Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services *

Md E. Haque[†] Yong hun Eom[‡] Yuxiong He[*]
Sameh Elnikety[*] Ricardo Bianchini[†*] Kathryn S. McKinley[*]

[†]Rutgers University [‡]University of California, Irvine [*]Microsoft Research
mdhaque@cs.rutgers.edu yeom@uci.edu {yuxhe,samehe,ricardob,mckinley}@microsoft.com

Abstract

Interactive services, such as Web search, recommendations, games, and finance, must respond quickly to satisfy customers. Achieving this goal requires optimizing *tail* (e.g., 99th+ percentile) latency. Although every server is multicore, parallelizing individual requests to reduce tail latency is challenging because (1) service demand is unknown when requests arrive; (2) blindly parallelizing all requests quickly oversubscribes hardware resources; and (3) parallelizing the numerous short requests will not improve tail latency.

This paper introduces Few-to-Many (FM) *incremental parallelization*, which dynamically increases parallelism to reduce tail latency. FM uses request service demand profiles and hardware parallelism in an offline phase to compute a policy, represented as an interval table, which specifies when and how much software parallelism to add. At runtime, FM adds parallelism as specified by the interval table indexed by dynamic system load and request execution time progress. The longer a request executes, the more parallelism FM adds. We evaluate FM in Lucene, an open-source enterprise search engine, and in Bing, a commercial Web search engine. FM improves the 99th percentile response time up to 32% in Lucene and up to 26% in Bing, compared to prior state-of-the-art parallelization. Compared to running requests sequentially in Bing, FM improves tail latency by a factor of two. These results illustrate that incremental parallelism is a powerful tool for reducing tail latency.

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

Keywords Dynamic Parallelism; Interactive Services; Multithreading; Tail Latency; Thread Scheduling; Web Search

* Haque and Eom contributed equally to this work as interns at Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '15, March 14–18, 2015, Istanbul, Turkey..
Copyright © 2015 ACM 978-1-4503-2835-7/15/03. . . $15.00.
http://dx.doi.org/10.1145/2694344.2694384

1. Introduction

Interactive online services, such as Web search, financial trading, games, and online social networks require consistently low response times to attract and retain users [14, 33]. Interactive service providers therefore define strict targets for *tail latencies* — 99th percentile or higher response times [9, 10, 17, 39] to deliver consistently fast responses to user requests. The lower the tail latency, the more competitive the service. Moreover, reducing each server's tail latency is critical when a request spans several servers and responses are aggregated from these servers. In this case, the slower servers typically dominate the response time [22].

Reducing tail latency is challenging, in part because requests exhibit highly variable demand. For example in finance servers and Web search, most user search requests are short, but a significant percentage are long [9, 17, 32]. Prior work on search engines shows that in a distributed system, the longest requests (99th-percentile execution times) are up to a factor of 10× larger than the average execution times, and even 100× larger than the median [9, 19]. While other sources of variability, such as interference from other workloads, hardware variability, and network congestion, contribute to tail latency, this prior work establishes long requests in computationally intensive workloads are a primary factor in tail latency. A critical component of reducing tail latency is reducing the execution time of long requests.

This paper shows how to reduce tail latency with the judicious use of parallelism at the level of an individual request. We exploit the opportunity afforded by multicore hardware and parallelism in these applications. However, simply parallelizing all requests oversubscribes multicore resources, degrading average and tail latency on search workloads. We therefore seek to parallelize only the long requests, which contribute the most to tail latency. However, when a request arrives, we do not know its service demand (and it is difficult to predict if the request is short or long [26]). Thus, we determine demand as the request executes. We introduce *incremental* parallelism to reduce tail latency, which dynam-

ically adds parallelism to an individual request based on the requests' progress and the system load.

Few-to-Many (FM) incremental parallelization targets interactive services. It progressively increases parallelism by increasing the number of worker threads for a request, from 1 to a given maximum, over the duration of its execution. Short requests execute sequentially, saving resources, and long requests execute in parallel, reducing tail latency. The key challenge for dynamic parallelization is determining *when* to increase the parallelism and by *how much*, as a function of hardware resources, parallelism efficiency, dynamic load, and individual request progress.

FM has an offline and an online phase. The offline phase takes as input a demand profile of requests with (1) their individual sequential and parallel execution times, and (2) hardware parallelism (core resources). We perform a scalability analysis to determine a maximum degree of software parallelism to introduce. Although the individual requests submitted to a service change frequently, the demand profile of these requests changes slowly [26, 32], making periodic offline or online processing practical. The offline phase computes a set of schedules that specify a time and degree of parallelism to introduce based on dynamic load and request progress. The algorithm that produces these schedules seeks to fully utilize available hardware parallelism at all loads. At low load, FM aggressively parallelizes requests. At moderate to high load, FM runs short requests sequentially and incrementally adds parallelism to long requests to reduce their tail latency. Online, each request self-schedules, adding parallelism to itself based on its current progress and the instantaneous system load. Decentralized self-scheduling limits synchronization costs and therefore improves scalability.

We implement FM in Lucene, an open-source Enterprise search engine, and Microsoft Bing, a commercial Web search engine. On production request traces and service demand profiles, FM reduces tail latency significantly. We compare FM to fixed parallelism polices on a single server and explore sensitivity to load and workload. FM improves the 99th percentile latency by 32% in Lucene and 26% in Bing, compared to state-of-the-art search parallelization techniques [18, 19] because it dynamically adapts to instantaneous load. Compared to sequential processing, incremental parallelization reduces tail latency on a single server by a factor of two. Bing engineers have already deployed incremental parallelization on thousands of production servers.

Our results have implications for the total cost of ownership (TCO) of large services. Given a target tail latency, FM allows higher utilization of servers while meeting the latency target. This higher utilization allows service providers to reduce their infrastructure costs. For example, our Bing results show that the provider can leverage FM to service the same user load with 42% fewer servers. This result is significant because the infrastructure TCO of commercial services is typically dominated by the cost of purchasing servers [9, 15].

This paper makes the following contributions.

- We introduce Few-to-Many (FM) incremental parallelization for interactive services.
- We develop an FM scheduler that determines when and how much parallelism to introduce based on maximum software parallelism, hardware parallelism, dynamic load, and individual request progress to optimize tail latency.
- We evaluate our approach in open-source and commercial search engines, using production workloads and service demand profiles.
- We show substantial improvements in tail latency over prior parallelization approaches that improve users' experiences and reduce service providers' infrastructure cost.

Although we evaluate FM on search, the results apply to other interactive services, such as online ads, financial recommendations, and games, that are computationally intensive, have stable workload distributions, and are easy to parallelize incrementally [13, 21].

2. Background

This section overviews the characteristics and parallelism opportunities of interactive services. It also discusses the production service demand distributions and scalability characteristics of our workloads. Section 3 shows how we exploit these characteristics to reduce tail latency.

Characteristics Many interactive services, such as search, financial trading, games, and social networking are *computationally intensive* [1, 13, 18, 28, 32]. To meet stringent latency constraints, they are carefully engineered such that (1) their working set fits in memory, since any disk access may compromise responsiveness, and (2) although they may frequently access memory, they are not memory-bandwidth constrained.

As an example, consider Web search, which divides the work among many worker servers that compute over a subset of the data, and then a few servers aggregate the responses [3, 6]. In Bing Web search, each worker server has 10s of GBs of DRAM used for caching a partition of the inverted index that maps search keywords to Web documents. This cache is designed to limit disk I/O [18]: the average amount of disk I/O is less than 0.3 KB/s. To attain responsiveness and avoid queuing delay, Bing provisions additional servers to ensure that workers operate at low to modest loads [18]. The average queuing delay at a worker is 0.35 ms even with high 70% CPU utilization. Network I/O is also a small fraction of the overall request latency at 2.13 ms on average. CPU computation is the largest fraction of response time at well over 70% and is even higher for long queries. Consequently, reducing tail latency requires reducing compute time.

