 10 to over 100. Moreover, the workload is highly parallelizable — it exhibits inter-request parallelism (multiple distinct requests execute in parallel) and fine-grain intra-request parallelism (Jeon et al. 2013; Haque et al. 2015; Zircon Computing 2010). For instance, search could process every document independently. More generally, the parallelism of these requests may change as the request executes.

Such workloads are excellent candidates for dynamic multithreading, where the application expresses its logical parallelism with spawn/sync, fork/join, or parallel loops, and the runtime schedules the parallel computation, mapping it to available processing resources dynamically. This model has the following advantages. 1) It separates the scheduling logic from the application logic and thus the runtime adapts the application to different hardware and number of processors without any changes to the application. 2) This model is general and allows developers to structure parallelism in their programs in a natural and efficient manner. 3) Many modern concurrency platforms support dynamic multithreading, including Cilk dialects (Intela 2013; Frigo et al. 1998; Danaher et al. 2006; Leiserson 2010; Lee et al. 2013), Habanero (Barik et al. 2009; Cavigli et al. 2011), Java Fork/Join (Lea 2000; Kumar et al. 2012, 2014), OpenMP (OpenMP 2013), Task Parallel Library (Leijen et al. 2009), Threading Building Blocks (TBB) (Reinders 2010), and X10 (Charles et al. 2005; Kumar et al. 2012, 2014).

Most of these platforms use a work-stealing scheduler, which is efficient both in theory and in practice (Blumofe et al. 1995; Blumofe and Leiserson 1999; Arora et al. 2001; Kumar et al. 2012, 2014). Work-stealing uses a distributed scheduling strategy with low scheduling overheads. The goal of traditional work-stealing is...
to deliver the smallest possible makespan for one job. In contrast, interactive services have multiple requests (jobs) and the goal is to meet a target latency. To exploit the benefits of work-stealing’s dynamic multithreading for interactive services, we modify work-stealing so it can optimize for target latency.

While work-stealing has not been explored for interactive services, researchers have considered parallelizing requests for the related problem of minimizing tail latency in interactive services (Jeon et al. 2013; Kim et al. 2015a; Haque et al. 2015). This work applies higher degrees of parallelism to large requests that perform significantly more work than other small requests. They assume that small requests will meet the target latency even without parallelism. In contrast, we show when the system is heavily loaded, a large parallel request can monopolize many cores, imposing queuing delays on the execution of many small requests and thus increase their latencies much more than necessary. In this work, we investigate a different strategy — under heavy instantaneous load, the system explicitly sacrifices a small number of large requests. We execute them serially so they occupy fewer resources, enabling smaller waiting requests to make the target latency.

Designing such a strategy faces several challenges. (1) When a request arrives in the system, the request’s work is unknown. (2) Which requests to serialize depends on the request’s progress, the target latency, and current load. When inter-request parallelism is exploited to lower tail latencies, serialized large requests will almost certainly miss the target latency and thus we must not unnecessarily serialize large requests. (3) In interactive services, we know the expected average system load and workload distribution, but the instantaneous load can vary substantially due to unpredictable request arrival patterns. Whether a particular request should be serialized depends on its progress and the instantaneous load. If the system is heavily loaded, even a moderately large request may increase the waiting time of many small requests. Therefore, we must be more aggressive about serializing requests. When the system is lightly loaded, we should be less aggressive and execute even very large requests in parallel so they can meet the target latency. In short, there is no fixed threshold for classifying a request as large and serializing it; rather, the runtime should adapt the threshold based on instantaneous system load.

Based on these intuitions, we introduce the tail-control work-stealing scheduling strategy for interactive workloads with multiple simultaneous requests. Tail-control continuously monitors the system load, estimated by the number of active requests. It aggressively parallelizes requests when the system is lightly loaded. When the system is momentarily overloaded, it identifies and serializes large requests to limit their impact on other waiting requests. Specifically, the contributions of this paper are as follows:

1. We design an offline threshold-calculation algorithm that takes as input the target latency, the work demand distribution, and the number of processors on the server and calculates a large request threshold for every value of the instantaneous load. It generates offline a table indexed by system load that the online scheduler uses to decide which requests to serialize.

2. We extend work-stealing to handle multiple requests simultaneously and incorporate the tail-control strategy into the stealing protocol. We perform bookkeeping to track a) the instantaneous system load; and b) the total work done by each request. We perform most bookkeeping in a distributed fashion, which amortizes its overhead with steal attempts. Our modified work-stealing scheduler does not add any constraints on the programming model and thus can schedule any workload with dynamic multithreaded jobs.

3. We implement a tail-control work-stealing server in the Intel Thread Building Block (TBB) runtime library (Reinders 2010) and evaluate the system with several interactive server workloads: a Bing search workload, a finance server workload (He et al. 2012), and synthetic workloads with long-tail distributions. We compare tail-control with three baseline work-stealing schedulers, steal-first, admit-first, and default TBB. The empirical results show that tail-control significantly outperforms them, achieving up to a 58% reduction in the number of requests that exceed the target latency.

2. Background and Terminology

This section characterizes interactive services and available intra-request parallelism. We review work-stealing basics and start with some terminology.

**Terminology.** The response time (latency) of an interactive service request is the time elapsed between the time when the request arrives in the system and the time when the request completes. Once a request arrives, it is active until its completion. A request is admitted once the system starts working on it. An active request is thus either executing or waiting. Since the service may perform some or all of the request in parallel to reduce its execution time, we define request work as the total amount of computation time that all workers spend on it. A request is large or small according to its relative total work. A request misses a specified target latency if its latency is larger than the target.

**Characteristics of Interactive Services.** Three important characteristics of interactive services inform our scheduler design. First, many interactive services, such as web search, stock trading, online gaming, and video streaming are computationally intensive (Dean and Barroso 2013; He et al. 2012; Jeon et al. 2013; Kanev et al. 2015). For instance, banks and fund management companies evaluate thousands of financial derivatives every day, submitting requests that value derivatives and then making immediate trading decisions — many of these methods use computationally intensive Monte Carlo methods (Broadie and Glasserman 1996; Cortazar et al. 2008). Second, the work demand per request can be highly variable. Figure 1 shows representative workload distributions for Bing and a finance server requests executing sequentially. Many requests are small and the smallest are a factor of 40 smaller than the largest requests for Bing and 13 for the finance server. Third, the requests often have internal parallelism and each request can potentially utilize multiple cores.

