Work Stealing for Interactive Services to Meet Target Latency *

Jing Li* Kunal Agrawal* Sameh Elnikety† Yuxiong He†
I-Ting Angelina Lee* Chenyang Lu* Kathryn S. McKinley†

*Washington University in St. Louis †Microsoft Research
{li.jing, kunal, angelee, lu}@wustl.edu, {samehe, yuxhe, mckinley}@microsoft.com

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 target latency. 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 admit 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; DeCandia et al. 2007; 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, i.e., the latency of requests in the high percentiles, such as 99^{th} 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; Jalaparti et al. 2013; Kim et al. 2015b). The amount of work per request varies widely. The work of the 99^{th} percentile requests is often larger than the median by orders of magnitude, ranging from 10 to over 100 times larger. Moreover, the workload is highly parallelizable — it has **inter-request parallelism** (multiple distinct requests execute simultaneously) 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 (Intel 2013; Frigo et al. 1998; Danaher et al. 2006; Leiserson 2010; Lee et al. 2013), Habanero (Barik et al. 2009; Cavé et al. 2011), Java Fork/Join (Lea 2000; Kumar et al. 2012), 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).

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). It is a distributed scheduling strategy with low scheduling overheads. The goal of traditional work stealing is to deliver the

* This work and was initiated and partly done during Jing Li's internship at Microsoft Research in summer 2014.

PPoPP '16 March 12-16, 2016, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/2851141.2851151

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 strategy'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 meet 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 sacrifice depends on the request's progress, the target latency, and current load. Serializing large request can limit their impact on queuing delay of waiting small requests and enabling more requests to meet the target latency. However, 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 work 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 also 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:

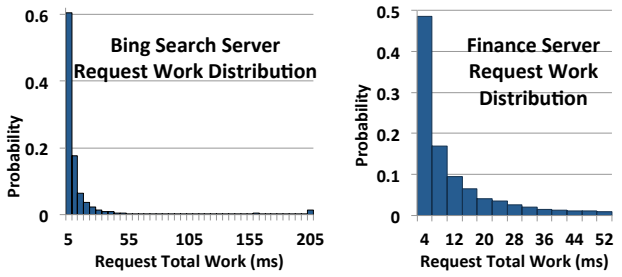
1. We design an offline **threshold-calculation algorithm** that takes as input the target latency, the work demand distribution, and the number of cores on the server, and calculates a *large request threshold* for every value of the instantaneous load. It generates offline a table indexed by system 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 thus far. Most bookkeeping is done distributedly, which amortizes its overhead with steal attempts. Our modified work-stealing scheduler add no constraints on the programming model and thus can schedule any dynamic multithreaded program.
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 defines terminology used throughout the paper, characterizes interactive services and available intra-request parallelism, and lastly reviews work stealing basics.

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 request work distributions for Bing and a finance server. Many requests are small and the smallest are more than 40 times smaller than the largest for Bing and 13 for the finance server. Third, the requests often have internal parallelism and each request can potentially utilize multiple cores.



(a) Bing search (b) Finance server

Figure 1: Work distribution of two interactive services: Bing search server (Kim et al. 2015a) and an option pricing finance server (Ren et al. 2013). Note that in Bing search server the probabilities of requests with work between 55ms to 200 are small but larger than zero and total probability of these requests is around 3.5%.

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). The following properties of work stealing contribute to making it efficient. (1) It requires little synchronization — workers are generally working on their own deques. (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 deques. 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 the number of steal attempts is bounded.

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 work is often highly variable, 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.

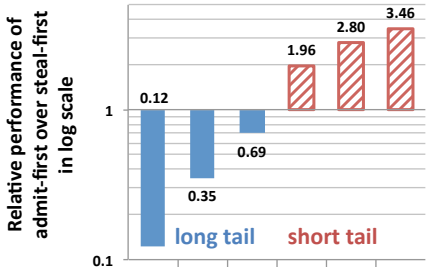
Baseline Variations of Work Stealing for Interactive Services. Since traditional work-stealing does not handle multiple requests, we first make a simple modification to work stealing to handle dynamically arriving multiple requests. We add a global FIFO queue that keeps all the requests that have arrived, 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 victims, but if there is no stealable work, 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 ones. The second workload, named "long tail", contains 98% small requests and 2% large requests with 100 times more work than the small ones. 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 performs poorly due to the relative difference in large and small requests — large requests delay many small requests.



Target latency (ms)	1013	1515	1904	51.83	76.93	85.48
AF: Admit-First miss ratio	5.0%	1.0%	0.5%	9.8%	2.8%	1.7%
SF: Steal-First miss ratio	40.7%	2.9%	0.7%	5.0%	1.0%	0.5%
Relative performance = AF / SF	0.12	0.35	0.69	1.96	2.80	3.46

Figure 2: Neither steal-first nor admit-first always performs 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 those that miss the target latency under steal-first. When the bar is above 1, steal-first is better. Below 1, admit-first is better.