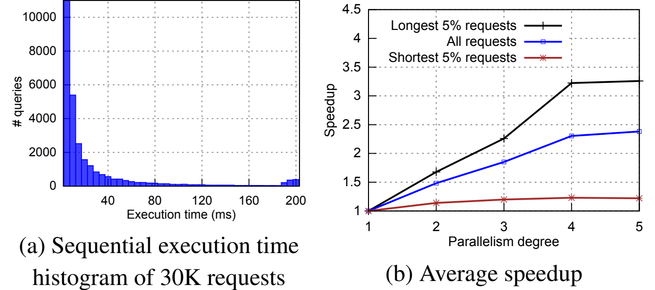
Opportunity for parallelism Interactive services today commonly exploit large-scale parallelism in two ways. (1) They distribute the processing over hundreds or thousands of servers at data center scale because they must process requests over vast amounts of data that do not fit on a single server. (2) At each server, they process multiple requests concurrently. In this paper, we exploit parallelism in a third complementary way. We explore intra-request parallelism on a multicore server. In particular, we execute individual requests using concurrent threads on multiple cores to reduce their execution time. Prior work demonstrates that individual requests in interactive systems, such as search, finance trading, and games are easily parallelized [13, 18]. In addition, these workloads are often amenable to dynamic parallelism, in which the scheduler can vary the number of worker threads per request during the request execution. Dynamic parallelism is supported by many parallel libraries and runtimes, such as Cilk Plus [5], TBB [7], TPL [25], and Java [12]. We introduce *incremental parallelism*, a new form of dynamic parallelism that incrementally increases the parallelism degree for individual requests. Section 6.1 and 7.1 demonstrate how to implement incremental parallelism on enterprise search and Web search services.

Demand distributions and scalability To motivate our approach, we study the demand distributions and scalability of Lucene and Bing, our two evaluation systems. However, our approach is more generally applicable than these workloads, because other interactive services have similar demand and parallelism characteristics [13, 21]. We gather production user requests for both Bing and Lucene and measure the sequential and parallel execution times for each request executing alone on a single server. (Sections 6 and 7 describe the methodologies and systems in more detail.)

Figure 1(a) shows the service demand distribution of 30K requests for the Bing Index Server Nodes (ISN). The x-axis is the execution time in 5 ms bins and y-axis is the frequency of requests in each bin. Most requests are short, with more than 85% taking below 15 ms. A few requests are very long, up to 200 ms. The gap between the median and the 99th percentile is a factor of 27×. The slight rise in frequency at 200 ms is because the server terminates any request at 200 ms and returns its partial results. We observe that these workload characteristics are fairly consistent across hundreds of ISN servers with different partitions of the index.

Figure 1(b) presents Bing parallelism efficiency, i.e., the speedup of requests with different parallelization degrees for all requests, the longest 5%, and the shortest 5%. Long requests have over 2 times speedup with 3 threads. In contrast, short requests have limited speedup, a factor of 1.2 with 3 threads. These results show that at degrees higher than 4, additional parallelism does not lead to speed up.

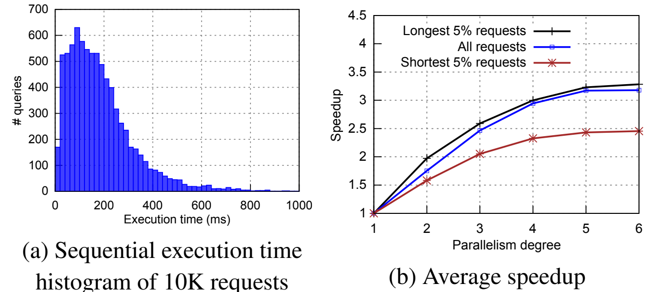
Similarly, Figure 2(a) shows the service demand histogram of 10K Wikipedia search requests for Lucene in 20 ms bins and Figure 2(b) shows the parallelism efficiency.



(a) Sequential execution time histogram of 30K requests

(b) Average speedup

Figure 1: Bing demand distribution and average speedup.



(a) Sequential execution time histogram of 10K requests

(b) Average speedup

Figure 2: Lucene demand distribution and average speedup.

Again, we observe many short requests and few long requests. The maximum number of requests are in the bin around 90 ms and the median service demand is 186 ms. Since the ratio of short to long requests is high, parallelizing the long requests has the potential to reduce tail latency. Figure 2(b) shows that on average, requests exhibit almost linear speedup for parallelism degree 2. Parallelism is slightly less effective for 2 to 4 degrees and is not effective for 5 or more degrees.

Both workloads show diminishing effectiveness of parallelism and motivates limiting the maximum degree of parallelism. Both also show that parallelism is most effective on long requests, which suggests devoting parallel hardware resources to long requests instead of short ones. Moreover, long requests impact tail latency the most.

3. Rationale Behind FM Parallelism

This section discusses the intuition and theory behind FM incremental parallelism and requirements for implementing it effectively.

3.1 Intuition Behind FM Incremental Parallelism

A simple approach to using intra-request parallelism is to use a fixed number of worker threads for each request. Depending on the number of threads per request, *fixed* parallelism would either oversubscribe resources at high systems loads or underutilize them under light loads. To see an example of oversubscription and its impact on response times, consider Figure 3. The figure shows the mean and 99th percentile response times of Lucene as a function of load when all requests run with 1 worker thread (SEQ) and 4 worker threads (FIX-4). (See Section 6 for our methodology.) Clearly, using

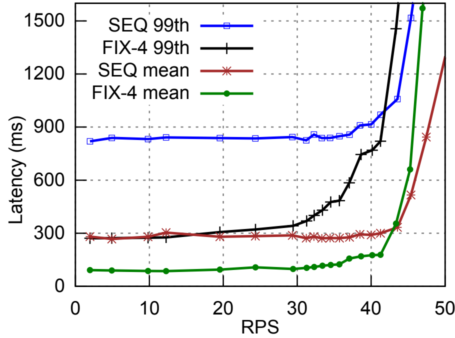


Figure 3: Effect of fixed parallelism on latency in Lucene.

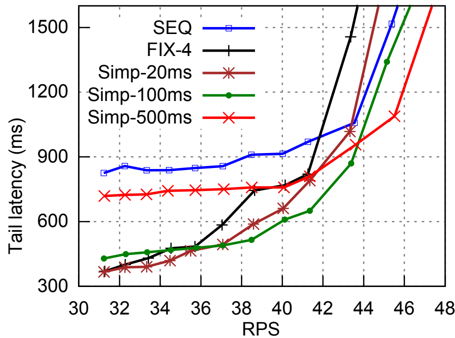


Figure 4: 99th percentile tail latency of sequential, degree 4, and simple fixed addition of dynamic parallelism in Lucene.

4 threads for all requests reduces tail latency well with low load, but gets progressively worse with higher load. In fact, using 4 threads becomes worse than 1 thread around 42 requests per second (RPS). Henceforth, we focus on the range 30 to 48 RPS, since the latency is typically flat below 30 RPS and too high beyond 48 RPS.

Another problem with fixed parallelism is that it targets all requests equally. However, long requests have a greater impact on tail latency than short ones. We prefer to parallelize long requests, but it is difficult to predict if a request will be long or short [26]. Fortunately, requests in many interactive services are amenable to *incremental* parallelization. By dynamically increasing the degree of parallelism as a request executes, long requests will exploit more parallelism than short requests. This insight is the key rationale behind FM incremental parallelization.

3.2 Theoretical Foundation of FM Parallelization

Intuitively, FM parallelization increases the probability that short requests will finish with less parallelism, which saves resources, while it assigns long requests more parallelism, which reduces tail latency. This section presents the theoretical foundation behind this intuition. Theorem 1 shows that, given a tail latency constraint, the optimal policy that minimizes average resource usage assigns parallelism to requests in non-decreasing order. In other words, to minimize resource usage under a latency constraint, if a request ever changes its parallelism, it will only increase, transitioning from *few to many* degrees of parallelism. The theorem makes two assumptions. (1) We do not know if a request is long or short a priori, but we know the service demand distribution, i.e., the distribution of *sequential* request execution times (see Figure 2(a) for an example). (2) Each request exhibits sublinear speedup, i.e., parallelism efficiency decreases with increase in parallelism degree, as we showed in the previous section and is true for many workloads.

THEOREM 1. *Given a request service demand distribution and a sublinear parallelism speedup function, to meet a tail latency constraint, an optimal policy that minimizes average resource usage assigns parallelism to requests in non-decreasing order.*

Proof. Please see Appendix. ■

Intuitively, Theorem 1 means that given an optimal schedule that first adds and then removes parallelism, there exists an equivalent schedule which only adds parallelism. We exploit this theorem to limit our offline search to finding an optimal few-to-many schedule. The dual problem of Theorem 1 also holds: given a fixed amount of resources, few-to-many minimizes latency. For a server system where each request gets a limited amount of resources, FM minimizes tail latency.