**Work-Stealing Baseline for a Single Job.** In a work-stealing scheduler for a single job, there is typically one worker (software thread) for each hardware context (core). Each job has multiple
tasks that may execute in parallel and are created by the workers working on that job. Each worker manages its tasks in a double ended work queue, called a deque. When the task a worker \( p \) is currently executing spawns additional tasks, it places these tasks at the bottom of its own deque. When a worker \( p \) finishes executing its current task, it pops a task from the bottom of its own deque. If \( p \)’s deque is empty, then \( p \) becomes a thief. It picks a victim uniformly at random from the other workers, and steals work from the top of the victim’s deque. If the victim’s deque is empty, \( p \) performs another random steal attempt.

Given a single job, work stealing attains asymptotically optimal and scalable performance with high probability (Blumofe and Leiserson 1999; Arora et al. 2001) and performs well in practice (Blumofe et al. 1995; Kumar et al. 2012, 2014). The following properties of work stealing contribute to making it efficient. (1) It requires little synchronization — workers are generally working on their own deque. (2) Even when steals occur, it is unlikely that multiple workers will steal from the same victim due to randomization. (3) It often has good cache performance, again since workers work on their own deque. (4) It is mostly work-conserving — workers mostly perform useful work. The only time they are not doing useful work is when they steal and most systems bounded number of steal attempts.

3. Intuitions for Tail-Control

This section describes intuitions for reducing the number of requests that miss a target latency. We first review results from scheduling sequential requests to derive inspiration and then show how to adapt them to work stealing for parallel requests.

**Theoretical Results on Minimizing Tail Latency.** The theoretical results for online scheduling of sequential requests on multiple processors indicate that no single scheduling strategy is optimal for minimizing tail latency (Borst et al. 2003; Wierman and Zwart 2012). When a system is lightly loaded or when all requests have similar execution times, a first-come-first-serve (FCFS) strategy is asymptotically optimal (Wierman and Zwart 2012). The intuition is that under these circumstances, the scheduler should minimize the execution time of each request and FCFS does exactly that. However, under heavily loaded systems and when requests have high variability in work, FCFS is not optimal, since under high loads, large requests can delay many small requests by monopolizing processing resources. Under these circumstances, schedulers that prevent large requests from monopolizing processing resources perform better. Examples include a processor sharing scheduler (Kleinrock 1967) that divides up the processing resources equally among all requests and an SRPT-like (shortest remaining processing time) scheduler (Schroeder and Harchol-Balter 2006) that gives priority to small requests.

In interactive services, the request variability in terms of work is often high and the instantaneous load varies as the system executes. When the instantaneous load is low, an approximation of FCFS will work well. However, when the instantaneous load is high, the system should prevent large requests from delaying small requests.

**Baseline Variations of Work Stealing for Interactive Services.** Since traditional work-stealing does not handle multiple requests, we first make a simple modification to work stealing to handle dynamically arriving multiple requests. We add a global FIFO queue that keeps all the requests that have arrived, but have not started executing on any worker. A request is admitted when some worker removes it from the global queue and starts executing it. Now consider how to achieve FCFS in work stealing for multiple parallel requests. To minimize execution time, we want to exploit all the parallelism — which leads to the default steal-first work-stealing policy. We modify steal first for the server setting as follows. When a worker runs out of work on its deque, it still randomly steals work from a victim, but if it cannot find a victim, then it admits a new request. Intuitively, steal-first minimizes the execution time of a request, because it executes requests with as much parallelism as the application and hardware support and only admits new requests when fine-grain parallelism of already admitted requests is exhausted. However, we find it has the same weakness as FCFS — when the system is heavily loaded, a large request can delay many small requests. In fact, this effect is amplified since in steal first for the server setting because a large request with ample parallelism may monopolize all of the processors.

A simple and appealing strategy for high load is admit-first, where workers preferentially admit new requests and only steal if the global queue is empty. Intuitively, admit first minimizes queuing delay, but each request may take longer, since the system does not fully exploit available fine-grain parallelism.

**Motivating Results.** We perform a simple experiment that confirms these intuitions. We create two workloads with only small and large requests, both with ample software parallelism (more inter-request parallelism than cores). The first workload, named “short tail,” contains two kinds of requests: 98% are small requests and 2% medium requests with 5 times more work than the small requests. The second workload, named “long tail,” contains 98% small requests and 2% large requests with 100 times more work than the small requests. Both systems have an average utilization of 65% and the request arrival rate follows the Poisson distribution. Therefore, even though the average utilization is modest, on occasion, the system will be under and over subscribed due to the variability in arrival patterns. We run both workloads 10,000 requests on a 16-core machine (see Section 5 for hardware details).

Figure 2 shows the results. When all requests are about the same length, the short-tail workload shown in hashed red bars, steal first works better since it approximates FCFS and processes requests as quickly as it can. On the other hand with the long-tail workload, steal-first perform poorly due to the relative difference in large and small requests — large requests delay many small requests.

![Figure 2: Neither steal-first nor admit-first always perform best. Each bar plots a target latency and the ratio on a log scale of requests that miss the target latency under admit-first over the requests that miss the target latency under steal first. When the bar is above 1, steal first is better. Below 1, admit-first is better.](image)

**Tail-Control Algorithm Overview.** Since the workloads for interactive services have a range of time varying demand, we design a scheduler that adapts dynamically based on the instantaneous load. Under low load, we use steal-first to maximize parallelism, executing each request as fast as possible. Under high load, we wish to
minimize the impact of large requests on the waiting time of small requests. Therefore, we execute large requests sequentially (thereby giving up on them, making them unlikely to achieve the target latency) and use steal first for the other requests. At any moment, say \( k \) requests of length greater than \( l \) are serialized. Given \( m \) workers, the remaining \( m - k \) workers execute all remaining requests of current length smaller than \( l \) in parallel. Therefore, the policy effectively converts a high-load, high-variable-work instance on \( m \) workers into a low-variable-work instance on \( m - k \) workers. Since steal-first is a good policy for low-variable-work instances, we get good performance for the requests that were not serialized.