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 work greater than l are serialized. Given m workers, the remaining $m - k$ workers execute all the remaining requests of work smaller than l in parallel. Thus, 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 work is unknown at arrival time and hard to predict; (b) determining how much and when requests are imposing queuing delay on other requests — thereby determining large requests as a function of 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 the 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 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 their 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, but as needed, serializes large requests based on the large request threshold table.

```
1  for each number of active request q_t from 1 to q_max
2      for each large request threshold candidate l ∈ L
3          calculate the length of pileup phase T
4          calculate the number of large request exceeding target miss_l
5          calculate the number of small request exceeding target miss_s
6      l_{q_t} is the l minimizing total misses miss_l + miss_s for given q_t
7      Add (q_t, l_{q_t}) 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 $\{q_t : 1 \text{ to } q_{max} \mid (q_t, l_{q_t})\}$ indicating the large request threshold l_{q_t} indexed by the number of active requests q_t. Figure 3 presents the pseudo code for calculating the table. For each q_t, 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_t} as the l that has the minimum expected requests whose latency will exceed the target.

The algorithm takes as input: (a) a target latency, $target$; (b) requests per second, rps; (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_{max} from profiling or a heuristic such as twice the average number of active requests based on rps 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: $\{\text{bin } i : 1 \text{ to } n \mid (p_i, w_i)\}$, 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. $\sum_{\text{bin } i} p_i = 1$. For each bin in the work distribution, we only know the maximum work of requests falling in this bin. Therefore, we only need to 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 is $L = \{i : 1 \text{ to } n \mid w_i\}$.

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 tail 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 miss the target latency during a pileup if the system load is q_t and the large request threshold is l, for every candidate $l \in L$ given the other parameters. We first calculate the expected work of 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 and the approximate number of requests whose latency will exceed the target. Table 1 defines 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 $\bar{w} = \sum_{\text{bin } i} p_i w_i$. Next, we can calculate the probability that a particular request is large: $p_l = \sum_{w_i > l} p_i$; and the probability that a request is small: $p_s = 1 - p_l$. Additionally, the expected work per small request can be derived as $\bar{w_s} = (\sum_{w_i \le l} p_i w_i)/(1 - p_l)$.

Now consider the **expected essential work** per request $\bar{w_e}$ — the work that tail-control may execute in parallel. In tail-control, the entire work of small requests may execute in parallel. In addition, because a large request is only detected and serialized after being processed for l amount of work, hence every large request's initial l amount of work will execute in parallel given sufficient resources.

$$\bar{w_e} = \sum_{w_i \le l} p_i w_i + \sum_{w_i > l} p_i l$$

We call this essential work since processing this work quickly allows us to meet target latency and detect large requests.

The remaining work of a large request exceeding l is serialized. We deem this work superfluous since these requests are very likely

Symbol	Definition
$target$	Target Latency
rps	Request per second of a server
m	Number of cores of a server machine
p_i	Probability of a request falling in bin i
w_i	Work of a request falling in bin i
q_t	Instantaneous number of active requests at time t
l	Large request threshold
L	Set of potential large request thresholds
$\bar{w}$	Average work per request
p_l	Probability of a request being a large request
$\bar{w_s}$	Expected work per small request
$\bar{w_f}$	Expected superfluous work per large request
$\bar{w_e}$	Expected essential work per request

Table 1: Notation Table

to miss the target latency and therefore this work will not contribute to the scheduling goal. We calculate the expected amount of work that is serialized per large request, formally denoted as the **expected superfluous work** per large request $\bar{w_f}$:

$$\bar{w_f} = \frac{\sum_{w_i > l} p_i(w_i - l)}{\sum_{w_i > l} p_i}$$

Pileup Phase Length. Now we calculate the length of a pileup when the number of active requests is q_t and the large request threshold is l. We define the pileup start time as the time when a large request is first detected in the non-pileup state. The pileup ends when the large request that caused the pileup is completed, all the work that has accumulated in the server due to this overload is also completed, and the system reaches a steady state again.

We can approximately calculate the amount of accumulated work at the start of a pileup. First, let us look at the first large request that was detected. This request has the essential work of l and the remaining expected superfluous work of $\bar{w_f}$; it has a total of $\bar{w_f} + l$ work in expectation. In addition, each of the remaining $q_t - 1$ active requests has an expected work of $\bar{w}$. Thus, the total accumulated work at the start of a pileup phase is estimated by $\bar{w_f} + l + (q_t - 1)\bar{w}$.

We also need to account for the work that arrives during the pileup. We define the average utilization of the server as $U = \bar{w} \times rps$. We assume that $U < m$ since otherwise the latency of the requests will increase unboundedly as the system gets more and more loaded over time. In expectation, the new requests that arrive during the pileup have a utilization of U, which means that we need U cores to process them. Therefore, tail-control has $m-U$ remaining cores with which to execute the already accumulated work. Thus, to entirely finish the accumulated work, it will take about $\frac{(\bar{w_f}+l)+(q_t-1)\bar{w}}{m-U}$ time.

Now consider the first large request in a pileup: it executes in parallel for l amount of work and then executes sequentially for $\bar{w_f}$ time. Its minimum completion time would be $l/m + \bar{w_f}$, which assumes that it runs on all the cores before being detected and serialized. Thus, the expected length of the pileup phase T is the maximum of the time to entirely finish the accumulated work and the time to finish the first large request:

$$T = \max\left\{\frac{(\bar{w_f} + l) + (q_t - 1)\bar{w}}{m - U},\ l/m + \bar{w_f}\right\}$$

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 $rps \times T$. 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 $p_l(rps \times T + q_t - 1) + 1$ 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 target latency, 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 $miss_l$ is

$$miss_l = p_l(rps \times T + q_t - 1) + 1$$

Small Request Target Misses. We now calculate the number of small requests, $miss_s$, that will miss 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 $miss_l$ large requests, each in expectation has $\bar{w_f}$ superfluous work. Therefore, the total amount of superfluous work in the pileup is $miss_l \times \bar{w_f}$. Hence, on average, $miss_l \times \bar{w_f}/T$ 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 $U < m$.[1] Thus, the remaining $m_s = m - (miss_l \times \bar{w_f}/T)$ cores 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 rps 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 target latency $target$, it will exceed the target if its waiting time is more than $target - \bar{w_s}/m_s$. 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 $l + \bar{w_e}x$. Therefore, its waiting time can be estimated as $(l + \bar{w_e}x)/m_s$. A request misses the target latency if there are at least x requests ahead of it where $x = (target \times m_s - \bar{w_s} - l)/\bar{w_e}$.

Among the q_t requests that are active when we detect the large request, there are $y_1 = \max(q_t - 1 - x, 0)$ requests that have at least x requests in addition to the first large request that are ahead of it. These y_1 requests (where y_1 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 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 rps. Hence the number of requests in the system decreases with a rate of $m_s/\bar{w_e} - rps$.[2] Thus, it takes $y_1/(m_s/\bar{w_e} - rps)$ amount of time for the number of active requests to decrease from q_t to $x + 1$. During this time, the number of newly arrived requests is then $y_2 = y_1/(m_s/\bar{w_e} - rps) \times rps$.

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 $(1 - p_l)$, the expected number of small requests that miss the target

[1] If $miss_l \times \bar{w_f}/T$ is greater than m, this configuration of l would lead to system overrun, which means that this particular l is not a good candidate for l_{q_t}. Thus, we simply set $miss_s$ to be ∞.

[2] Note that this rate is positive in order for the l in consideration to be a good candidate for l_{q_t}. Thus if the rate is not positive, we again set $miss_s$ to be ∞

latency is

$$miss_s = (y_1 + y_2)(1 - p_l)$$
$$= \max(q_t - 1 - x, 0) \times \frac{m_s/\bar{w_e}}{(m_s/\bar{w_e} - rps)} \times (1 - p_l)$$

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 $\bar{w_s}/m_s$, 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 requests, because this calculation may overestimate the waiting time before the small requests miss the target latency. If the system profiler also profiles the average parallelism of requests in all bins (or the average span[3] 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 an example output from threshold-calculation with input of the same work distribution and rps but three different target latencies. Let's first examine a single curve, say the one with 14.77ms target latency. As the number of active requests increases (i.e., higher instantaneous system load), the large request threshold decreases, indicating that the system serializes requests more aggressively. When examining all three curves collectively, Figure 4 illustrates how the relationship between the threshold and active number of requests varies according to the target latency. As the target latency increases, the curves shift towards the right, showing that when given the same number of active request, the system can increase the large request threshold because of the longer target latency. In other words, given the same instantaneous load, the system is less aggressive about serializing requests when the target latency is longer.

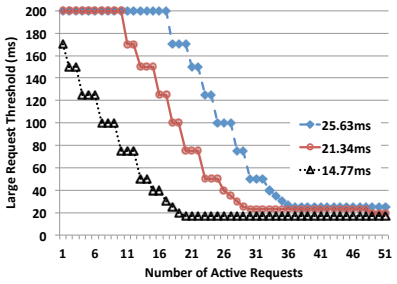


Figure 4: Example large request threshold tables output by threshold-calculation with input of the same work distribution and rps, but three different target latencies. The x-axis is the number of active requests, and the y-axis is the large request threshold. Each curve plots the output threshold table for a given latency.

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.

[3] The span of a request is the longest dependency chain in the request that must execute sequentially.

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. Tail-control behaves like steal-first by default. 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 tracks 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