3.3 Practical and Effective FM Parallelization

The simplest approach to incremental parallelism is to simply add parallelism periodically, e.g., add one thread to each request after a fixed time interval. Unfortunately, this approach does a poor job of controlling the total parallelism (resource usage and contention), regardless of the interval length. Figure 4 illustrates this problem by comparing the 99th percentile tail latency of Lucene when simply adding parallelism at fixed 20, 100, and 500 ms intervals with executing each request with 1 thread and 4 threads, as a function of load. The figure shows that increasing parallelism dynamically does reduce tail latency more than fixed parallelism at medium and high loads. Short requests use fewer than 4 threads, and thus limit oversubscription of resources. However, no fixed interval is ideal across the entire load spectrum. The shorter the interval, the higher the tail latency at high load. Conversely, the longer the interval, the higher the tail latency at low load. These results suggest that FM parallelization can be effective, but to select intervals correctly FM must carefully consider the system load. Fundamentally, the main requirements for effective incremental parallelization are the following.

FM scheduling must efficiently utilize resources. When the load is low (no resource contention), FM should be aggressive and choose shorter intervals to better utilize the hardware resources. At high load (high contention), it must be conservative and choose longer intervals to apply parallelism more selectively to just the longest requests. We observe that maintaining a fixed overall number of software threads is a good way to control hardware resource utilization as the load varies.

FM scheduling must consider workload characteristics. FM must consider the distribution of the service demand. For example, if the vast majority of requests take less than 100 ms, an interval of 100 ms will not exploit much parallelism. Moreover, FM must consider any overhead due to parallelism. With lower overhead, we choose smaller intervals, parallelizing requests more aggressively. With higher overhead, we choose larger intervals, parallelizing requests more conservatively to avoid wasting resources.

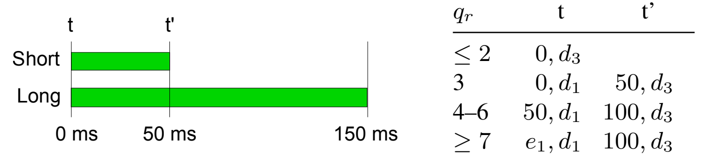
FM scheduling must consider scalability of the workload. When speedups tail off at high degrees, adding more parallelism is a less effective use of hardware resources. FM thus limits parallelism to an effective maximum.

4. Few-to-Many Incremental Parallelization

This section describes Few-to-Many (FM) incremental parallelization for a single server. Our goal is to reduce tail latency. The FM scheduler achieves this goal by exploiting all hardware parallelism and judiciously adding software parallelism. FM has two phases. (1) An offline analysis phase produces an *interval table*. (2) An online dynamic phase schedules requests by indexing the interval table. FM computes the interval table offline using as inputs the maximum software parallelism per request, hardware parallelism, and service demand profiles of sequential and parallel execution times. The interval table specifies when during the execution of a request to add parallelism and how much, as a function of load and request progress. At runtime, the service demand of each request is unknown. FM thus monitors the progress of each request and the total load, and then at the specified intervals, it adds software parallelism to the request. FM is decentralized and each request self-schedules. FM aggressively introduces parallelism under light load, but under high load, it makes efficient use of resources by judiciously executing short requests sequentially and long requests in parallel. The following key insights lead to an efficient solution.

Favoring long requests FM gives more parallelism to long requests. At moderate to high loads, FM assigns only one thread to each new request. Short requests thus execute sequentially, only ever consuming one thread. As long requests continue to execute, FM assigns them more parallelism (software threads). FM performs admission control. At moderate to high load, it may delay adding a new request in favor of adding parallelism to existing requests. Since new requests are more likely short, FM optimizes for tail latency.

Judicious use of hardware parallelism FM explicitly controls total load, neither undersubscribing nor oversubscribing hardware resources. Undersubscribing causes resources to needlessly sit idle when they could be reducing tail latency. Oversubscribing increases contention and thus tail latency, since independent requests and parallel tasks within the same request may interfere, competing for the same resources. Thus, FM slightly oversubscribes the hardware, because threads may occasionally block for synchronization or more rarely I/O. When software parallelism (threads) on occasion exceeds the hardware parallelism (cores), we boost the oldest threads priorities, so they complete without interference from younger requests. FM thus matches software parallelism to hardware parallelism.



q_r	t	t'
≤ 2	$0, d_3$	
3	$0, d_1$	$50, d_3$
4–6	$50, d_1$	$100, d_3$
≥ 7	e_1, d_1	$100, d_3$

Figure 5: Simple workload and interval table for 50 ms intervals ($t = 0, t' = 50$), number of requests (q_r), parallelism degree (d_j), for 6 cores with speedups $s(2) = 1.5$, $s(3) = 2$.

Judicious use of software parallelism Since parallelism introduces overhead and has diminishing returns, the degree to which software parallelism can reduce tail latency is a function of the service, hardware, and workload. Based on workload speedup efficiency, the service provider specifies a maximum amount of software parallelism per request that will deliver a target tail latency.

4.1 Offline Analysis

Our offline analysis takes as input a request demand profile, maximum software parallelism, and target hardware parallelism. It outputs an *interval table* indexed by load and each request's current processing time.

Example Consider the simple example in Figure 5 that uses the notation defined in Table 1. Short and long requests occur with equal probability. The sequential execution time of short requests is 50 ms and long requests is 150 ms. With parallelism degree 3, both short and long requests obtain a speedup of 2. Assume 6 cores for hardware parallelism and 50 ms intervals for simplicity. The resulting interval table consists of 4 rows indexed by the number of instantaneous requests q_r. For 1 or 2 requests, the pair $t = 0, d_3$ specifies that at time $t = 0$ every request starts immediately with parallelism d_3 degree 3, resulting in tail latency of 75 ms for long requests. Average total parallelism is 3 times active requests q_r. If $q_r = 3$, then $t = 0, d_1$, so short requests run sequentially. With $t' = 50, d_3$, long requests run sequentially until 50 ms, when parallelism degree increases to 3. Long requests finish 50 ms later with a speedup of 2 and a tail latency of 100 ms. The average parallelism per request is $(1 \times 50 + 1 \times 50 + 3 \times 50)/(50 + 100) = 1.67$ since the numbers of short and long requests are equal. With 7 or more requests ($q_r \geq 7$), $t = e_1, d_1$ indicates that new requests must wait until another request exits and then start executing sequentially.

Interval table Formally, we compute a function $f : \mathcal{R} \rightarrow \mathcal{I}$, where $\mathcal{R}$ is the set of all potential instantaneous requests

in the system and $\mathcal{I}$ is a table indexed by $q_r \in \mathcal{R}$ with each q_r corresponding to one interval selection (or schedule) Φ. A schedule Φ consists of pairs (t_i, d_j), which specify that at load q_r when a request reaches time t_i, execute it with parallelism degree d_j. If $t_0 = 0$, the request immediately starts executing. If $t_0 > 0$, the interval table is specifying admission control and the request must wait to begin its execution until the specified time. If $t_0 = e_1$, the request must wait until another request exits to begin its execution. For example, $f(3) = \{(t_0 = 0, d_1), (t_1 = 50, d_3)\}$ in Figure 5 specifies that all requests start executing immediately with one thread and after a request executes for 50 ms $(t_1 = 50, d_3)$, all requests execute with 3 degrees of software parallelism. Load changes dynamically at runtime. A particular request will consult different entries in the interval table during its execution.

Interval selection algorithm We formulate interval selection as an offline search problem, which takes as inputs the request demand profile, maximum software parallelism, target hardware parallelism *target*$_p$, and parallelism speedup. The profiled sequential and parallel request demand and parallelism speedup are collected offline. The maximum software parallelism per request is selected based on the parallelism efficiency of requests, to limit the parallelism degree to the amount effective at speeding up long requests. We select the target hardware parallelism *target*$_p$ to moderately oversubscribe the hardware threads through profiling. The search algorithm enumerates potential policies that satisfy *target*$_p$, i.e., schedules that use all available hardware resources, and then chooses ones that minimize tail latency.