Note that we have been playing fast and loose with the term large request. There is no fixed threshold which determines if a request is large or not — this threshold is a function of system load.

A key insight of this paper is that the higher the load, the more aggressively we want to serialize requests by lowering the large request threshold.

The challenges for implementing such a policy include (a) identifying large requests dynamically, because the individual request demand is unknown at arrival time and hard to predict; (b) determining how much and when requests are imposing queueing delay on other requests — thereby determining large requests as a function current load; and (c) dynamically adjusting the policy.

Since we cannot determine the work of a request when it arrives, tail-control conservatively defaults to a steal-first policy that parallelizes all requests. We note that the large requests reveal themselves. Tail-control keeps track of the total work done so far by all active requests. Therefore only later, when a request reveals itself by executing for some threshold amount of work and when system load is high, do we force a request to execute sequentially. As mentioned earlier, the higher the load, the smaller the threshold at which we serialize requests. This approach also has the advantage that higher the load, the earlier we serialize large requests, thereby reducing the amount of parallel processing time we spend on hopeless requests.

The next section presents our offline algorithm, threshold-calculation, that calculates the large request threshold for each value of system load based on the probabilities in a given work distribution. The higher the load, the lower the large request threshold calculated by this algorithm. It also shows how tail-control uses this information, load, and request progress to dynamically control parallelism.

4. Tail-Control Scheduler

This section describes the tail-control scheduler which seeks to reduce the number of requests that exceed a target latency. At runtime, the scheduler uses the number of active requests to estimate system load and continuously calculates each request’s work progress. Tail-control serializes large requests to limit their impact on the waiting time of other requests. The scheduler identifies large requests based on when its work exceeds a large request threshold, which is a value that varies based on the instantaneous system load and the target latency. The higher the load, the lower the threshold and the more large requests tail-control serializes, such that more small requests make the target latency.

The tail-control scheduler has two components: (1) an offline threshold-calculation algorithm that computes a table containing large request thresholds indexed by the number of active requests, and (2) an online runtime that uses steal-first work stealing, but as needed, it serializes large requests based on the large request threshold table.

```plaintext
1 for each number of active request \( q_i \) from 1 to \( q_{\text{max}} \)
2 for each large request threshold candidate \( l \in L \)
3 calculate the length of pileup phase \( T \)
4 calculate the number of large request exceeding target \( \text{miss}_{T_i, l} \)
5 calculate the number of small request exceeding target \( \text{miss}_{S_i} \)
6 \( \text{miss} \) is the \( l \) minimizing total misses \( \text{miss}_{T} + \text{miss}_{S} \) for given \( q_i \)
7 Add \( (q_i, \text{miss}) \) to large request threshold table
```

Figure 3: Algorithm for calculating large request threshold table

4.1 The Threshold-calculation Algorithm

We first explain how we compute the large request threshold table offline. The table is a set of tuples \( \{ q_i : 1 \leq q_i \leq q_{\text{max}} \} \) indicating the large request threshold \( l_{q_i} \) indexed by active requests \( q_i \). Figure 3 presents the pseudo code for calculating the table. For each \( q_i \), the algorithm iterates through the set of candidate large request thresholds \( l \) and calculates the expected number of requests that will miss the target latency if the threshold is set as \( l \). It sets \( l_{q_i} \) as the \( l \) that has the minimum expected requests whose latency will exceed the target.

The algorithm takes as input: (a) a target latency, \( \text{target} \); (b) requests per second, \( qps \); (c) the number of cores in the server \( m \); and (d) the work distribution, such as the examples shown in Figure 1. We derive the maximum number of active requests \( q_{\text{max}} \) from profiling or a heuristic such as twice the average number of active requests based on \( qps \) using queuing theory.

The work distribution is a probability distribution of the work per request, which service providers already compute to provision their servers. Here we use it for scheduling. We represent it as a set of non-overlapping bins: \( \{ \bin{i} : 1 \leq \bin{i} \leq n \} \) where each bin has two values: the probability of a request falling into this bin \( p_i \) and the maximum work of the requests in this bin \( w_i \). Note that the sum of the probability of all bins is one, i.e. \( \sum_{\bin{i}} p_i = 1 \). For each bin in the work distribution, we only know the maximum work of requests falling in this bin. Therefore, we only need consider these discrete values for large request thresholds \( l \), since we pessimistically assume any request that exceeds \( w_i \) will definitely execute until \( w_{i+1} \). Therefore, formally the set of large request threshold candidate set \( L \) is:

\[ L = \{ \bin{i} : 1 \leq \bin{i} \leq n \} \]

We define an instantaneous pileup as a short period of time when the system experiences a high load. To reduce the number of requests that miss the target latency constraint during a pileup, our scheduler limits the processing capacity spent on large requests, to prevent them from blocking other requests.

The key of this algorithm is thus to calculate the expected number of requests that will exceed the target latency during a pileup if the system load is \( q_i \) and the large request threshold is \( l \), for every candidate \( l \in L \) given the other parameters. We first calculate the expected work, summing the expected work in small requests, expected parallel work, and expected sequential work due to requests that exceed \( l \). We then use these quantities to calculate the expected length of a pileup phase and the approximate number of requests whose latency will exceed the target during this pile-up phase. Table 1 defines all the notations for this calculation.

**Basic Work Calculation.** To start, consider the simple expected work calculations that depend only on the work distribution and a large request threshold candidate \( l \). First, we calculate the expected work of a request \( \bar{w} = \sum_{\bin{i}} p_i w_i \). Next, we can calculate the probability that a particular request is large: \( p_l = \sum_{\bin{i} > l} p_i \); and the probability that a request is small: \( p_s = 1 - \sum_{\bin{i} > l} p_i \). Additionally, the expected work per small request can be derived as \( \bar{w}_s = (\sum_{\bin{i} < l} p_i w_i)/(1 - p_l) \).
initial lentire work of small requests may execute in parallel. In addition, work that tail-control may execute in parallel. In tail-control, the request that was detected. This request has the essential work of state again.