```
1  void scheduling_loop(Worker *w) {
2    Task *task = NULL, *last_task = NULL;
3    while (more_work_to_do()) {
4      if (task != NULL) {
5        do {
6          execute(task);
7          last_task = task;
8          task = pop_deque(w);
9        } while(task != NULL);
10       if (last_task->parent() != NULL) {
11         task = last_task->parent();
12         if (task->ref_count() == 0) continue;
13       } else if (last_task->is_root() &&
14                  last_task->is_done()) {
15           global_queue->dec_active_request_count();
16         }
17       }
         // about to switch request; update bookkeeping info
18       long long proc_time = get_time() - w->start_time;
19       w->last_request = w->curr_request;
20       w->curr_request = NULL;
21       w->last_request->accum_process_time(proc_time);
22     } //end of if (task != NULL)
23     check_pileup_phase();
24     task = try_random_steal();
25     if (task == NULL)
26       task = global_queue->admit_request();
27     if (task != NULL) {
28       w->start_time = get_time();
29       w->curr_request = task->get_request();
30     }
31   } // end of while (more_work_to_do())
32 } // end of scheudling_loop
```

(a) The main loop.

```
1  Task * try_random_steal() {
2    Task *task = NULL;
3    while (has_stealable_victims()) {
4      Worker *vic = choose_random_victim();
5      task = try_steal_deque_top(vic);
6    }
7    return task;
8  }

9  void check_pileup_phase() {
10   int active = global_queue->get_active_request_count();
11   int req_thresh = large_request_table[active];
12   long long added_time, new_proc_time,
13   long long curr_time = get_time();
14   hashtable req_map;

