An Efficient Scheduler for Task-Parallel Interactive Applications

Kyle Singer
Washington University in St. Louis
St. Louis, Missouri, USA
kdsinger@wustl.edu

Kunal Agrawal
Washington University in St. Louis
St. Louis, Missouri, USA
kunal@wustl.edu

I-Ting Angelina Lee
Washington University in St. Louis
St. Louis, Missouri, USA
angelee@wustl.edu

ABSTRACT

Modern software is often ***interactive*** — applications communicate frequently with the external world. For such applications, responsiveness — how quickly they respond to requests — is as important as throughput. Efficiently implementing these applications using traditional processes or static threads is difficult and error-prone. Task parallelism has the potential to significantly simplify the implementation of these applications — it allows the programmer to express the high-level logical flow of the program and letting the scheduler handle the low level details of scheduling, synchronization, and asynchronous I/O operations.

Researchers have begun to study how to best support such interactive applications where different components have different responsiveness requirements on ***priority-oriented*** task-parallel platforms. We propose Prompt I-Cilk, a practically efficient scheduler for priority-oriented task-parallel interactive applications. Our scheduler exhibits superior performance when compared to the state-of-the-art scheduler design, including on the Memcached object server, a large scale real-world interactive applications that we ported to run on a task-parallel platform.

Our scheduler design defies the conventional folk wisdom on how to schedule task-parallel code — we moved away from randomized work stealing, and we implemented "prompt" scheduling with frequent checking of core-to-priority-level assignments. We show that such design choices make sense based on the workload characteristics of the parallel interactive applications we tested.

CCS CONCEPTS

• **Software and its engineering → Software performance**; *Multithreading*; • **Computing methodologies** → *Shared memory algorithms.*

KEYWORDS

work stealing, interactive applications, priority scheduling, adaptive scheduling, task parallelism

ACM Reference Format:
Kyle Singer, Kunal Agrawal, and I-Ting Angelina Lee. 2023. An Efficient Scheduler for Task-Parallel Interactive Applications. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3558481.3591092

SPAA '23, June 17–19, 2023, Orlando, FL, USA

ACM ISBN 978-1-4503-9545-8/23/06.
https://doi.org/10.1145/3558481.3591092

1 INTRODUCTION

Much of modern software is service-oriented, long running, and ***interactive***, where applications frequently communicate with the external world through the use of Inputs / Outputs (I/Os). Examples include modern desktop software such as email clients, web services hosted on cloud platforms such as video streaming services, or game servers. Unlike traditional parallel applications, which mainly care about high throughput (i.e., doing as much work as possible as quickly as possible), such interactive applications commonly have additional performance criteria, such as ***responsiveness***, where some components — generally the components that interact with the external world — require low response time because humans or other applications are waiting on the response. Crucially, different components may have different responsiveness requirements.

Task-parallelism has been a popular programming paradigm for traditional, non-interactive parallel applications for some time now. On traditional task-parallel platforms, the programmer simply expresses the *logical* parallelism of the computation via high-level parallel control constructs, and lets an underlying runtime system automate load balancing and synchronization. Task parallelism can simplify parallel programming by allowing the programmer to focus on implementing the parallel semantics of the application rather than the scheduling logic, which is left to the runtime.

While most interactive applications are currently programmed using persistent threads such as pthreads, in recent years, researchers have begun to study how to best support interactive applications, where different components have different responsiveness requirements, on ***priority-oriented task-parallel platforms*** (e.g., [29–32, 41]). Specifically, to achieve responsiveness, this work assigns priorities to tasks so the scheduler can distinguish between tasks with stringent response-time requirements and those with looser requirements. Well-designed task-parallel platforms have a potential to significantly simplify the programming of interactive applications particularly due to the presence of frequent I/Os within these applications (see Section 2).

In order to do so, however, one must provide efficient scheduling support specifically designed for interactive applications, which is the goal of this work. Most prior work [29–32] on this topic focuses on type systems to check for ***priority inversions*** where a higher-priority task must wait for a lower-priority task to complete. Assuming the program type checks, the corresponding cost semantics can guarantee bounded response times of tasks assuming a "prompt" and greedy / work-conserving scheduler. Here, we say that the scheduler is ***prompt*** if it follows strict prioritization such that no core does low-priority work when high-priority work exists, and it is ***greedy*** or ***work-conserving*** if no core is idle when some work is available. Even though some prior work provides a prototype prompt scheduler for small applications written in ML, the focus of prior work is not on how to design a low-overhead and scalable scheduler for large-scale task-parallel interactive applications.