To ease the presentation, we introduce an intermediate representation of a schedule as $\mathcal{S} = \{v_0, v_1, ..., v_{n-1}\}$: a request starts its first thread at time v_0, and adds parallelism from d_i to d_{i+1} after interval v_{i+1}. It is easy to see that any schedule Φ has an alternative but equivalent representation as $\mathcal{S}$. For example, for $\Phi = \{(t_0 = 0, d_1), (t_1 = 50, d_3)\}$, the equivalent $\mathcal{S} = \{0, 50, 0\}$ when the maximum software parallelism $n = 3$.

The interval selection algorithm has two parts. First, Figure 6 shows the mathematical formulation for computing parallelism and latency of requests for a given interval selection (schedule) $\mathcal{S}$ and load q_r. Equation (1) computes the total time a request takes given the time it spends in its sequential portion (if any) and time it takes in each parallel interval (if any). Equation (2) computes the request average parallelism under the schedule. Equation (3) computes the total parallelism of the system when there are q_r requests. Equation (4) and Equation (5) calculate the average and tail latency of the requests under the schedule.

Second, we enumerate all loads and all potential schedules, evaluate if they satisfy the parallelism target $target_p$, and compute tail and mean latency. If multiple schedules have the same minimum tail latency, we choose the one that minimizes the mean. Figure 7 shows the pseudocode for this search process. For each potential system load q_r, ranging from one request to the maximum system capacity, we generate all candidate schedules $\mathcal{S}$. Each component interval of a candidate schedule $v_i \in \mathcal{S}$ will take a value from 0 to y, where y is the maximum request length in the workload. We choose the schedules whose total average parallelism of all concurrent requests does not exceed the target hardware parallelism *target*$_p$, so we avoid oversubscribing the system. It is also important algorithmically: the formulation in Figure 6 calculates the request latency assuming all software parallelism nicely maps to hardware resources, which no longer holds when the total software parallelism exceeds $target_p$. Note that we do not need a lower bound on total parallelism. While optimizing tail latency, we will find schedules with total parallelism close to *target*$_p$, maximizing the utilization of all resources to reduce tail latency. Moreover, if we include a lower bound of *target*$_p$, we may not find a feasible schedule to meet it under light load, e.g., there is only one request with maximum software parallelism 4, but *target*$_p$ = 20.

Symbol	**Definition**
$r \in R$	Request profiles
seq_r	Sequential runtime of request r
d_n	Max degree n of software parallelism
$s_r(d_j)$	Speedup of r with d_j, $j \leq n$, $\forall r,\ s_r(1) = 1$
q_r	Instantaneous number of requests
$target_p$	Target hardware parallelism
$\Phi = \{(t_0, d_j), (t_1, d_{j+1})..., (t_k, d_n)\}$,	$t_i < t_{i+1}, d_j < d_{j+1}$ schedule, at t_i increase parallelism to degree d_i
$\mathcal{S} = \{v_0, v_1, ..., v_{n-1}\}$,	intermediate representation of schedule Φ start request at time v_0, and add parallelism d_i to d_{i+1} after interval v_i
time$_r(\mathcal{S})$	Execution time of r with schedule $\mathcal{S}$
ap$_r(\mathcal{S})$	Average parallelism of r with $\mathcal{S}$
ap$_R(\mathcal{S}, q_r)$	Total average parallelism of q_r of R $requests$ with $\mathcal{S}$
$time_R(\mathcal{S}, mean)$	Average latency of requests from R with $\mathcal{S}$
$time_R(\mathcal{S}, \beta - tail)$	β-tail latency of R with $\mathcal{S}$ at 99th-percentile latency with $\beta = 0.99$

Table 1: Symbols and definitions for interval selection.

From Figure 7, we can easily derive the complexity of interval table construction as

$$(y/step)^n \times req_{max} \times |R| \ .$$

This search problem is rather compute intensive. We take several steps to make it faster. First, we search in steps. If a target tail latency is in the range of 100 ms, then we limit the intervals to steps of 10 ms. Second, we only search for intervals in the range of the lifetime of the longest request. For example, when searching for intervals to increase par-

$$time_r(\mathcal{S}) = \tag{1}$$

$$\begin{cases} v_0 + seq_r & \text{if } seq_r \le v_1 \\ v_0 + v_1 + \frac{seq_r - v_1}{s_r(2)} & \text{if } v_1 < seq_r \le v_1 + s_r(2) \times v_2 \\ \ldots \\ \sum_{i=0}^{n-1} v_i + \frac{seq_r - \sum_{i=1}^{n-1} s_r(i) \times v_i}{s_r(n)} & \text{if } seq_r > \sum_{i=1}^{n-1} s_r(i) \times v_i \end{cases}$$

$$ap_r(\mathcal{S}) = \tag{2}$$

$$\begin{cases} \frac{0 \times v_0 + 1 \times seq_r}{time_r(\mathcal{S})} & \text{if } seq_r \le v_1 \\ \frac{0 \times v_0 + 1 \times v_1 + 2 \times \frac{seq_r - v_1}{s_r(2)}}{time_r(\mathcal{S})} & \text{if } v_1 < seq_r \le v_1 + s_r(2) \times v_2 \\ \ldots \\ \frac{\sum_{i=0}^{n-1} i \times v_i + n \times \frac{seq_r - \sum_{i=1}^{n-1} s_r(i) \times v_i}{s_r(n)}}{time_r(\mathcal{S})} & \text{if } seq_r > \sum_{i=1}^{n-1} s_r(i) \times v_i \end{cases}$$

$$ap_R(\mathcal{S}, q_r) = \frac{\sum_{r \in R} time_r(\mathcal{S}) \times ap_r(\mathcal{S})}{\sum_{r \in R} time_r(\mathcal{S})} \times q_r \tag{3}$$

$$time_R(\mathcal{S}, \text{mean}) = \frac{\sum_{r \in R} time_r(\mathcal{S})}{|R|} \tag{4}$$

$$time_R(\mathcal{S}, \beta\text{-tail}) = L[\lceil \beta \cdot |R| \rceil], \tag{5}$$

L is execution times $time_r(\mathcal{S})$ of all requests $r \in R$ in non-decreasing order.

Figure 6: Mathematical formulation of average parallelism, mean and tail latency for interval $\mathcal{S}$.

allelism from 1 to 4, we only search where the sum of all 3 intervals is less than the lifetime of a request. Third, if some interval does not satisfy *target*$_p$ for lower number of requests, it will not satisfy the *target*$_p$, for any higher number of requests. For accuracy, we use individual request profiles for Bing and Lucene. This process takes about four to six hours, as we process 10K - 100K requests one by one for each schedule. We may further reduce the search time by grouping requests into demand distribution bins with their frequencies, which reduces our computation time to a few minutes. The offline analysis can run daily, weekly, or at any other coarse granularity, as dictated by the characteristics of the workload.

Admission control Optimizing for *target*$_p$ does not directly control the number of active requests in the system. In particular, at high load, we want to determine whether to admit a request or to increase parallelism of the existing requests. Our search algorithm explicitly explores this case by enumerating non-zero values for the first interval (v_0). Furthermore at very high load, if the search returns the maximum value of $v_0 = y$, then the schedule specifies a new request must wait for one to exit and then starts executing with parallelism degree 1. We denote this schedule as (e_1, d_1) in an interval table.

```
Input: req_max, Maximum number of simultaneous requests
Input: y, step Maximum time interval and interval step values
for q_r := 1 to req_max step 1 do
  min_tl = min_ml = inf   Minimum tail and mean latency
  result = φ
  for v_0 = 0 to y step step do
    for v_1 = 0 to y step step do
      ...
        for v_{n-1} = 0 to y step step do
          S = (v_0, v_1, ..., v_{n-1})
          if ap_R(S, q_r) ≤ target_p
            tail = time_R(S, β-tail)
            mean = time_R(S, mean)
            if (tail < min_tl) or
              (tail = min_tl & mean < min_ml)
              min_tl = tail, min_ml = mean, result = S
  Add result to interval table entry q_r
```

Figure 7: Pseudocode for interval table construction.

4.2 Online Scheduling

The online FM scheduler is invoked when new requests enter the system and requests terminate. Each request self-schedules itself periodically based on a scheduling quanta, e.g., every 5 or 10 ms. If FM detects oversubscription of hardware parallelism, it boosts the priority of all the threads executing a long request to insure its quick completion.