     // update the processing time for each executing request
15   for (Worker *w : worker_list) {
16     Request *req = w->curr_request;
17     if (req == NULL) continue;
18     added_time = curr_time - w->start_time;
       // find returns 0 if req is not found
19     if (req_map.find(req) == 0)
20       new_proc_time = req->get_proc_time() + added_time;
21     else
22       new_proc_time = req_map.find(req) + added_time;
23     req_map.insert(req, new_proc_time);
       // mark a request that exceeds threshold
24     if (new_proc_time > req_thresh) {
25       if (req.is_valid()) req->set_stealable(false);
26     }
27   }
28 }
```

(b) Helper routines.

Figure 5: The pseudo code for tail-control in a work-stealing runtime system. Tail-control adds only the bold lines and the function `check_pileup_phase` to steal-first.

scheduling loop to perform book keeping 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 line 24 and line 26, one trivially obtains admit-first.

Figure 5b shows the pseudo code for subroutines invoked by the main scheduling loop. Again, the differences between tail-control and steal-first are few and are highlighted in bold. Specifically, in tail-control, a thief gives up on the steal if the task on top of the victim's deque is a not-stealable large request. The `try_random_steal` (line 5) and the `has_stealable_victims` (line 3) implement this difference. This implementation causes the marked large request to serialize lazily; the parallelism dissolves when all workers working on that request exhaust their local tasks. We choose not to enforce the marked request to serialize immediately to avoid additional overhead.

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 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.

Note the pseudo code simply shows the high-level control flow, abstracting many operations, including synchronization. Shared fields in a request object and the global queue are protected by locks. The locks are acquired only when concurrent writes are possible. One notable feature is that when a thief executes `check_pileup_phase`, it does not acquire locks when reading shared fields, such as `w->start_time`, `w->curr_request`, and `req->get_proc_time`. We intentionally avoid this locking overhead at the cost of some loss in accuracy in calculating the requests' processing time due to potentially stale values of these fields the thief may read. In this way, the processing time calculation happens entirely in a distributed manner and a thief will never interfere workers who are busy executing requests.

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 memory and Linux version 3.13.0. When running experiments, we disable processor throttling, processor sleeping, and hyper-threading. The Tail-control scheduler 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 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 $1/rps$, and a log-normal distributions with a mean equal to $1/rps$ and varying standard deviations. In other words, for all the distributions, the requests are generated in an open loop at the rate of rps (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. 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 algorithms to position the performance of steal-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 admit-first and steal-first, the two 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 Bing,finance, and log-normal work distributions respectively, where each bar graph in a figure shows the result for one load 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,[4] 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'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 steal-first and of 24% to 65% over admit-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 provides 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 bookkeeping, the overhead is lightweight and not significant enough to affect performance. 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 a 36% improvement. These results indicate that tail-control performs well under different types of workload characteristics.

Log-normal Workload.

In Figure 8, we generate the work of a request using a log-normal distribution with mean of 10ms and standard deviation of 13ms. We selectively choose the mean of 10ms in order to directly compare the results with that of Bing workload, as the mean work of Bing workload is also 10ms. The standard deviation is chosen, such that the log-normal work distribution has a slightly shorter tail 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 66%.

5.2 Different Arrival Distributions