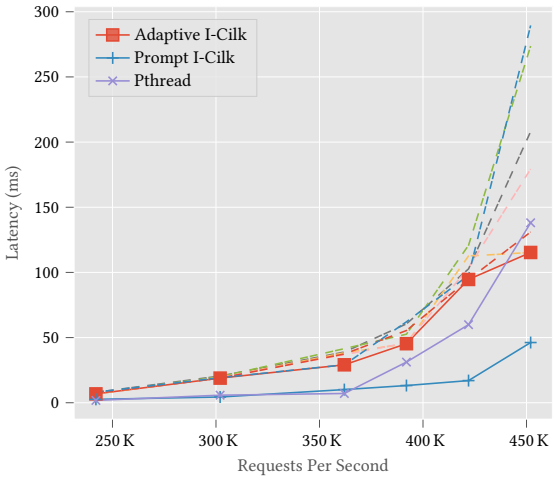


Figure 1: The 99^{th} percentile latency of handling memcached requests using Pthread, Adaptive I-Cilk, and Prompt I-Cilk versions of the memcached server. For Adaptive I-Cilk, a parameter sweep was performed and each data point shown is drawn from the best runtime parameter set for the given RPS. The dashed lines show the results for each set of parameters. Pthread and Prompt I-Cilk do not require parameter sweeps.

Recent work by Singer et al. [41], referred to as Adaptive I-Cilk, proposed an online adaptive scheduler for priority-oriented task-parallel computations written in C++. We wanted to evaluate how well the Adaptive I-Cilk scheduler works when running real-world application workloads, so we ported the Memcached object caching server [2] to run on Adaptive I-Cilk. As far as we are aware, no one has conducted empirical evaluation on such a large-scale task-parallel code for interactive applications. The Memcached server consists of over 18,000 lines of C/C++ code and the porting effort is non-trivial — it involved converting all the I/Os within the application, removing the event loop in favor of something more streamlined, and removing the concurrency mechanisms of pthreads and rewriting them to utilize task-parallel constructs (i.e., `spawn`, `sync`, and futures [19]). Unfortunately, when compared to the original pthreaded implementation the Adaptive I-Cilk scheduler fell short.

Figure 1 shows the performance of pthreaded, Adaptive I-Cilk, and our proposed new scheduler Prompt I-Cilk (discussed later) running the Memcached server[1] for the 99^{th} percentile latencies across different requests per second (RPS) values (higher RPS means heavier server load). As this data shows, Adaptive I-Cilk performs significantly worse than the pthreaded implementation.

In order to understand the shortcomings of Adaptive I-Cilk, let us briefly review its design. At a high-level Adaptive I-Cilk tries to approximate prompt scheduling by preferentially executing high priority tasks, while simultaneously trying to maintain efficiency and low overhead. In order to do so, the scheduler in Adaptive I-Cilk utilizes a two-level scheduling strategy. The top-level processor allocating scheduler, given runtime executions, adaptively determines how to best assign cores to priority levels; it re-evaluates the core assignments at every scheduling quantum (i.e., at every L time steps where L is an input parameter to the runtime) based on how well each priority level utilized its assigned cores from the previous quantum. At the bottom-level, a randomized work-stealing scheduler is used to schedule tasks for a given priority level.[2]

Intuitively, the scheduling design in Adaptive I-Cilk makes sense. The re-evaluation of core assignments and moving workers across different priority levels incurs overhead; therefore, the granularity at which Adaptive I-Cilk does this can be adjusted to be infrequent enough to minimize overhead while being frequent enough to approximate promptness. In addition, randomized work-stealing has been shown to be provably efficient [6, 8] and incurs little scheduling overhead in practice [17]. So why does this well-designed scheduler perform poorly?

Upon investigation, we realized that a key contributor to the superior performance of the pthreaded version is its "aging heuristic." Throughout the executions, a connection can become blocked due to its I/O operation and unblocked when the I/O operation completes. To obtain performance, the pthreaded version uses asynchronous I/O and event-driven style programming to time-multiplex among multiple concurrent client connections. However, its implementation preferentially handles execution contexts that got unblocked earlier over the later ones. In contrast, while Adaptive I-Cilk time-multiplexes among concurrent connections, the randomized work stealing means that the unblocked execution contexts are resumed in a randomized order.