FM tracks the load by computing the number of requests in the system in a synchronized variable and uses this number to index the interval table. This simple method has several advantages. First, the number of requests is fast and easy to compute compared to other indicators, such as CPU utilization. Second, in contrast with coarse-grained load indicators, such as RPS, it measures the instantaneous load. FM exploits instantaneous spare resources to avoid transient overloading. Third, FM self-corrects quickly. If the number of requests increases due to transient load, FM will index a higher row in the table, which has larger interval values and will introduce parallelism more conservatively. Similarly, when the number of requests decreases, FM will promptly introduce more parallelism for longer requests, as specified by a lower row in the table, which has shorter intervals.

Each time a request enters, FM computes the load, consults the interval table, and either starts or queues the request. When a request leaves, FM computes the load and starts a queued request (if one exists). After a request starts, it self-schedules, regularly examining the current load and its progress at the periods defined by the scheduling quanta. Each self-scheduling request indexes the interval table by the instantaneous load and if it has reached the next interval, adds parallelism accordingly. We choose relatively short scheduling quanta, less than the interval size in the table, because if requests leave the system, then FM can react quickly to add more parallelism. FM self-scheduling increases scalability of the scheduler by limiting synchronization.

FM will on occasion oversubscribe the hardware resources at high load because we choose a target hardware parallelism that exceeds the number of cores. This choice ensures that FM fully utilizes hardware resources when threads are occasionally blocked on synchronization, I/O, or terminating, but under high load will degrade tail latency if long requests must share resources with short requests. For example, operating systems generally implement a round robin scheduling to give equal resources to all the threads. To mitigate this issue, we implement *selective thread boosting*. Boosting increases the priority of all threads executing a single long request. We ensure that the number of boosted threads is always less than the number of cores by using a synchronized shared variable to count the total number of boosted threads. We only boost a request when increasing its parallelism to the maximum degree and when the resulting total number of boosted threads will be less than the number of cores. This mechanism instructs the OS to schedule these threads whenever they are ready. The longer requests will thus finish faster, which improves tail latency.

5. Evaluation

The next two sections evaluate the FM algorithm in two settings. We implement the offline table construction algorithm using around 100 lines of Python that we use for both systems. We compare both systems to the prior state-of-the-art parallelization approaches and find that FM substantially improves tail latencies over these approaches. We compare FM to the following schedulers.

Sequential (SEQ) Each request executes sequentially.

Fixed parallelism (FIX-N) Each request executes with a predefined fixed parallelism degree of N.

Adaptive (Adaptive) This scheduler [18] selects the parallelism degree for a request based on load when the request first enters the system. The parallelism degree remains constant.

Request Clairvoyant (RC) This scheduler is oracular, because it is given all requests' sequential execution times. It is an upper bound on predictive scheduling [19], which estimates request length. It selects a parallelism degree for long requests when they enter the system based on a threshold and executes other requests sequentially. The parallelism degree is constant.

SEQ and *FIX-N* are reference points, and *Adaptive* is the prior state-of-the-art for exploiting parallelism in interactive services. *RC* assumes perfect prediction of request length, but does not adapt to load. An algorithm for an optimal scheduler is unknown and at least NP hard. It is harder than bin packing, since it may divide jobs. It also requires knowledge of future request arrivals. We configure FM to use instantaneous load and each requests' progress to add parallelism and, when necessary, to use selective thread priority boosting. Because FM dynamically adapts to total load, it is significantly better than RC, which only considers the demand of individual requests when they enter the system.

6. Lucene Enterprise Search

This section presents our experimental evaluation of FM in Lucene. Apache Lucene [1] is an open-source Enterprise search engine. We configure it to execute on a single server with a corpus of 33+ million Wikipedia English Web pages [27, 38]. We use 10K search requests from the Lucene nightly regression tests as input to our offline phase and 2K search requests for running the experiments. While the nightly tests use a range of request types, we use the term requests. The client issues requests in random order following a Poisson distribution in an open loop. We vary the system load by changing the average arrival rate expressed as RPS. The index size is 10 GB and it fits in the memory of our server. Figure 2 shows the service demand distribution and the speedup profile of the workload.

6.1 Methodology

Implementation We execute 10K requests in isolation with different degrees of parallelism and gather their execution times. Each time is an average of at least 10 executions. For a specific parallelism degree, we compute the speedup of all requests and the average speedup across all requests. The sequential execution times and speedups of all requests constitute the input to the offline phase. Since the online module of FM implements admission control and assigns work to threads, we implement it within the Lucene request scheduler. Lucene is implemented in Java and we implement the scheduler in Lucene in roughly 1000 lines of Java code.

We make minor changes to the existing Lucene code base. Lucene arranges its index into segments. To add parallelism, we simply divide up the work for an individual request by these segments. We do *not* change how Lucene's default mechanisms create its index and maintain the segments. We note that this type of data organization is common to many services and makes implementing incremental parallelism simple. We extend Lucene to execute each request in parallel by adding a Java *ExecutorService* instance. We use the *ThreadPoolExecutor* class that implements *ExecutorService* and that configures the number of threads in the thread pool. Each main thread retrieves a request from a shared queue and processes the request. The main thread self-schedules periodically (every 5 ms) and checks the system load. As specified by the interval table, it increases parallelism of a request by adding threads. FM adds a thread by simply changing a field of *ThreadPoolExecutor*. Lucene starts a new thread that works on a new segment and synchronizes it with other worker threads.

Hardware We use a server with two 8-core Intel 64-bit Xeon processors (2.30 GHz) and turn off hyperthreading. (Reasoning about job interference with hyperthreading together with parallelism is beyond the scope of this pa-

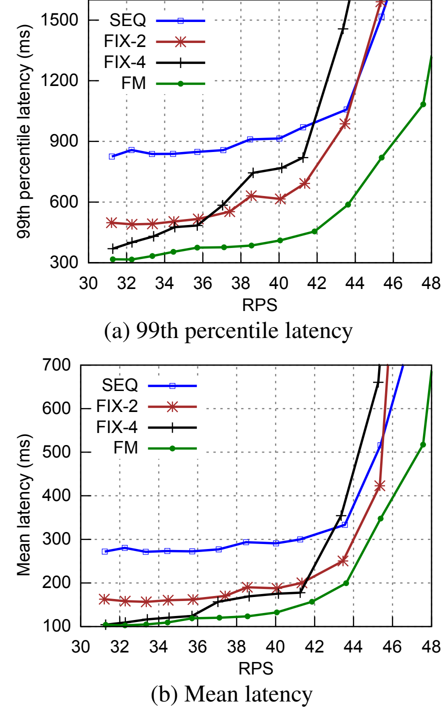
n_r	t_0	t_1	t_2	t_3
≤6	0, d_4			
7	0, d_3	75, d_4		
8	0, d_2	25, d_3	150, d_4	
9	0, d_2	50, d_3	150, d_4	
10	0, d_1	25, d_2	100, d_3	175, d_4
11	0, d_1	25, d_2	125, d_3	175, d_4
12	0, d_1	75, d_2	150, d_3	200, d_4
13	0, d_1	100, d_2	175, d_3	225, d_4
14	10, d_1	110, d_2	210, d_3	235, d_4
15	30, d_1	130, d_2	205, d_3	255, d_4
16	40, d_1	140, d_2	240, d_3	265, d_4
17	60, d_1	160, d_2	245, d_3	285, d_4
18	60, d_1	185, d_2	260, d_3	310, d_4
19	70, d_1	195, d_2	270, d_3	320, d_4
20	70, d_1	220, d_2	295, d_3	370, d_4
21	80, d_1	255, d_2	305, d_3	375, d_4
22	80, d_1	280, d_2	330, d_3	380, d_4
23	90, d_1	290, d_2	365, d_3	415, d_4
24	90, d_1	315, d_2	390, d_3	440, d_4
≥25	e_1, d_1	315, d_2	390, d_3	440, d_4

Table 2: Lucene interval table in milliseconds for 99th percentile latency with *target*$_p$ = 24 threads and maximum parallelism $n = 4$. When $t_0 = e1$, FM waits for a request to complete and then admits one waiting request. Each entry specifies execution thus far and parallelism d_k to add.

per [32].) The server has 64 GB of memory and runs Windows 8. Out of the available 16 cores, we use 15 cores to run our experiments and 1 core to run the client that generates requests. We empirically set the target hardware parallelism, *target*$_p$ = 24. We explored several values for *target*$_p$ and observed only small differences for *target*$_p \in [20, 28]$.