The accumulated work at the start of a pileup phase is estimated by

\[
\bar{w}_t - t_{\text{start}} \times \bar{q}_{\text{ps}} \times T
\]

Large Request Target Misses. Once we know the length of the pileup phase, we can trivially derive the number of requests that are expected to arrive during this window, which is \( q_{\text{ps}} \times T \). Now we can calculate the expected number of large requests during the pileup phase. Since the probability of one request being large is \( p_l \), we expect \( p_l(q_{\text{ps}} \times T + q - 1) + 1 \) large requests in total, including the first large request and the other potential large requests that become active or arrive in the interim. In the calculation, we pessimistically assume that large requests always exceed the latency requirement, as we will serialize them. This assumption enforces the serialization of requests to be conservative. Hence, the number of large requests exceeding the target latency \( \text{miss}_s \) is

\[
\text{miss}_l = p_l(q_{\text{ps}} \times T + q - 1) + 1
\]

Small Request Target Misses. We now calculate the number of small requests, \( \text{miss}_s \), that will exceed the target latency, given a particular value of active requests \( q_t \) and large request threshold \( l \).

We optimistically assume that small requests always run with full parallelism. Therefore, if a small request starts executing as soon as it arrives, it will always meet the target latency. Thus, we first calculate how long a small request must wait to miss the target latency. Then we calculate \( x \), which is how many requests must be ahead of this request in order to have this long of a wait. Finally, we calculate how many requests could have \( x \) requests ahead of them during a pileup phase.

We first calculate the average number of cores that are spent on executing the superfluous work of large requests. There are \( \text{miss}_s \) large requests, each in expectation has \( \bar{w}_f \) superfluous work. Therefore, the total amount of superfluous work in the pileup is \( \text{miss}_s \times \bar{w}_f \). Hence, on average, \( \text{miss}_s \times \bar{w}_f / T \) cores are wasted on working on the superfluous work since the pileup lasts for time \( T \). Note that this quantity is very likely to be less than \( m \) assuming the system utilization \( U < m \). Thus, the remaining \( m_s = m - (\text{miss}_s \times \bar{w}_f / T) \) can work on the essential work of both small and large requests. Since we now only consider the essential work, we can think of it as scheduling on a new system with \( m_s \) cores, the same \( q_{\text{ps}} \) as the original system, but the expected work per request is now \( \bar{w}_e \).

For a small request with total expected work \( \bar{w}_s \), its minimum execution time on the remaining \( m_s \) cores is \( \bar{w}_s / m_s \). Given the latency target \( \text{target} \), it will exceed the target if its waiting time is more than \( \text{target} - \bar{w}_s / m_s \). Given \( x \) requests ahead of this request in addition to the first large request that triggered the pileup, then the essential work that needs to execute on these \( m_s \) cores before we get to the small request in consideration is \( (l + \bar{w}_s) / m_s \). Therefore, its waiting time can be estimated as \( (l + \bar{w}_s / x) / m_s \). A request misses the target latency if there are at least \( x \) requests ahead of it where \( x = (\text{target} - \bar{w}_s / m_s - l) / \bar{w}_s \).

Among the \( q_t \) requests that are active when we detect the large request, there are \( q_t = \max(q_t - 1, 0) \) requests that have at least \( x \) requests in addition to the first large request that are ahead of it. These \( q_t \) requests (where \( q_t \) can be 0) will likely overrun the target latency. In addition, we must account for the additional requests that arrive while the queue is still longer than \( x \). Assuming

\[
\text{miss}_s = T \times p_l(q_{\text{ps}} \times T + q - 1) + 1
\]

5 If \( \text{miss}_s \times \bar{w}_f / T \) is greater than \( m_s \), this configuration of \( l \) would lead to system overwork, which means that this particular \( l \) is not a good candidate for \( l_q \). Thus, we simply set \( \text{miss}_s \) to be \( \infty \).
optimistically that requests run one by one in full parallelism, then requests leave at the rate of \(1/(\bar{w}_e/m_s) = m_s/\bar{w}_e\). On the other hand, the system has a request arrival rate of \(qps\). Hence the number of requests in the system decreases with a rate of \(m_s/\bar{w}_e - qps\). This time is how long it takes for the number of active requests to decrease from \(q_t\) to \(x + 1\) is \(1/(m_s/\bar{w}_e - qps)\). During this time, the number of newly arrived requests is then

\[
y_2 = y_t/(m_s/\bar{w}_e - qps) \times qps.
\]

Therefore in total, we have \(y_1 + y_2\) requests that will wait for more than \(x\) requests. Since a request is small with probability \((1-p_t)\), the expected number of small requests that miss the target latency is

\[
miss_s = (y_1 + y_2)(1-p_t) = \max(q_t - 1 - x, 0) \times m_s/\bar{w}_e \times (m_s/\bar{w}_e - qps) \times (1 - p_t)
\]

Thus, we get the expected total number of requests exceeding the target latency for \(q_t\) number of active requests and \(l\) large request threshold is

\[
miss = miss_s + miss_l.
\]

**Discussion.** The complexity of the offline algorithm is \(O(q_{max} \times n^2)\), where \(n\) is the number of bins in the request work distribution. Note that we are conservative in serializing large requests. We estimate the execution time for a small request to be \(\bar{w}_e/m_s\), which is minimum by assuming that it can occupy all available cores \(m_s\) and execute with linear speedup. However, some requests (especially small requests) may not have enough parallelism to use all \(m\) cores. The system is thus conservative in serializing large request, because this calculation may overestimate the waiting time before the small requests miss the target latency. However, if the system profiler also profiles the average parallelism of requests in all bins (or average critical-path length of requests in all bins; these are equivalent), we can incorporate this information into the calculation easily to perform more accurate calculations, potentially leading to more aggressive parallelization of large requests.