Moreover, the Adaptive I-Cilk scheduler has multiple runtime parameters and the performance is extremely sensitive to these parameters. A parameter sweep is necessary to choose the right values, which can vary widely between applications and even within the same application (Figure 1 shows the best possible performance after the parameter sweep and paints an optimistic picture of Adaptive I-Cilk; each data point shown uses a different runtime parameter, the set that performs the best for the given RPS).

Based on our experience evaluating Memcached, we decided to rethink the scheduler design from scratch. Our resulting scheduler, Prompt I-Cilk, is surprisingly simple and yet exhibits superior performance with respect to responsiveness for Memcached, as can be seen in Figure 1, and for the other application benchmarks tested. Prompt I-Cilk does not use randomized work stealing. Instead, it utilizes something between work stealing and work sharing, and keeps a centralized deque pool to keep track of all execution contexts for a given priority level; this allows it to easily implement the aging heuristic. In Prompt I-Cilk, each worker thread also checks, somewhat frequently, whether it should switch levels, thus eliminating the need for a processor allocating scheduler and parameters.

The design of Prompt I-Cilk moves away from the conventional wisdom about task-parallel scheduler design. The first bit of wisdom was that randomized work-stealing provides better performance than centralized deterministic policies. However, in large-scale interactive applications there are many parallel contexts; therefore, contention is less of an issue. It is more important to find the "oldest" work, which is harder to do with a randomized policy. The second bit of wisdom was the "work-first" principle — first articulated in [17] — which has long been considered important in scheduler

[1]The experimental setup is in Section 5.

[2]Even though their analysis assumes a greedy bottom-level scheduler for each priority level, the implementation utilizes a randomized work-stealing scheduler.

design; it states that we should avoid adding overhead when a worker has local work on its deque. It turns out that to minimize latency for high-priority tasks, it is better to abandon the work-first principle in favor of increasing promptness; therefore Prompt I-Cilk frequently checks to ensure that workers are working on the "correct" tasks. In particular, we make the following contributions:

- We ported the Memcached server, a large-scale real-world interactive application, to run on task-parallel platforms and conducted an empirical evaluation with it. Using Memcached as an example application, we argue that task-parallelism can simplify implementations of interactive applications. Using empirical evaluation, we show that classic work stealing with randomization may not be ideal for the Memcached's workload characteristics (Section 3).
- Based on our experience with the Memcached server, we propose Prompt I-Cilk, an approximation of the prompt-greedy scheduler proposed in prior work and often thought of as infeasible to implement efficiently; its design involves using a centralized pool with aging heuristics and frequent checking of priority-level assignment (Section 4); and
- We conducted a detailed empirical evaluation of Prompt I-Cilk using three application benchmarks including Memcached. Empirical results indicate 'that Prompt I-Cilk, albeit simple, is a highly effective scheduler for these applications (Section 5).

2 PRELIMINARIES

This section briefly overviews the linguistics of Prompt I-Cilk and Adaptive I-Cilk— both systems are identical in terms of linguistic support and differ only in terms of scheduler design. The section also briefly describes the scheduler design for Adaptive I-Cilk.

Language Support for Interactive Applications

Like prior work on task parallelism, Prompt I-Cilk provides constructs for both fork-join parallelism and futures. The keyword `spawn` can precede a function invocation, indicating that the invoked function may execute in parallel with the continuation of the caller. The keyword `sync` serves as the counterpart for `spawn` and indicates that the control cannot pass beyond `sync` until all previously spawned functions have returned. The keyword `fut-create` is similar to `spawn`, but creates a future and returns a ***future handle***, which can be used to wait for the termination of the function invoked and query its result using `get`. Futures allow a more expressive form of parallelism relative to `spawn` and `sync`, because a future handle can escape the lexical scope in which it is created, allowing arbitrary parallel subcomputations to wait for its associated function invocation. These constructs express the logical parallelism; the underlying scheduler is responsible for efficient scheduling of the generated work.