We report the 99th percentile latency, average latency, and CPU utilization of different policies. Latency includes both queuing delay and execution time. We use Java *NanoTime* for fine-grain time measurements. Our reported tail latency is the 99th percentile of the response times of all requests and the mean is the average response time measured over the 2K requests under various loads.

Interval selection for FM Table 2 shows the intervals generated by the offline phase for a target parallelism of 24. The table is indexed by the number or requests in the system, our metric for system load. Each column shows the time in ms at which FM adds parallelism. We use a step size of 5 ms to generate the interval table. As discussed in Section 4.1, the second column is for admission control. New requests must wait for this time before they start processing. When the load is very low (less than 6 requests), FM starts each request with parallelism degree 4. At other low loads (7 to 13 requests), FM starts a new request whenever it arrives and adds parallelism following the corresponding row. Finally,



(a) 99th percentile latency

(b) Mean latency

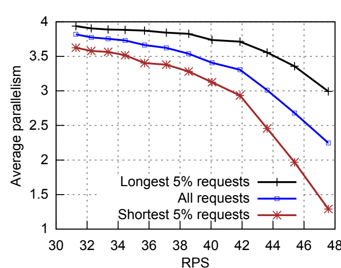
Figure 8: Lucene latency compared to fixed parallelism.

as load increases (more than 14 requests), FM delays new requests and uses longer intervals to add parallelism.

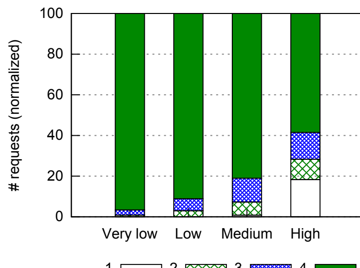
6.2 Results

Comparison to fixed parallelism policies We compare FM to sequential execution (SEQ) and a fixed degree of parallelism for each request (FIX-2 and FIX-4). Figures 8(a) and 8(b) show the 99th percentile and mean latency. FM has consistently lower tail and average latency than the other policies. At low load, many cores are available; FM is close to but better than FIX-4. Here FM aggressively parallelizes almost all requests with 4 threads (as shown in Table 2, row $n_r <= 6$). Occasional request bursts and short requests reduce intra-request parallelism on occasion even at low load. At high load, FM uses thread boosting and instantaneous load to carefully manage the degree of parallelism per request. At a medium load (40 RPS), FIX-4 is already worse than FIX-2. In contrast, FM reduces the 99th percentile latency by 33% and mean latency by 29% compared to FIX-2. At high load (43 RPS), FM reduces the 99th percentile and mean latency by 40% and 20%, respectively.

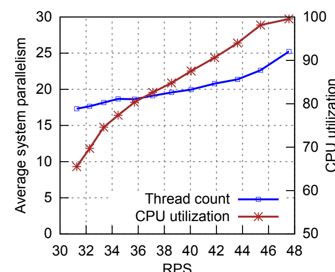
FM characteristics We now examine the parallelism degree and number of threads in FM. Figure 9(a) shows the average parallelism degree for all requests, the longest 5%, and the shortest 5%. At low load, FM assigns a high parallelism degree (almost 4) to all requests. As load increases, FM becomes less aggressive and assigns requests less parallelism. On average however, long requests have a higher degree of parallelism than short requests. At high load (47



(a) Average request parallelism

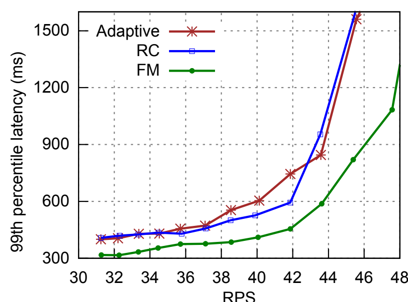


(b) Parallelism by degree by load

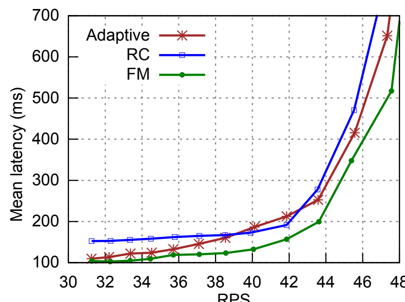


(c) Threads in the system (left y-axis) and CPU utilization (right y-axis)

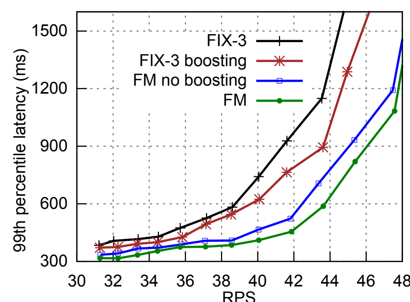
Figure 9: Lucene breakdown of parallelism degree by requests and total number of threads in the system.



(a) 99th percentile latency



(b) Mean latency



(c) Thread priority boosting

Figure 10: Lucene latency comparison with Adaptive and Predictive policies (a,b). Effect of thread priority boosting (c).

RPS), short requests mostly run sequentially with an average parallelism degree of 1.29 and long requests run with an average degree of 3. These results show that FM adapts parallelism to the system load and favors running longer requests with higher parallelism than shorter requests.

We select four RPS values, (31, 36, 40, 45) to represent ("Very low", "Low", "Medium" and "High") loads. Figure 9(b) shows the parallelism degree distributions over all requests. At very low load, most requests run with degree 4. At high load, 19% of the requests finish sequentially and 41% finish with degree 3 or less. At high load, a request that runs with parallelism degree 4 would have run with a lower parallelism degree for part of its execution. Short requests are thus likely to complete sequentially, which is the most efficient way to execute them, saving processing resources for parallelizing longer requests.

We report the average number of threads in the system and CPU utilization in Figure 9(c). The average number of threads is between 17 and 25, which is close to our target of 24. CPU utilization increases with load. At very high load (48 RPS), CPU utilization is almost 100% and the average number of threads is 25.

Comparison to the state-of-the-art policies We compare FM to state-of-the-art "Adaptive" and perfect prediction "RC" polices described in Section 5. For RC, we execute short requests sequentially and long requests with parallelism degree 4. We experimentally search for the best threshold to divide requests between short and long requests. This threshold is 225 ms.

Figure 10(a) and 10(b) show the 99th percentile latency and mean latency. FM performs consistently better than Adaptive and RC. At low load, FM executes most requests with high degrees of parallelism. Adaptive aggressively runs all requests with high degrees, but does not differentiate between long and short requests. RC misses the opportunity to parallelize requests shorter than its threshold. At medium load (40 RPS), FM reduces the 99th percentile latency by 32% and 22% compared to Adaptive and RC, respectively. Adaptive has higher latency than RC because it executes short requests in parallel, while RC runs short requests sequentially. At high load (43 RPS), RC is worse than Adaptive because it does not adapt to high load by reducing parallelism. At this load, FM reduces the tail latency by 30% and 38% compared to Adaptive and RC, respectively.

FM reduces the tail latency more than Adaptive and RC for three reasons. First, both Adaptive and RC select the parallelism degree at the start and do not dynamically adapt. FM has an extra degree of freedom. It may dynamically increase the parallelism of each individual request. Second, FM favors longer requests with additional parallelism, and shorter requests are more likely to execute sequentially especially under high load. In contrast, Adaptive does not differentiate between short and long requests, and RC, despite being clairvoyant of the request service demand, uses a static threshold,

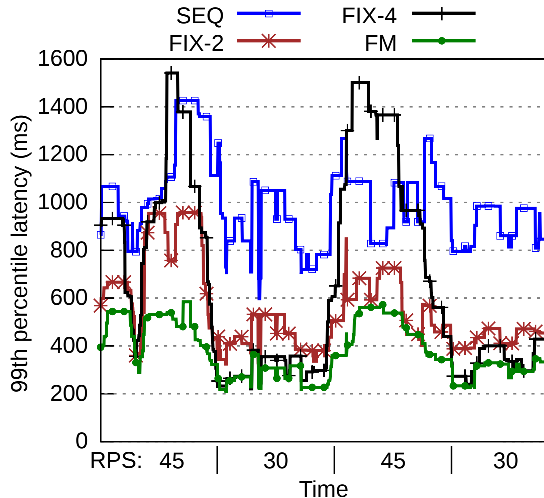


Figure 11: 99th percentile latency for the last 100 requests in a 500-request quanta, while varying the load in Lucene.

fixed parallelism degree, and ignores system load. Third, FM exploits thread boosting to further reduce the tail latency.