Note that in all the previous experiments, requests arrive according to a Poisson arrival process. In these experiments, we vary the inter-arrival time distribution and select the log-normal distribution, which has larger variance than the Poisson distribution. Figure 9 shows experimental results for a log-normal workload and a log-normal arrival distribution with mean inter-arrival time of 0.83ms (rps of 1200) and standard deviation of 1.09ms. Note that this experiment is constructed such that it has almost the same parame-

[4] The 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.

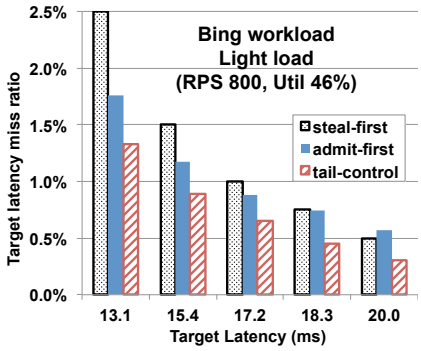


Figure 6(a)

Target (ms)	13.1	15.4	17.2	18.3	20.0
Imp. over SF	47%	41%	35%	40%	39%
Imp. over AF	24%	25%	26%	39%	47%

Table 6(a)

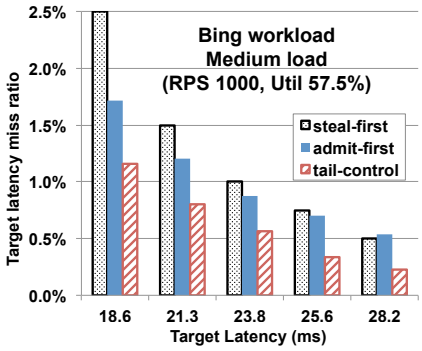


Figure 6(b)

Target (ms)	18.6	21.3	23.8	25.6	28.2
Imp. over SF	54%	46%	43%	55%	54%
Imp. over AF	33%	33%	35%	52%	57%

Table 6(b)

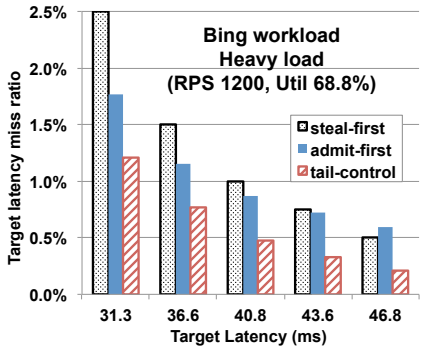


Figure 6(c)

Target (ms)	31.3	36.6	40.8	43.6	46.8
Imp. over SF	52%	49%	52%	56%	58%
Imp. over AF	31%	34%	45%	54%	65%

Table 6(c)

Figure 6: Results for the 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.

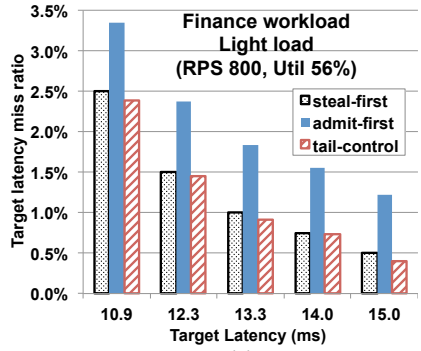


Figure 7(a)

Target (ms)	10.9	12.3	13.3	14.0	15.0
Imp. over SF	5%	3%	9%	1%	19%
Imp. over AF	29%	39%	51%	53%	67%

Table 7(a)

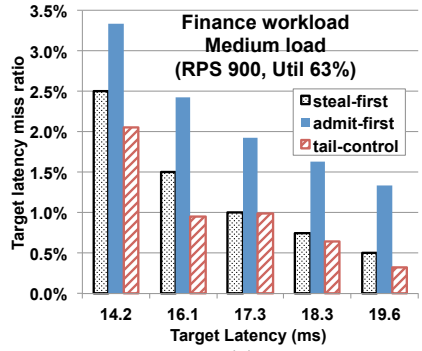


Figure 7(b)

Target (ms)	14.2	16.1	17.3	18.3	19.6
Imp. over SF	18%	36%	2%	14%	36%
Imp. over AF	39%	61%	49%	60%	76%

Table 7(b)

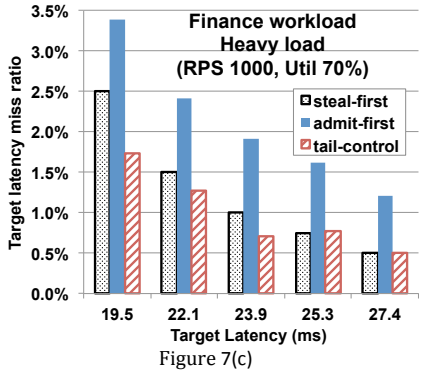


Figure 7(c)

Target (ms)	19.5	22.1	23.9	25.3	27.4
Imp. over SF	31%	15%	29%	-3%	-1%
Imp. over AF	49%	47%	63%	52%	58%

Table 7(c)

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

ters as that of Figure 8(c), except that the latter's inter-arrival time has a smaller standard deviation of 0.91ms. By comparing the two experiments, we can see that the relative trend remains the same. In particular, steal-first is better than admit-first, while tail-control outperforms the best of them by 25% to 44%. The comparison between Poisson and log-normal arrival distribution with Bing workload are similar too. The results indicate that request arrival distribution does not impact the relative performance much.

5.3 Request with Sub-Linear Speedup

For the rest of the paper, we focus on the Bing and log-normal workloads at the heavy load of 1200 rps. 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 Figure 10 we intentionally add sequential segments with a total length of 10% work into each request, resulting a parallelism degree of less than 10 and smaller than the total 16 cores. As discussed in Section 4.1, we incorporate the span into the tail-control threshold calculation. Note that this experiment has almost the same parameters as Figure 6(c), except for a smaller parallelism degree. By comparing the two, 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 are similar to that of Bing workload. Tail-control improves less in this setting, because small requests are not able to utilize all the available cores, even when large requests are serialized. Moreover, the large request does not monopolize the entire system due to its long 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 and the target latency. As the degree of request parallelism 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

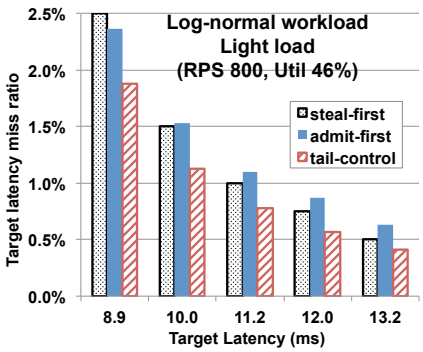


Figure 8(a)

Target (ms)	8.9	10.0	11.2	12.0	13.2
Imp. over SF	25%	25%	23%	24%	17%
Imp. over AF	20%	26%	29%	35%	34%

Table 8(a)

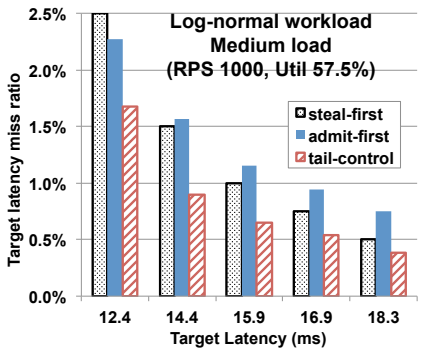


Figure 8(b)

Target (ms)	12.4	14.4	15.9	16.9	18.3
Imp. over SF	33%	40%	35%	28%	22%
Imp. over AF	26%	43%	44%	43%	49%

Table 8(b)

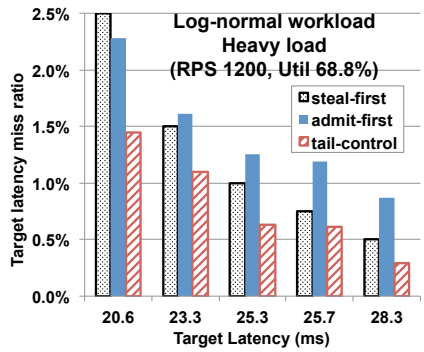
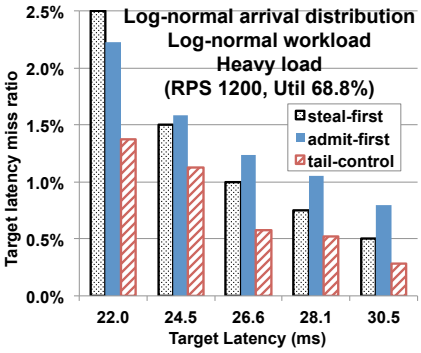


Figure 8(c)

Target (ms)	20.6	23.3	25.3	25.7	28.3
Imp. over SF	42%	27%	37%	18%	41%
Imp. over AF	37%	32%	50%	49%	66%

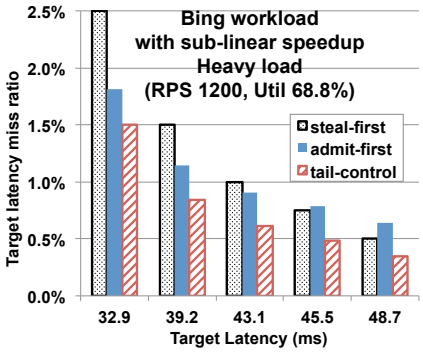
Table 8(c)

Figure 8: The log-normal workload results with the same figure and table configuration as in Figure 6.



Target (ms)	22.0	24.5	26.6	28.1	30.5
Imp. over SF	45%	25%	42%	31%	44%
Imp. over AF	38%	29%	53%	51%	65%

Figure 9: Results for the log-normal workload and a log-normal arrival distribution with 1200 rps; Figure and table configurations are similar as in Figure 8.



Target (ms)	32.9	39.2	43.1	45.5	48.7
Imp. over SF	40%	44%	39%	35%	31%
Imp. over AF	17%	27%	33%	38%	46%

Figure 10: Results for the Bing workload and a Poisson arrival distribution with 1200 rps for requests with sub-linear speedup; Figure and table configurations are similar as in Figure 6.