Like prior work [40], I/Os in Prompt I-Cilk are expressed using I/O futures, a special type of future. Therefore, Prompt I-Cilk essentially provides a *synchronous interface* for I/O operations which greatly simplifies the application logic. Generally synchronous I/Os are inefficient because workers may spin waiting for an I/O; however, because the Prompt I-Cilk runtime system understands these I/Os, it transparently and automatically time-multiplexes among concurrent subcomputations whenever the I/O operations block providing the *performance advantages of asynchronous I/Os*. In addition, when a task is created (via a `spawn` or `fut-create`), the programmer can specify its priority. The type system then checks for priority inversions — whether any higher priority task might wait for a lower priority task to complete — and the runtime system uses these priorities to preferentially execute higher priority tasks.

The Adaptive I-Cilk Scheduler Design

Although Prompt I-Cilk and Adaptive I-Cilk differ in important aspects of how they schedule tasks within a priority level, they both utilize deques to store execution contexts and operate on deques similarly. At any time, a worker has an ***active deque***, and when the worker encounters a `spawn` or `fut-create`, the worker simply pushes the continuation of the spawning parent onto the bottom of its own deque, and continues to execute the spawned subroutine or future (assuming the subroutine is invoked with the same level of priority).[3] If a worker encounters a failed `sync`, just as in classic work-stealing, the deque must be empty, and the worker tries to find work via work stealing. If a worker encounters a failed `get`, the deque may or may not still contain ready work; regardless, the worker ***suspends*** the deque and tries to find work via work stealing. Once the `get` that causes the deque to be suspended becomes ready, the deque becomes ***resumable*** and can be resumed by any worker looking for work. If a suspended deque has ready nodes which can be stolen, it is called a ***stealable deque***.

This design in both Prompt I-Cilk and Adaptive I-Cilk is inspired by proactive work-stealing [42], which was designed to handle futures (and later extended to handle I/O futures [40]) efficiently. Proactive work-stealing is designed to provide efficient execution via reducing the number of "deviations" in the presence of futures; intuitively, reducing the number of deviations acts as a proxy for both reducing the scheduling overhead and improving locality. However, unlike classical work-stealing [8] (where the number of deques is the same as the number of workers), proactive work-stealing can generate many more deques than workers.

Adaptive I-Cilk and Prompt I-Cilk differ considerably, however, in most other aspects of their design — in particular, how the workers are moved between different priority levels and how the deques are organized. Adaptive I-Cilk [41] utilizes a two-level scheduling system. At the bottom level, each priority has its own a randomized work-stealing scheduler. Since the number of deques is larger than the number of workers, instead of each worker having a single deque, each worker has its own ***deque pool*** — a set of deques. Each worker has one ***active*** deque, the deque it is currently working on; but may also have several suspended and resumable deques in its deque pool. When a worker tries to steal, it first picks a random worker and then picks a random deque in that worker's deque pool as a victim. The Adaptive I-Cilk scheduler tries to ensure that the probability of stealing from a each deque is about the same by performing rebalancing of deques among workers from time to time. In addition, recall that any steals from suspended deques which are not stealable are completely unproductive; therefore, Adaptive I-Cilk removes these non-stealable suspended deques from workers' deque pools and reinserts them when they become resumable.

[3] If the spawned subroutine belongs to a different priority level, a deque is generated to store the subroutine and tossed to the appropriate priority level.

At the top-level, Adaptive I-Cilk has a centralized scheduler that moves workers between different priorities. At any particular time, each bottom-level scheduler is assigned a certain number of workers by the top-level scheduler and this allocation can change at quantum boundaries. At the end of each quantum, the system measures the ***utilization*** — percentage of the processing used for application work — of each bottom level scheduler and the top-level scheduler decides the allocation for each bottom-level scheduler for the next quantum based on this quantity. The intuition behind this design in Adaptive I-Cilk is as follows: If a lot of work is available at a particular priority level (as measured by utilization), more workers will be allocated to that level (with preference given to higher priorities). It simultaneously tries to minimize the overhead of moving workers between priority levels by doing so only infrequently.

3 MOTIVATING EXAMPLE: THE MEMCACHED SERVER

This section reports our experience porting and running the Memcached object server [2], a widely used interactive application, on Cilk-based task-parallel platforms. We use Memcached as an example application to explain how task parallelism can simplify the implementation of interactive applications. We also use it to explain how the typical workload characteristics of interactive applications may differ from traditional parallel applications, thereby making randomized work-stealing non-ideal for these applications.