Selective thread priority boosting To study benefits of boosting, we disable thread boosting in FM and add it to a fixed parallelism configuration. Since the average parallelism of FM ranges from 2.2 to 3.8 (Figure 9(a)), we select fixed degree 3 (FIX-3) and compare FM with FIX-3 with boosting. Figure 10(c) shows the 99th percentile latencies. Selective thread priority boosting improves the tail latency for both FM and FIX-3. In particular, thread priority boosting reduces the tail latency of FM by 12% at 40 RPS, and by 16% at 43 RPS. Selective thread priority boosting ensures that short requests do not interfere with long requests, reducing tail latency.

Load variation To study the effect of load variation and burstiness on tail latency, we configure the client to vary the request submission rate in quick succession. The client varies load in four quanta: high (45 RPS) to low (30 RPS) to high (45 RPS) to low (30 RPS). The client submits 500 requests in each quanta. Figure 11 compares 99th percentile latencies of the last 100 requests in each quanta for FM, SEQ, FIX-2, and FIX-4. The performance of SEQ is usually the worst, but is slightly better than FIX-4 at the beginning of a load burst. FM adapts well to changes in load and consistently achieves the best tail latency. FIX-4 is the most aggressive and performs almost as well as FM at low load, but performs substantially worse during the load burst. Comparing FIX-2 and FIX-4, we see that FIX-2 performs better than FIX-4 during the high load quanta, but worse at low load. This experiment shows that FM is stable, responding quickly and smoothly to highly variable load. FM tunes its scheduling policy to the load to reduce tail latency.

7. Bing Web Search

This section presents our evaluation of FM in the index servers of Microsoft Bing Web search with a production index and query log. The Bing index serving system consists of aggregators and index serving nodes (ISNs). An entire index, containing information about billions of Web documents, is document-sharded [3] and distributed among hundreds of ISNs. When a user sends a request and the result is not cached, the aggregator propagates the request to all ISNs hosting the index. Each ISN searches its fragment of the index and returns the top-k most relevant results to the aggregator. Because Bing is compute-bound at the ISN servers (see Section 2), a long latency at any ISN manifests as a slow response [9]. To reduce the total tail latency, each ISN must reduce its tail latency. For example, assuming the aggregator has 10 ISNs, if we want to process 90% of user requests within 100 ms, then each ISN needs to reply within 100 ms with probability around 0.99. In other words, for a total latency of 100 ms at the 90th-percentile response time, the response time of each ISN must be at most 100 ms at the 99th-percentile. These results motivate our evaluation of Bing on an individual server.

7.1 Methodology

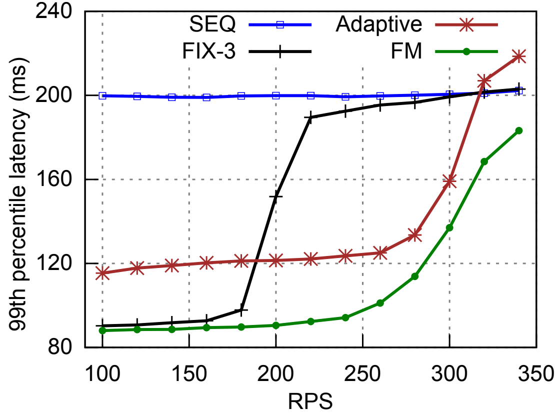
Implementation We implement FM in the Bing request dispatch code. To eliminate thread creation time, we use a thread pool. Bing organizes its indexes in groups and thus adding incremental parallelism is not onerous. We configure two servers. One server is the index server, which executes requests. A second server is the aggregator, which sends requests to the index server by replaying a trace containing 30K Bing production user requests from 2013. We vary system load by changing the average request arrival rate in RPS. This version of FM does not perform thread boosting. FM uses a target number of threads, $target_p = 16$, a slightly higher number than the 12 available cores (see below). As Section 2 showed, the efficiency of parallelism drops significantly at degree 4, thus we configure FM to increase the parallelism degree of a request up to 3.

Hardware and OS For the index service, we use a server with two 2.27 GHz 6-core Intel 64-bit Xeon processors and 32 GB of main memory with Windows Server 2012. The ISN manages a 160 GB index partition on an SSD, and uses 17 GB of its memory to cache index pages from the SSD.

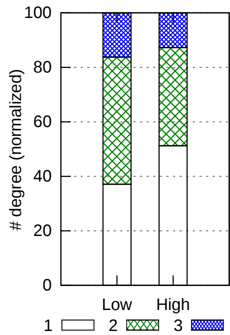
7.2 Results

We compare FM to fixed parallelism with degree 3 (FIX-3), Adaptive, and SEQ. The production version of FIX-3 applies load protection: it parallelizes each request using degree 3 when the total number of requests in the system is less than 30. Otherwise, it runs requests sequentially.

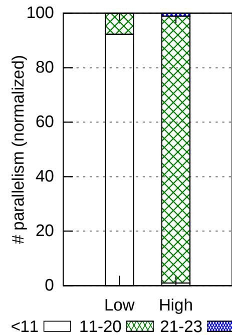
Figure 12 presents the 99th-percentile latency over different loads. These experimental results show that FM consistently has the lowest tail latency. In particular, it effectively



(a) Comparison to SEQ, FIX-3, and Adaptive



(b) Request parallelism



(c) Thread Distribution

Figure 12: Bing comparisons and parallelism.

reduces the tail latency at moderate to high load. For example, up to 260 RPS, FM exhibits a 99th percentile latency of 100 ms. In contrast after 150 RPS, FIX-3 has latencies of over 200 ms because it oversubscribes the resources, parallelizing all requests regardless of system load and the request length. FM also performs better than Adaptive at all load levels. For example, at 180 RPS, the tail latency reduction over Adaptive is 26% and at 260 RPS, the improvement is 24%. FM reduces the tail latency more than Adaptive because it runs long requests with higher parallelism degrees than short requests. These results show that FM enables the provider to service the same load with 42% fewer servers compared to Adaptive for a target tail latency of 120 ms.

To study FM parallelism, we select two representative load values: 200 RPS for "Low" and 280 RPS for "High" load. Figure 12(b) shows the distribution of parallelism per request. Figure 12(c) shows total threads, which are underutilized at low load and match the target load of 16 at high load, as expected. Comparing low to high load, FM uses less intra-request parallelism as load (system parallelism) increases. About 35% of requests execute sequentially at low load, whereas over 50% are sequential at high load.

8. Related Work

This section compares incremental parallelism to related work on parallelism for minimizing latency.

Parallelizing a single job to reduce latency. Many systems adapt parallelism to run-time variability and hardware characteristics [4, 8, 20, 23, 24, 30, 34, 36]. They focus on the execution of a single job to reduce execution time and improve energy efficiency. However, they do not consider a server system running concurrent jobs. Simply applying parallelism to minimize the execution time of every single job will not minimize the response time for concurrent jobs. Our work focuses on a server environment with many requests where parallelizing one request may affect others. We choose the degree of parallelism for a request based on both its impact on the request itself and on other requests.

Parallelization in a multiprogrammed environment for reducing mean response time. Adaptively sharing resources among parallel jobs has been studied empirically [11, 29] and theoretically [2, 16]. However, these studies focus on a multiprogrammed environment, in which jobs have different characteristics unknown *a priori* to the scheduler. The scheduler learns the job characteristics and adjusts the degree of parallelism as the job executes. In contrast, interactive requests in server systems share characteristics that we describe in Section 2 and then exploit. For example, the request service demand distribution and parallelism efficiency are stable over time, and therefore we profile them and use the results to improve scheduling decisions. Moreover, as many requests complete quickly, the scheduler must act quickly. We perform offline processing so that FM makes efficient online decisions. Finally, this prior work focuses only on reducing mean response time, whereas FM reduces tail latency, which requires different techniques.

Parallelization in interactive server systems for reducing mean and tail response time. Adaptive resource allocation for server systems [35, 37] focuses on allocating resources dynamically to different components of the server, while still executing each request sequentially. Raman *et al.* proposed an API and runtime system for dynamic parallelism [31], in which developers express parallelism options and goals, such as minimizing mean response time. The runtime dynamically chooses the degree of parallelism to meet the goals, and does not change it during the execution of the requests. Jeon *et al.* [18] proposed a dynamic parallelization algorithm to reduce the average response time of Web search queries. The algorithm decides the degree of parallelism for each request before the request starts, based on the system load and the average speedup of the requests. Neither approach [18, 31] targets tail latencies. Moreover, since the service demand of each request is typically unknown before the request starts, these approaches cannot differentiate long requests from short ones. Thus, they oversubscribe resources under moderate or high load, and are ineffective at reducing

the tail latency. We implement the "Adaptive" algorithm [18] in Section 6, and show FM provides lower tail latency.