tail-control performs when the work distribution differs from the input work distribution for the threshold-calculation algorithm. We experimentally evaluate how brittle tail-control is when provided with a somewhat inaccurate input work distribution. The experiment in Figure 11 has the same setting as Figure 8(c) with target latency of 28ms. In addition to the tail-control with the correct input, we also run tail-control using inaccurate input. In particular, we slightly alter the input work distribution by changing the standard deviation while keeping the same mean work. As 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 10ms and 17ms instead of the true 13ms, 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. When the standard deviation of the profiling workload is small, tail-control less aggressively serializes requests and it performs similarly 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. In this case, it unnecessarily serializes only a few large requests. Since the probability of large requests is 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 case causes tail-control to 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 or an online profiling mechanism should profile the new work distribution and rps. Although we presented the threshold calculation (Section 4.1) as an offline algorithm, it is fast; thus, an interactive service could sample its workload and recalculate the large request thresholds on-line and then adapt the tail-control scheduler 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

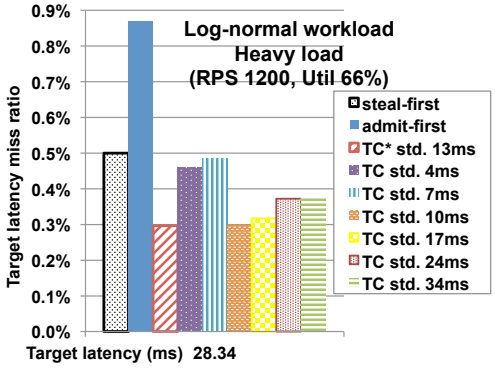


Figure 11: Results for the log-normal workload a and Poisson arrival distribution with 1200 rps and a target latency of 28ms. We compare tail-control when using inaccurate input distributions with smaller to larger standard deviation from left to right.

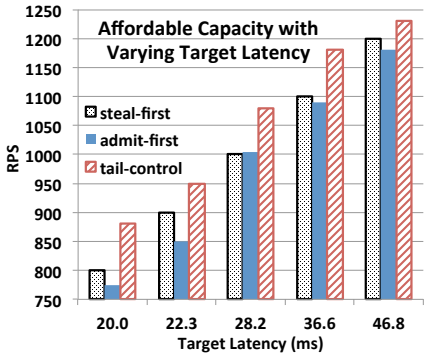


Figure 12: The Bing workload results. Tail-control increases system capacity for different target latencies compared to steal-first and admit-first.

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 20.00ms, tail-control sustains 880 rps 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 Algorithms

Now we provide a comparison with two additional 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 to 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. Default-TBB acts similarly to admit-first, but it admits requests in a random order. A request that has waited for a very long time could potentially be admitted later than a request that just arrived, causing significantly increases in the latency of some requests when the system is busy and consequently increasing the request target miss ratio. On the other hand, without any global queue default-TBB has smaller overheads. In this case, the system processes requests slightly faster and hence reduces the length of pileup phase. Thus, default-TBB performs slightly better than admit-first in these cases. However, tail-control still outperforms default-TBB by at least 31% in Figure 13 and 32% in Figure 14, respectively.

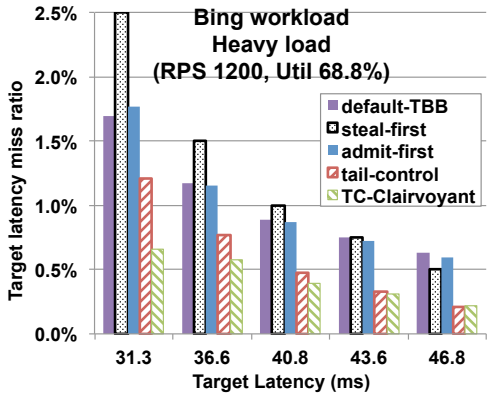


Figure 13: The Bing workload results. The figure is the same as Figure 6(c), except it adds default-TBB and TC-Clairvoyant.

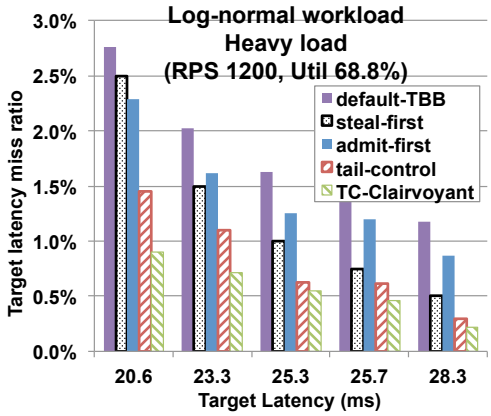


Figure 14: The log-normal workload results. The figure is the same as Figure 8(c), except it adds default-TBB and TC-Clairvoyant.

The second baseline algorithm marked as **TC-Clairvoyant** in Figure 14 and Figure 13 shows the lower bound of tail-control, because this scheduler utilizes the exact work of each request to perform tail-control strategy. Specifically, we modified TBB and the 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 (thus clairvoyant). 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 serves as a 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 known ahead-of-time to the scheduler. Of course, this this knowledge is not typically available.

From Figure 14 and Figure 13, we can see that TC-Clairvoyant and tail-control have comparable performance when the target latency is long, because to minimize for long target latency, only the

very large requests need to be serialized. When the work distribution has a relatively long tail, we can more easily distinguish large requests from other medium or small requests. TC-Clairvoyant improves much more when the target latency is short because it on occasion will serialize some medium requests as soon as they begin executing. In the case of tail-control, medium requests will execute for most of their work before tail-control can distinguish them from small requests. The TC-Clairvoyant results shows that our tail-control strategy would improve performance even more if there was an accurate and efficient mechanism to predict 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. The x-axis is time t as the trace executes. The lower y-axis is the queue length at time t. Recall that tail-control works in the same way as steal-first, except that under highly loaded conditions, it serializes large requests so as to reduce the waiting time of small requests in the queue. This figure shows that tail-control succeeds in this stated goal — when the instantaneously system load is high (for instance, at time 0.2s and 0.6s), the queue length is shorter under tail-control than under steal-first, but otherwise they follow each other closely.