The Memcached Server Overview

The Memcached server provides an in-memory key-value store for small objects; it is widely used as a caching backend for many performance-critical systems and is designed to be scalable. At a high-level, it maintains a hash table; within each bucket, the objects are organized in (approximately) least-recently-used (LRU) order such that recently accessed objects are likely to be faster to access. One can configure Memcached to support hash table expansion and writing of the cached content to an external persistent storage.

The original implementation is written in C with POSIX threads [22] and organized as follows. Upon startup, the main thread creates several background threads and a fixed number of worker threads to handle client connections. The background threads perform tasks such as reorganizing the items in a bucket to maintain the LRU policy in the cache, assisting hash table expansion, crawling through the cache to collect statistics, and writing in-memory cache content to the external storage (if configured to do so); these are designed to run periodically. The main thread listens for requests from new clients; once a client is connected, a specific worker thread is assigned to handle its requests.

The Memcached server uses event-driven style programming with asynchronous I/O operations via the `libevent` library [1]. A worker thread time-multiplexes among multiple client connections at any given time via an event loop to handle events from multiple concurrent client connections. Each client, upon connecting to the server, is assigned to a particular worker, and a callback function is registered with the `libevent` library for events associated with that particular client connection. A worker thread never waits for a blocking I/O operation when handling a request. Rather, upon encountering a blocking I/O operation for a request r, the worker thread returns from the callback function to handle requests from other connections, and only comes back to resume handling request r when the I/O can complete, which generates an event and causes the same callback function to be invoked again.

Due to asynchronous I/Os and event-driven style programming, the control flow of the application is extremely complex. A given request r might require multiple I/O operations before completion (with some computation interleaved between them). Thus, when the worker thread handling r encounters a blocking I/O and leaves to handle other requests, it must leave enough bookkeeping information so that it knows which I/O operation it blocked on when it returns to this callback function. Therefore, the callback function effectively encodes a large state machine using a switch-statement in a loop, where a read can transition to multiple states depending on what the input is and what state the request was in previously. Such control flow obscures the application logic, as the logic for handling a single request is scattered across different switch statement cases.

As mentioned in Section 1, the pthreaded version implements an ***aging heuristic*** that handles execution contexts in the order in which they become unblocked. This aging heuristic is implemented implicitly due to how `libevent` and the underlying OS handles asynchronous I/O operations. As the I/O operations become ready, the OS detects the available I/O events and returns them to `libevent` in the same order. Thus, as worker threads process the I/O events from `libevent`, they effectively process them in the same order as they became ready, at least among clients handled by the same worker thread. There are exceptions to this aging heuristic: if a particular client connection has multiple outstanding requests all of which can be processed without blocking, then the worker thread will process them together upon entering the callback function corresponding to that connection, up to some threshold before the worker thread voluntarily "yields" and returns from that callback so as to not starve other connections.

Simplifications using Task Parallelism

We rewrote the Memcached server using task-parallel control constructs and found that the resulting control flow is much simpler and exposes the underlying program logic. Specifically, we removed the event loop and we rewrote the giant callback function that encodes the state machine to handle client requests with I/O futures, which provide a synchronous I/O interface with the performance of asynchronous I/O. This rewrite turns the function that handles client requests into something much more streamlined; the I/O futures allow the programmer to express the client request handling logic as sequences of synchronous I/O and, based on the I/O result, some amount of computation. Furthermore, we no longer need to explicitly assign client connections to a given worker thread. Instead, each client connection translates into a future routine that handles the requests for that connection.

This simplification is possible because the runtime scheduler understands the semantics of I/O futures. It transparently time-multiplexes among multiple concurrent connections, by switching away from one execution context when a client request is blocked on an I/O operation, and resuming it when the I/O completes.

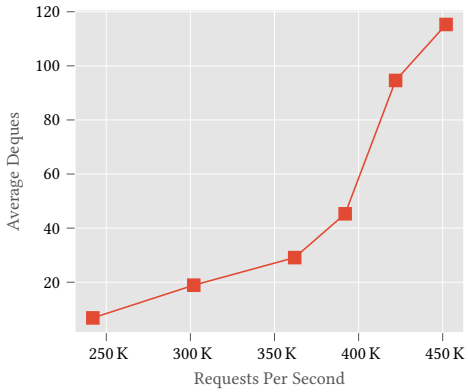


Figure 2: The average number of high priority deques at quantum boundaries in memcached using Adaptive I-Cilk. Counts are drawn from the best performing configurations, as in Figure 1.