**Example Output.** Figures 4 shows the output from threshold-calculation with input of a lightly loaded work distribution with \(qps\) of 1200 and three different target latencies. The x-axis is the number of active requests, and the y-axis is the large request threshold. Each curve plots the output for a given latency.

![Table 4: Large request threshold table output by threshold-calculation with input of a lightly loaded work distribution with qps of 1200 and three different target latencies.](image)

4.2 Extending Work-Stealing with Tail-Control

This section describes how to use the large request table to implement the tail-control strategy in a work-stealing runtime. While we implemented our system in TBB (Reinders 2010), this approach is easy to add to other work-stealing runtimes.

**Scheduler Overview.** As with the basic work-stealing scheduler, tail-control is a distributed scheduler, and each worker maintains its own work deque. A shared global FIFO queue contains requests that have arrived but not yet being admitted. For the most part, the tail-control behaves like a steal-first. Only when the system identifies a large request, does its behavior diverge from steal-first with respect to the identified large request — it serializes the large request’s remaining execution. Specifically, tail-control performs the following additional actions: 1) it tracks the number of active requests in the system to estimate system load; 2) it tracks processing time spent thus far on each executing request; 3) it identifies a large request based on processing time and serializes its remaining execution.

Tail-control performs actions 1) and 2) in a distributed fashion. Each worker accumulates processing time for its active request. Whenever a worker finishes a request completely, it decrements a shared global counter that maintains the number of executing requests in the system. Overhead for action 1) is minimal since it occurs infrequently. Overhead for action 2) is amortized against the cost of steals, since it needs to be done only between steals.

Action 3) requires more coordination. To identify a large request, it is necessary to collect processing time scattered across all the workers that are executing a request, which can incur overhead. Ideally, we would like to check each request’s processing time frequently so that a request does not execute any longer once it exceeds the large request threshold. On the other hand, we do not want to burden executing workers with the overhead of frequent checks. Thus, we piggyback the execution of action 3) on a thief looking for work to do. Whenever a worker runs out of work, before it steals again, it first computes processing time of executing requests on all other workers, computes the accumulated processing time for each request, and marks any requests that exceed the threshold. Once a request is marked as large, it needs to be serialized. We modify the stealing protocol to mark large requests lazily. If a thief tries to steal from a deque with tasks that belong to a large request, the thief simply gives up and tries to steal somewhere else. Again, the overhead is amortized against the cost of steals.

**Implementation of Tail-Control.** Figure 5a shows the pseudo code for the top-level scheduling loop that a worker executes in tail-control. The bold lines mark the code that is only necessary for tail-control but not steal-first. During execution, a worker always first tries to pop tasks off its local deque as long as there is more work in the deque (lines 5–8). When a worker’s deque becomes empty, it tries to resume the parent of its last executed task if it is ready to be resumed (lines 9–12). In short, a worker always executes tasks corresponding to a single request for as long as work can be found locally (lines 5–12). As part of the bookkeeping in tail-control, if the last executed task has no parent (i.e., root), the worker checks for the completion of a request and decrements the active-request counter if appropriate (lines 13–16).

Once the control reaches line 18, a worker has exhausted all its tasks locally, and it is ready to find more work. Before performing...
a random steal, a worker performs the necessary bookkeeping to accumulate the processing time it spent on last request lines 18–21). It updates its curr_request field to reflect the fact it is not working on a request. This field keeps track of the worker’s current request, and is read by a thief in check_pileup_phase when it accumulates processing time of an executing request.

Then the worker calls check_pileup_phase (line 23) to identify large requests and mark them as not stealable. It then becomes a thief and tries to steal randomly (line 24). If no currently executing requests are stealable, try_random_steal returns NULL, and tail-control admits a new request (line 26) and assigns it to the worker. Regardless of how the worker obtains new work, it updates its curr_request and the start time for this request (lines 27–30). When it completes this work, it loops back to the beginning of the scheduling loop to perform bookkeeping and find more work. In a server system, this scheduling loop executes continuously.

The tail-control implementation has the same high-level control flow as in steal-first, since except for the pileup phase, it follows the steal-first policy. Moreover, the difference between steal-first and admit-first is simply in the order in which a thief finds new work. By switching the sequence of stealing and admitting, i.e., switching head at the cost of some loss in accuracy in calculating the requests’ processing time as values. A hashtable is needed because there could be multiple workers working on the same request. The processing time of an executing request is essentially the time accumulated on the request thus far, and the additional processing time elapsed on an executing worker since the last time it updated its start time. lines 16–23 does exactly that calculation.

The check_pileup_phase function implements the detection and marking of large requests. When a thief executes this function, it first evaluates the current load by getting the active-request count, and uses it to index the large request threshold table (lines 10–11). With the given threshold, it then examines all workers and accumulates the processing time of all executing requests into a local hashtable (lines 15–27). The hashtable uses requests as keys and stores their processing time as values. A hashtable is needed if (req_map.find(req) == 6) 
new_proc_time = req->get_proc_time() + added_time;
else 
new_proc_time = req_map.find(req) + added_time;
req_map.insert(req, new_proc_time);
// mark a request that exceeds threshold
if (new_proc_time > req_threshold) 
for (Task *req : req->set_stealable(false);
25 } 
26 } 
27 } 
28 }
31 } // end of while (more_work_to_do())
32 } // end of scheduling loop

(a) The main loop. Tail-control adds only the bold lines to steal-first.

(b) Helper routines. Tail-control adds only the bold lines to steal-first.

5. Experimental Evaluation

We now present an empirical evaluation of the tail-control strategy as implemented in the TBB work-stealing scheduler. We compare tail-control to steal-first and admit-first, and show that tail-control can improve over all baselines. Our primary performance metric is the number of requests that exceed the latency target.
**Experimental Setup.** Experiments were conducted on a server with dual eight-core Intel Xeon 2.4Ghz processors with 64GB main memory. The server runs Linux version 3.13.0. When running experiments, we disable processor throttling, processor sleeping, and hyper-threading. The tail-control algorithm is implemented in the Intel Thread Building Block (TBB) library (Reinders 2010), version 4.3.