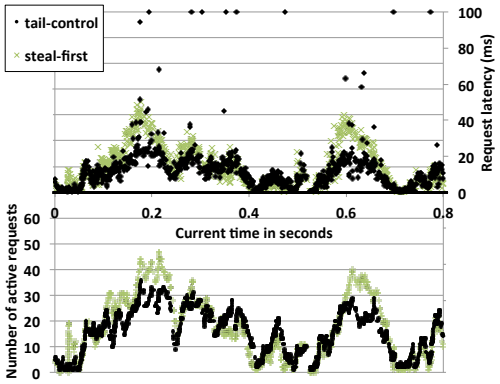


Figure 15: Number of active requests (lower part) and request latency (upper part) traces of the same workload under steal-first and tail-control in a 0.8 second window (Bing workload).

The top part of the figure is even more interesting. For the same experiment, the upper y-axis shows the latency of each request released at time t along the x-axis. We can see clearly that tail-control sacrifices a few requests (e.g., the ones with over 60ms latency), increasing their latency by a large amount in order to reduce the latency of the remaining requests. For the high instantaneous load at time $0.2s$ for instance, tail-control has a few requests that have very large latencies, but the remaining requests have smaller latencies than steal first. In particular, under steal-first, 183 requests have latencies that exceed the target latency of 25ms, while only 42 requests exceed the target under tail-control.

6. Related Work

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; 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 with multiple jobs.

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 Mean Response Time. Raman et al. propose an API and runtime system for dynamic parallelism (Raman et al. 2011), in which developers express parallelism options and goals, such as reducing mean response time. The runtime dynamically chooses the degree of parallelism to meet the goals at admission time, but does not change it during request execution. Jeon et al. (Jeon et al. 2013) propose a dynamic parallelization algorithm to reduce the mean response time of web search queries. It decides the degree of parallelism for each request at admission time, based on the system load and the average speedup of the requests. Both use mean response time as their optimization goal, and neither differentiates large requests from small requests.

Interactive Server Parallelism for Tail Latency. Jeon et al. (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. They exploit the large variance of request service demand to reduce 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, and (2) their request parallelism decision is made regardless of the system load and workload characteristics.

Haque et al. (Haque et al. 2015) present a few-to-many parallelism technique which dynamically increases request parallelism degree during execution. Their system completes small requests sequentially to save resources, parallelizes large requests to reduce tail latency, and uses system load and workload characteristics 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 objectives 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 requests 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 requests that meet a target latency for multiple concurrent requests. Our experiments indicate that our prototype work-stealing scheduler using TBB is very effective for highly parallel interactive workloads. Although no current interactive service uses work stealing, as far as we are aware, the work stealing framework is appealing because it supports the most general models of both static and 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-1150036, and by ONR grants N000141310800. The authors thank the shepherd Charlie Curtsinger and other anonymous reviewers for their suggestions on improving this paper.

References

K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson. Adaptive scheduling with parallelism feedback. In *PPoPP*, pages 100–109, 2006.

K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems (TOCS)*, 26(3):7, 2008.

Apache Lucene. `http://lucene.apache.org/`, 2014.

N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34 (2):115–144, 2001.

R. Barik, Z. Budimlić, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar. The Habanero multicore software research project. In *ACM Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 735–736, Orlando, Florida, USA, 2009.

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

R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP*, pages 207–216, 1995.

S. C. Borst, O. J. Boxma, R. Núñez-Queija, and A. Zwart. The impact of the service discipline on delay asymptotics. *Performance Evaluation*, 54 (2):175–206, 2003.

M. Broadie and P. Glasserman. Estimating security price derivatives using simulation. *Manage. Sci.*, 42:269–285, 1996.

V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In *International Conference on Principles and Practice of Programming in Java (PPPJ)*, pages 51–61, 2011.