Shortcomings of Adaptive I-Cilk

Task parallelism makes it easier to write interactive applications, but puts the burden on the scheduler to provide good performance by time-multiplexing between requests and providing low latency to high priority requests. As mentioned in Section 2, the Adaptive I-Cilk scheduler tries to satisfy these goals by providing good scheduling mechanisms for I/Os, futures, and by adding support for priorities. However, certain characteristics of interactive applications make this design inefficient.

Most importantly, the parallelism (one might call it concurrency in this context) in the Memcached server is purely due to the handling of multiple concurrent client connections. There isn't much parallelism otherwise — requests for a given connection are handled sequentially and there are not any parallel operations needed within a single request. [4] Therefore, the scheduler must juggle many execution contexts (futures) concurrently in order to get efficiency. In addition, because all these connections are active at once, the aging heuristic is absolutely essential.

As mentioned in Section 2, both Adaptive I-Cilk and Prompt I-Cilk have a large number of deques at each priority level. Figure 2 shows the average number of non-empty deques across quantum when running the Memcached server with different server loads on Adaptive I-Cilk. As the numbers show, even for a lower server load, the number of non-empty deques at a given time is on the order of hundreds. The overall number of deques (including suspended and empty ones) would be higher.

Recall that Adaptive I-Cilk is a randomized scheduler which tries to ensure that all deques are stolen from with about equal probability. The randomization in a classic work stealing scheduler serves two purposes. First, it avoids contention. When every thief randomly chooses a victim to steal from, one can show that probability of stealing from the same victim is low. Second, it allows the scheduler to amortize failed steal attempts against the ***span*** of the computation, the length of a longest dependence chain in the computation. This scheduler design makes sense for classic task-parallel applications, where parallelism in the applications is mostly expressed in the form of fork-join parallelism and the number of deques is approximately the same as the number of workers.

In the context of interactive applications, where there a lot of futures and I/O operations, and therefore deques, this design may not be ideal and can incur higher scheduling overhead than necessary. First, because the number of non-empty deques is abundant, contention is less of an issue. Second, there is little need to amortize the steal cost assuming a thief looking for work can locate a non-empty deque quickly without spending much time on failed steals, which is much easier to accomplish when the number of deques is large. For these two reasons, the randomization, at least for the purpose of practical considerations, no longer seems as important.

Yet, the design of Adaptive I-Cilk incurs scheduling overhead in favor of ensuring randomization due to (1) rebalancing deques across processors; and (2) high constant overhead when a deque becomes empty or resumable because the deque pool of each processor is protected by a lock. In particular, the second point is subtle — in order to provide randomization, the deque pool must support random access and removal from arbitrary locations, which is difficult to implement without a lock.[5] However, due to locking, accessing the deque pool can become expensive because a deque in its life time can repeatedly transition between being suspended / empty and resumable / non-empty (due to I/O operations), and each deque can thus be repeatedly removed from and re-inserted into different deque pools. In addition to the scheduling overhead, the randomization makes it quite challenging to implement the aging heuristic, which is absolutely crucial to get low latency.

When the number of deques in the system is large, which is typical for interactive applications that utilize frequent I/O operations, we argue that it is more important to provide a lightweight data structure to manage the many deques with fast insertion and removal than to ensure that the work stealing is randomized. As we will see in Section 4, our Prompt I-Cilk scheduler design moves away from randomized work stealing while keeping some other aspect of work stealing from [42], and our design favors access efficiency for managing the many deques. It also makes it easier to implement the aging heuristic. In Section 5, we show that the empirical data implies that the Prompt I-Cilk design incurs lower scheduling overhead than Adaptive I-Cilk.

4 THE PROMPT I-CILK SCHEDULER

This section describes the construction of the Prompt I-Cilk scheduler. At any time, each worker is working on a particular priority level. We use a bitfield to keep track of which priority levels have available work and each worker, asynchronously and somewhat frequently, checks whether it should switch to a different priority level. Within a priority level, we use a simple queue based scheduler

[4] In principle, when the hash table gets resized, one can parallelize the moving of buckets from the old hash table to the new resized hash table, but we didn't do that (nor did the pthreaded implementation) as a) the resizing occurs concurrently while the requests are being handled and does not seem to be a bottleneck and b) during our performance evaluation resizing doesn't get triggered because the initial capacity is configured to be large enough for the workload.