To specifically reduce tail latency, Jeon *et al.* [19] predict service demand of Web search requests using machine learning and parallelize the requests predicted to be long, regardless of the system load and workload characteristics. FM considers all of these factors, as well as instantaneous load to deliver better results. Even with a *perfect* predictor, the "Request Clairvoyant" (RC) approach does not outperform FM as shown in Section 6. RC provides an upper-bound on the performance of the predictive technique [19], but it is hard to predict the request service demand in many applications. FM performs better than RC, across all loads, because it continually adapts to total load without requiring prediction.

9. Conclusion

This paper introduces a new parallelization strategy called Few-to-Many (FM) incremental parallelism for reducing high-percentile latency in interactive services. FM uses the demand distribution and a target hardware parallelism to determine dynamically when and how many software threads to add to each request. At runtime, it uses system load and request progress to add parallelism, adapting each request individually. We implement FM in two search engines, Lucene and Bing, and evaluate it using production request traces and service demand profiles. Our results show that FM can significantly reduce tail latencies, and help service providers to reduce their infrastructure costs.

These results and our experience suggest that FM should be extremely useful in practice. In fact, Bing engineers have already deployed a basic version of FM in the production Bing system on thousands of servers.

A. Appendix: Proof of Theorem 1

Suppose a request needs to meet a β-th percentile latency of d. We show any optimal policy that minimizes the average resource usage assigns parallelism in non-decreasing order up to *the latency constraint at time* d. (Requests that take longer than d are in higher percentiles and, thus, may be completed with any degree of parallelism.)

We denote the service demand CDF of request sequential execution times as F and speedup with parallelism i as s_i. As requests have sublinear speedup, the parallelism efficiency decreases with increasing parallelism degree, i.e., $s_i/i > s_j/j$, if $i < j$. Using F, we find the β-th percentile service demand of requests, denoted as w, i.e., $w = F^{-1}(\beta)$. For a schedule to meet β-th percentile latency of d, we want to ensure the β-th percentile longest request can be completed by d, i.e., a work amount w is completed by d.

Let's denote $\mathcal{S}$ as a schedule that specifies how we parallelize a request. For a given piece of work at the x-th cycle where $x \in (0, w]$, and a schedule $\mathcal{S}$, if $\mathcal{S}(x) = i$, this work is parallelized using degree i. The speed to process the work is therefore $s_{\mathcal{S}(x)} = s_i$. We write the resource usage minimization problem as,

$$\min_{\mathcal{S}} \int_0^w [1 - F(x)] \times \frac{\mathcal{S}(x)}{s_{\mathcal{S}(x)}} dx \qquad (6)$$

$$\textit{s.t.} \qquad \int_0^w \frac{1}{s_{\mathcal{S}(x)}} dx \le d \, . \qquad (7)$$

Here, the integral in the objective function 6 computes the expected amount of resources a request consumes up to work w and latency d. Constraint 7 guarantees that the processing time of the β-th percentile request is bounded by d.

We now prove the theorem by contradiction. Suppose that there is an optimal schedule $\mathcal{S}'$ that gives higher parallelism to a request earlier and later decreases its parallelism. Thus, there exist x_1 and x_2 such that $0 \le x_1 < x_1 + dx \le x_2 < x_2 + dx \le w$ and $\mathcal{S}'(x_1') > \mathcal{S}'(x_2')$, where $x_1' \in [x_1, x_1 + dx]$, $x_2' \in [x_2, x_2 + dx]$ and dx is a sufficiently small positive number.

Since we assume sublinear speedup, i.e., $s_i/i > s_j/j$ if $i < j$, the following inequality holds:

$$\begin{aligned} & [1 - F(x_1')] \times \left[\frac{\mathcal{S}'(x_1')}{s_{\mathcal{S}'}(x_1')} - \frac{\mathcal{S}'(x_2')}{s_{\mathcal{S}'}(x_2')} \right] \\ & + [1 - F(x_2')] \times \left[\frac{\mathcal{S}'(x_2')}{s_{\mathcal{S}'}(x_2')} - \frac{\mathcal{S}'(x_1')}{s_{\mathcal{S}'}(x_1')} \right] \\ = & \left[\frac{\mathcal{S}'(x_1')}{s_{\mathcal{S}'}(x_1')} - \frac{\mathcal{S}'(x_2')}{s_{\mathcal{S}'}(x_2')} \right] \times [F(x_2') - F(x_1')] > 0 \end{aligned}$$

Thus, $[1 - F(x_1')] \times \frac{\mathcal{S}'(x_1')}{s_{\mathcal{S}'}(x_1')} + [1 - F(x_2')] \times \frac{\mathcal{S}'(x_2')}{s_{\mathcal{S}'}(x_2')}] > [1 - F(x_1')] \times \frac{\mathcal{S}'(x_2')}{s_{\mathcal{S}'}(x_2')} + [1 - F(x_2')] \times \frac{\mathcal{S}'(x_1')}{s_{\mathcal{S}'}(x_1')}]$. Following the optimization objective in Eqn. 6, the expected resource usage is reduced by exchanging the order of parallelism degree between the x_1'-th cycle and the x_2'-th cycle, while keeping the rest of the schedule $\mathcal{S}'$ unchanged. This contradicts the assumption that $\mathcal{S}'$ minimizes Eqn. 6 and therefore proves Theorem 1. ■

Acknowledgements We thank Xi Yang for helping us with the implementation and configuration of Lucene. We thank our shepherd, Jason Flinn, anonymous reviewers, and Jing Li for their helpful comments and suggestions. We thank Gregg McKnight for his feedback and support of this work.

References

[1] Apache Lucene. `http://lucene.apache.org/`. Retreived July 2014.

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

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

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

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55 – 69, 1996.

[6] B. Cahoon, K. S. McKinley, and Z. Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Transactions on Information Systems (TOIS)*, 18(1):1–43, 2000.

[7] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 57–66, 2008.

[8] 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.

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

[10] 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.

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

[12] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison Wesley Professional, 2006.

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

[14] J. Hamilton. The cost of latency, 2009. `http://perspectives.mvdirona.com/2009/10/31/-TheCostOfLatency.aspx`.

[15] J. Hamilton. Overall data center costs, 2010. `http://perspectives.mvdirona.com/2010/09/18-/OverallDataCenterCosts.aspx`.

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

[17] 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.

[18] 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.

[19] 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.

[20] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for SMT multiprocessor architectures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 236–246, 2005.

[21] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J. P. Laulajainen, R. Carmichael, V. Poulopoulos, A. Laikari, P. Perälä, A. De Gloria, and C. Bouras. Platform for distributed 3D gaming. *International Journal of Computer Games Technology*, 2009:1:1–1:15, 2009.

[22] 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)*, 2015.

[23] W. Ko, M. N. Yankelevsky, D. S. Nikolopoulos, and C. D. Polychronopoulos. Effective cross-platform, multilevel parallelism via dynamic adaptive execution. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 130, 2002.

[24] 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.

[25] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 227–242, 2009.

[26] 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.

[27] Lucene Nightly Benchmarks. http://people.apache.org-/˜mikemccand/lucenebench. Retreived June 2014.

[28] 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.

[29] 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.

[30] 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.

[31] 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)*, pages 26–37, 2011.

[32] S. Ren, Y. He, S. Elnikety, and K. McKinley. Exploiting processor heterogeneity in interactive services. In *USENIX International Conference on Autonomic Computing (ICAC)*, pages 45–58, 2013.

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

[34] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance exe-

cution of multi-threaded workloads on CMPs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–286, 2008.

[35] G. Upadhyaya, V. S. Pai, and S. P. Midkiff. Expressing and exploiting concurrency in networked applications with aspen. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13–23, 2007.

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

[37] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243, 2001.

[38] Wikipedia: Database download. http://en.wikipedia.org/-wiki/wikipedia:database_download#english-language_wikipedia/. Retreived May 2014.

[39] 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.