We evaluate our strategy on several different workloads that vary along three dimensions: (1) different work distributions (two real-world workloads and several synthetic workloads), (2) different inter-arrival time distributions (Poisson distribution with different means and long-tail distributions with varying means and standard deviations), and (3) different request parallelism degrees (embarrassingly parallel requests and requests with parallelism less than the number of cores).

First, we evaluate requests with different work distributions. The two real-world work load distributions are shown in Figure 1. Henceforth, we shall refer to them as the Bing workload and the finance workload, respectively. In addition, we also evaluate synthetic workload with log-normal distributions, referred as log-normal workload. A log-normal distribution generates random variables whose logarithm is normally distributed. Thus, it has a longer tail than a normal distribution and represents the characteristics of many real-world workloads. For all the workloads, we use a simple program to generate work — the program performs a financial calculation which estimates the price of European-style options with Black-Scholes Partial Differential Equation. Each request is parallelized using parallel-for loops. Note that the particular computation performed by the workload is not important for the evaluation of our strategy; any computation that provides the same workload distribution should provide similar results.

Second, we evaluate requests with different arrival distributions. In particular, to randomly generate the inter-arrival time between requests, we use two distributions: a Poisson distribution with a mean that is equal to $1/\text{qps}$, and a log-normal distributions with a mean equal to $1/\text{qps}$ and varying standard deviations. In other words, for all the distributions, the requests are generated in an open loop at the rate of qps (queries per second). We use 100,000 requests to obtain single point in the experiments.

In addition, we evaluate the effect of requests with different parallelism degrees. In addition to the embarrassingly parallel requests generated by parallel-for loops of Black-Scholes, we also intentionally insert sequential segments to make requests less parallel. In particular, for each request, we add sequential segments with total length of 10% of its work. By doing so, the parallelism of requests is less than 10, which is smaller than the 16 available cores.

Finally, we explore some additional properties of tail-control. We test its performance when the input work distribution is inaccurate and differs from the actual work distribution. We also present the improvement of tail-control in terms of system capacity. We conduct comparison with two additional baseline algorithms to position the performance of tail-first, admit-first, and tail-control. Lastly, we present a result trace to unveil the inner workings of tail-control.

### 5.1 Different Work Distributions

We first compare the performance of tail-control to the admit-first and steal-first baseline work-stealing policies described in Section 3, with various loads and target latencies. For all the experiments in this subsection, requests arrive according to Poisson distribution with varying mean inter-arrival times. Figures 6, 7 and 8 show the results on work distribution for finance, Bing, and log-normal workload distribution respectively, where each graph in a figure shows the result for one setting (light, medium or heavy when going from left to right). Within each graph, we compare the policies for five latency targets. As we go from left to right, the target latency increases, and thus all policies improve in performance from left to right. Now we can look at the specific results for each workload.

**Bing Workload.** From Figure 6, we can make three observations. First, for the Bing workload, admit-first performs better than steal-first in most cases. The reason is, as seen in Figure 1a, the Bing workload has high variability between the work of the largest vs. the smallest requests. As discussed in Section 3, steal-first is likely to perform worse in this situation since it allows very large requests to monopolize the processing resources, potentially delaying a high number of small and medium requests. Second, as target latency increases, steal-first strategy’s performance in comparison to admit-first improves, finally overtaking it slightly for the longest latency target. As the target latency increases, the impact on waiting time due to executing large requests reduces and steal-first starts reaping the benefits of exploiting intra-request parallelism; therefore, it starts performing better than admit-first. This observation reflects the trade-off between steal-first and admit-first and confirms that they cannot perform well in all settings. Finally, tail-control provides consistently fewer missed requests across the three load settings and target latencies — it has a relative improvement of 35% to 58% over admit-first and of 24% to 65% over steal-first in all settings. In particular, tail-control has higher improvement at the harsher setting — when the system load is heavier and the target latency is shorter. It limits the impact of large requests by serializing the large requests. However, it still reaps the benefit of intra-request parallelism since it parallelizes short and medium requests and processes them quickly.

**Finance Workload.** Figure 7 shows the results for the finance workload, and the results reinforce the observations made above. The finance workload has less variability in its work distribution compared to the Bing workload; therefore, as our intuition indicates, steal-first performs better than admit-first. Since large requests are not that much larger than other requests, they do not impact the other requests as significantly. Therefore, steal-first is a good strategy for this workload. In the settings under the heavy load and the two longest target latencies, tail-control performs exactly the same schedule as steal-first, since the calculated thresholds from threshold-calculation are high resulting in no request being serialized. This result indicates that when steal-first is the best strategy, tail-control essentially defaults to it and does not gratuitously increase target latency misses by serializing requests unnecessarily. In addition, even though tail-control has a higher overhead due to book-keeping, the overhead is lightweight and not significant enough to affect performance and increase miss ratio. Moreover, even on this workload where steal-first is already a good strategy, tail-control still significantly improves the miss ratio in some cases. For instance, under medium load with the longest target latency, tail-control provides 30% improvement. These results indicate that tail-control performs well under different types of workload characteristics and loads.

**Log-normal Workload.**

In Figure 8, we generate the work of a request using a log-normal distribution with mean of 10 ms and standard deviation of 13 ms. We selectively choose the mean of 10 ms in order to directly compare the results with that of Bing workload, as the mean work of Bing workload is also 10 ms. The standard deviation is chosen, such that the log-normal work distribution has a slightly shorter tail.

---

4The target latencies are chosen as the 97.5%, 98.5%, 99%, 99.5% and 99.75% tail latencies of steal-first in order to provide evidence that tail-control performs well under varying conditions.
than the Bing workload, but a longer tail than finance workload.
For a log-normal workload, steal-first performs better than admit-
first when target latency is long and slightly worse when the target
latency is short. In this setting, tail-control consistently outperforms
steal-first and admit-first with improvement from 17% up to 60%.