P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 519–538, 2005.

G. Cortazar, M. Gravet, and J. Urzua. The valuation of multidimensional american real options using the lsm simulation method. *Comp. and Operations Research.*, 35(1):113–129, 2008.

M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *ACM International Conference on Supercomputing (ICS)*, pages 157–166, 2006.

J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147–171, Dec. 2006.

J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

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

M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

M. E. Haque, Y. hun Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 161–175, 2015.

Y. He, W.-J. Hsu, and C. E. Leiserson. Provably efficient online nonclairvoyant adaptive scheduling. *Parallel and Distributed Systems, IEEE Transactions on (TPDS)*, 19(9):1263–1279, 2008.

Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling interactive services with partial execution. In *ACM Symposium on Cloud Computing (SOCC)*, page 12, 2012.

Intel. Intel CilkPlus v1.2, Sep 2013. `https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm`.

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

M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Adaptive parallelism for web search. In *ACM European Conference on Computer Systems (EuroSys)*, pages 155–168, 2013.

M. Jeon, S. Kim, S.-W. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: taming tail latencies in web search. In *ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 253–262, 2014.

C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for SMT multiprocessor architectures. In *PPoPP*, pages 236–246, 2005.

S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *ACM SIGARCH International Conference on Computer Architecture (ISCA)*, pages 158–169, 2015.

S. Kim, Y. He, S.-w. Hwang, S. Elnikety, and S. Choi. Delayed-dynamic-selective (DDS) prediction for reducing extreme tail latency in web search. In *WSDM*, pages 7–16, 2015a.

S. Kim, Y. He, S.-W. Hwang, S. Elnikety, and S. Choi. Delayed-Dynamic-Selective (DDS) prediction for reducing extreme tail latency in web search. In *ACM International Conference on Web Search and Data Mining (WSDM)*, 2015b.

L. Kleinrock. Time-shared systems: A theoretical treatment. *Journal of the ACM (JACM)*, 14(2):242–261, 1967.

W. Ko, M. N. Yankelevsky, D. S. Nikolopoulos, and C. D. Polychronopoulos. Effective cross-platform, multilevel parallelism via dynamic adaptive execution. In *IPDPS*, pages 8 pp–, 2002.

V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu. Work-stealing without the baggage. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 297–314, 2012.

D. Lea. A Java fork/join framework. In *ACM 2000 Conference on Java Grande*, pages 36–43, 2000.

I.-T. A. Lee, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang. On-the-fly pipeline parallelism. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 140–151, 2013.

J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *International Symposium on Computer Architecture (ISCA)*, pages 270–279, 2010.

D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *ACM SIGPLAN Notices*, volume 44, pages 227–242, 2009.

C. E. Leiserson. The Cilk++ concurrency platform. *J. Supercomputing*, 51 (3):244–257, 2010.

J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 50–61, 2001.

J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 248–259, 2011.

C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, 1993.

OpenMP. OpenMP Application Program Interface v4.0, July 2013. `http://http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 116–125, 2011.

A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using DoPE: The degree of parallelism executive. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, volume 46, pages 26–37, 2011.

J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2010.

S. Ren, Y. He, S. Elnikety, and K. S. McKinley. Exploiting processor heterogeneity in interactive services. In *ICAC*, pages 45–58, 2013.

B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Trans. Internet Technol.*, 6(1):20–52, 2006.

Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 75–84, 2009.

A. Wierman and B. Zwart. Is tail-optimal scheduling possible? *Operations research*, 60(5):1249–1257, 2012.

T. Y. Yeh, P. Faloutsos, and G. Reinman. Enabling real-time physics simulation in future interactive entertainment. In *ACM SIGGRAPH Symposium on Videogames*, Sandbox '06, pages 71–81, 2006.

J. Yi, F. Maghoul, and J. Pedersen. Deciphering mobile search patterns: A study of Yahoo! mobile search queries. In *ACM International Conference on World Wide Web (WWW)*, pages 257–266, 2008.

Zircon Computing. Parallelizing a computationally intensive financial R application with zircon technology. In *IEEE CloudCom*, 2010.