5.2 Request with Sub-Linear Speedup
Due to space limitations, henceforth, we focus on one load (the
heavy load of qps 1200) and two work distribution (Bing and log-
normal workloads). In all the previous experiments, requests are
embarrassingly parallel with near linear speedup. Here we evaluate
how well tail-control performs, when increasing the span and
decreasing the parallelism degree of requests. In particular, in Fig-

Figure 6: Bing workload with three different load settings and Poisson arrival. The x-axis shows different target latencies from shorter to longer from left to right. The y-axis shows the target latency miss ratio. The table below each figure shows tail-control's relative improvement over steal-first and admit-first for a given latency.

Figure 7: The finance workload results with the same figure and table configuration as in Figure 6.

Figure 8: The log-normal workload results with the same figure and table configuration as in Figure 6.
Figure 9: The Bing workload and a Poisson arrival distribution with mean qps of 1200 for requests with long sequential segments and sub-linear speedup.

Figure 10 shows an experiment with log-normal workload and arrival distribution. The experiment in Figure 11 has the same setting as Figure 8(c) with target latency of 28 ms. In addition to the tail-control with the correct input work distribution, we also run tail-control using inaccurate input work distributions. In particular, we slightly alter the input work distribution by changing the standard deviation while keeping the same mean work. Because the input distribution is inaccurate, the calculated thresholds are also inaccurate.

From Figure 11, we can see that when the input inaccuracies are small, for example standard deviation of 10 ms and 17 ms instead of the true 13 ms, tail-control still has comparable performance. However, the improvement of tail-control decreases when the error increases. Moreover, tail-control is less sensitive to a larger inaccurate standard deviation than a smaller one.

5.3 Different Arrival Distributions

Note that in all the previous experiments, requests arrive according to a Poisson arrival process. In this set of experiments, we vary the inter-arrival time distribution and select one with larger variance than Poisson distribution, namely log-normal distribution. Figure 10 shows an experiment with log-normal workload and log-normal arrival distribution with mean inter-arrival time of 0.83 ms (qps of 1200) and standard deviation of 1.09 ms. Note that this experiment is constructed such that it has almost the same parameters as Figure 6(c), except for a smaller parallelism degree. By comparing the two experiments, we can see that the relative trend among different algorithms remains the same. However, tail-control has less improvement over steal-first and admit-first from 17% to 46%. The results for log-normal workload is similar to that of Bing workload.

Tail-control improves less in this setting, because small requests are not able to utilize all the available core, even when large requests are serialized. Moreover, the large request is also not going to monopolize the entire system due to its sequential segments. Note that in the extreme case where all requests are sequential, all the three algorithms will be the same. The improvement that tail-control provides depends on the parallelism of requests. As request parallelism degree increases, tail-control provides more benefit.

5.4 Inaccurate Input Work Distribution

Note that tail-control calculates large request threshold using request work distribution as input. Hence, it is natural to ask how tail-control performs when the workload distribution differs from the input workload distribution for the threshold-calculation algorithm. That is, we experimentally evaluate how brittle the tail-control scheduler is when provided with a somewhat inaccurate input workload distribution. The experiment in Figure 11 has the same setting as Figure 8(c) with target latency of 28 ms. In addition to the tail-control with the correct input work distribution, we also run tail-control using inaccurate input work distributions. In particular, we slightly alter the input work distribution by changing the standard deviation while keeping the same mean work. Because the input distribution is inaccurate, the calculated thresholds are also inaccurate.

From Figure 11, we can see that when the input inaccuracies are small, for example standard deviation of 10 ms and 17 ms instead of the true 13 ms, tail-control still has comparable performance. However, the improvement of tail-control decreases when the error increases. Moreover, tail-control is less sensitive to a larger inaccurate standard deviation than a smaller one. This is because when the standard deviation of the profiling workload is small, tail-control less aggressively serialize requests and becomes closer to steal-first. In contrast, when the standard deviation of the profiling workload is large, tail-control is slightly more aggressive than it should be. But this only causes a few larger requests unnecessarily serialized. Since the probability of large requests are small, it affects the performance of tail-control only slightly. However, when the input is significantly different from the actual work distribution (for example when the mean work is significantly inaccurate), tail-control could have worse performance than steal-first and admit-first. This happens when tail-control very aggressively serialize requests, even when the number of requests is less than the number of cores.

In summary, tail-control does require a relatively accurate work distribution, but it does not need to be exact. If the system load changes significantly over time, the system operator needs to profile the new work distribution or qps. In fact, because the calculation of tail-control is very fast, an interactive service could sample its workload and let tail-control to calculate large request thresholds and to perform scheduling adaptively in real-time.

5.5 Increased System Capacity

The benefits of tail-control can be used to increase server capacity, thereby reducing the number of servers needed to run a particular aggregate workload as each server can run a higher number of requests per second. For instance, say we have $m$ servers and for a particular workload, steal-first enables 99.5% of the requests to meet the target latency. Here, tail-control can provide the same guarantee (99.5% requests meet the same latency target) with a higher system load. Figure 12 provides evidence for this claim. It shows the maximum load for several target latencies at the 99.5-
percentile. For instance, at target latency 20.00 ms, tail-control sustains 880 qps compared to 800 for steal-first and 775 for admit-first, showing 10% capacity increase over the best of the two.

5.6 Comparison with Additional Baseline Algorithms

Now we provide comparison with two additional baseline algorithms. The first one is denoted as default-TBB, as it is the algorithm of the default TBB implementation. Note that in both steal-first and admit-first, requests are first submitted to a global FIFO queue and workers explicitly decide whether and when to admit requests. In contrast, default-TBB implicitly admit requests by submitting requests to the end of a random worker’s deque. After doing so, workers can act as if there is a single request and they only need to randomly steal when running out of work. This strategy may have smaller overhead than the global FIFO queue. However, now requests are admitted randomly instead of in the FIFO order, so it may cause a waiting request starve in the worst case. We measure the performance of default-TBB for all three workloads with Poisson arrival and observe that default-TBB has comparable or worse performance than admit-first in most settings. This is because default-TBB acts similar to admit-first, but it admits requests with a random order. A request that has waited for a very long time could potentially be admitted later than a request that just arrives. This could significantly increases the latency of some requests when the system is busy, which increases the request target miss ratio. On the other hand, without any global queue default-TBB has smaller overheads. This allows system to process requests slightly faster and hence reduce the length of pileup phase. Thus, in such cases, default-TBB performs slightly better than admit-first. Even in these settings, tail-control still outperforms default-TBB by at least 4%. Due to space limit, we present two representative settings in Figure 13 and Figure 14.

The second baseline algorithm marked as TC-Clairvoyant in Figure 13 and Figure 14 shows the lower bound of tail-control, because this scheduler utilizes the exact work of each request to perform tail-control strategy. More specifically, we modified TBB and application server to mark each request with its actual work, when it is submitted to the global FIFO queue. For threshold calculation, we adjust the threshold calculation algorithm described in Section 4 for TC-Clairvoyant to account for the fact that a serialized large request under TC-Clairvoyant is never executed on more than one processor. During online execution, the TC-Clairvoyant scheduler knows the exact work of each request even before its execution (thusclairvoyant). Hence, unlike tail-control, TC-Clairvoyant knows whether a request is a large request and can directly serialize it without any parallel execution. Thus, TC-Clairvoyant plays the lower bound of non-clairvoyant tail-control, as it can more effectively limit the impact of large request if the actual work of each request is clairvoyant to the scheduler.

From Figure 13 and Figure 14, we can see that TC-Clairvoyant and tail-control have comparable performance when the target latency is long, while TC-Clairvoyant improves much more when the target latency is short. This is because to minimize for long target...
latency, only the very large requests need to be serialized. Because
the work distribution has a relatively long tail, we can more eas-
ily distinguish very large requests from other medium or small re-
quests. However to minimize for short target latency, we may need
to serialize some medium requests as well and it takes medium re-
quest to execute for most of its work before we can distinguish
them from small requests. The TC-Clairvoyant also shows that our
tail-control strategy can take advantage of request work and further
improves the performance, if this information is available or if there
is a efficient algorithm to predict request work.

5.7 The Inner Workings of Tail-Control

We now look a little deeper into the inner workings of tail-
control. Figure 15 shows the trace of the execution of the same
workload under both steal-first and tail-control in a 0.8 seconds window (Bing workload).

We build on work

Figure 15: Number of active requests (lower part) and request
latency (upper part) traces of the same workload under steal-first and tail-control in a 0.8 seconds window (Bing workload).

5. Related Work

Multiprogrammed Parallelism for Mean Response Time. For
multiprogrammed environments, most prior work has looked into
reducing mean response time (Feitelson 1994; McCann et al. 1993;
Agrawal et al. 2006, 2008; He et al. 2008) or some other fairness
metric. In this work, the characteristics of the jobs are unknown
in advance; the scheduler learns job characteristics and adjusts
the degree of parallelism as jobs execute. For example, Agrawal
et al. (Agrawal et al. 2006, 2008) and He et al. (He et al. 2008)
use dynamic parallelism demand and efficiency feedback in work-
sharing and work-stealing. In contrast, data center environments
collect extensive telemetry data on interactive services, including
the workload distribution, latencies, and load, which they use for
server provisioning. Our offline analysis exploits this information
in order to make online decisions about which requests to serialize.

Interactive Server Parallelism for Tail Latency. Jeon et al.
(2014) and Kim et al. (Kim et al. 2015a) predict the service
demand of web search requests using machine learning to execute
the (predicted) small requests sequentially to save resources, and
to parallelize large requests to reduce their tail latency. This prior
work exploits the large variance of request service demand to re-
duce tail latency, but they have two limitations: (1) request service
demand for many interactive services may not be predictable, thus
is unknown to the scheduler a priori; (2) their request parallelism
decision is made regardless of the system load and workload char-
acteristics.

Haque et al. (Haque et al. 2015) present a few-to-many parallel-
ism technique which dynamically increases request parallelism
degree during execution. Their system completes small requests se-
quently to save resources, parallelizes large requests to reduce
tail latency, and uses the system load and workload character-
istics to decide how to increase request parallelism degrees. Their
work differs from ours in three ways. (1) The scheduling objective is
different: they reduce tail latency while we reduce the number of
requests that miss a target latency. (2) Because the scheduling ob-
jectives differ, the key techniques differ substantially: They provide
more parallelism to large requests to reduce their latency, at the cost
of increasing latency of small requests. In contrast at high load, we
 sacrifice few large requests to increase the chance that other re-
quests meet the latency target. (3) They embed the scheduler in the
application, whereas we adapt a work-stealing runtime that works
for any dynamic multithreaded programs.

7. Conclusion

This paper presents the design and implementation of the tail-
control work-stealing scheduler for optimizing the number of re-
quests that meet a target latency for multiple concurrent requests.
Our experiments indicate that our prototype work-stealing sched-
uler using TBB is very effective for highly parallel interactive
workloads. Although no current interactive service uses work steal-
ning, as far as we are aware, the work stealing framework is appeal-
ing because it supports the most general models of both static and

Parallelizing Single Job to Reduce Latency. We build on work
stealing and related techniques that adapt parallelism to run-time
variability and hardware characteristics (Blagojevic et al. 2007;
Curtis-Maury et al. 2006; Jung et al. 2005; Ko et al. 2002; Lee
et al. 2010; Pusukuri et al. 2011; Suleman et al. 2008; Wang and
O’Boyle 2009). Prior work focuses on reducing the execution time
of a single job; our work in contrast focuses on reducing the number
of requests whose latency exceeds a predefined target in a server
environment with multiple jobs.
dynamic parallelism. To use our framework, services need only to express the parallelism in individual requests. The benefits of the tail-control scheduling strategy include improved user experience with more consistent responsiveness and increased system capacity for interactive services.

Acknowledgment

This research was supported in part by NSF grants CCF-1527692, CCF-1337218, CCF-1218017 and CCF-1155036, and by ONR grants N000141310800. The authors thank anonymous reviewers for their suggestions on improving this paper.

References


116–125, 2011.