[5] We are not aware of any lock-free concurrent data structures that satisfy these requirements. For instance, queue data structures (e.g., [28, 45]) do not allow for random accesses, nor does it allow one to remove items from arbitrary locations. On the other hand, a concurrent vector may support random access, but it cannot support removal from arbitrary locations in constant time (e.g., [10, 14]).

which can be viewed as a hybrid of work sharing and work stealing, with the goal of optimizing for efficient handling of many deques.

Work Stealing without Randomization

Within a priority level, our strategy can be viewed as a hybrid of work stealing and work sharing. The Prompt I-Cilk scheduler, like Adaptive I-Cilk, retains certain properties of proactive work stealing [42] (summarized in Section 2). Specifically, each worker operates on its own active deque (at the priority level that the worker is in); when the worker runs out of work or encounters a `get` that requires suspending its deque, it tries to steal work from other deques at the same priority level. Due to deque suspensions, there can be many deques at each priority level at any given time. Prompt I-Cilk mainly differs from Adaptive I-Cilk in how workers perform steals and how it manages the many deques in the system.

Prompt I-Cilk uses a relatively simple strategy for finding work. Instead of each processor having its own deque pool and performing randomized work-stealing, Prompt I-Cilk maintains a single deque pool for each priority level — this deque pool is implemented using an efficient concurrent non-blocking FIFO queue.[6] The queue utilizes fetch-and-add to implement fast insert (at the tail) and removal (from the head). It is organized as an array of arrays to allow for concurrent accesses while resizing. It uses the standard epoch-based reclamation technique [15] to ensure that no workers are still referencing the old arrays before recycling them.

Whenever a worker (thief) wants to steal, it pops the deque off the head of the queue. If the deque is suspended and stealable, the thief steals from the top of that deque. If the deque is resumable, the thief simply ***mugs*** the whole deque and resumes the bottom-most frame. If the deque, after being stolen from / resumed, still contains stealable work, the deque gets pushed back onto the tail of the queue. Similarly, if a deque becomes resumable and if it is not already in the queue, the system pushes it to the tail of the queue.

Another key distinction between Prompt I-Cilk and Adaptive I-Cilk is that, while Adaptive I-Cilk ensures that no non-stealable (e.g., suspended and empty) deques are in any deque pool, Prompt I-Cilk does not maintain this strict condition. If a thief discovers an empty deque at the head of the queue, it simply doesn't push it back and removes the next deque from the queue. Prompt I-Cilk maintains the invariant that all non-empty deques (active, suspended or resumable) are in the deque pool; however, some empty deques may also be in the deque pool because a deque may become empty while in the queue and remain there for some time. For instance, worker p's active deque may be non-empty and therefore in the queue, and p may pop off the bottom frame, making it empty. This deque remains in the queue until it reaches the head of the queue, some thief pops it, discovers that it is empty and refrains from adding it back. While this mechanism means that sometimes workers have to do multiple queue accesses in order to find work, it makes the queue design simple and fast since we needn't support removal or insert except at the head or tail respectively.

It is crucial, however, that all non-empty deques are in the queue since any available work on a deque must be discoverable by other workers. One possible design choice might be to keep all the deques in the queue — however, this would make the queue cluttered with many empty deques making it difficult for thieves to find work. Prompt I-Cilk strikes a balance — when a thief discovers an empty deque, it does not add it back to the queue. This, however, means that when an empty deque becomes non-empty, it might need to be added back. Therefore, whenever the system resumes a deque, it checks to see if this deque is already on the queue and pushes it back if it is not. Similarly, when a worker pushes something onto its active deque (via `spawn` or `fut-create`), it checks and pushes its active deque back onto the queue if necessary.

This checking of whether a deque needs to be pushed back onto the queue goes against the work-first principle [17]; however, in our experience, we find that this choice strikes the right balance. The added overhead is small enough that this choice is superior to both alternatives, namely (1) keeping all deques in the queue (making it difficult to find work); and (2) maintaining the strict invariant (as Adaptive I-Cilk does) that no empty deque is in the queue (requiring complex data structures that support removal of deques from arbitrary locations in the queue).

Support for Aging

Recall that the ***aging*** heuristic — handling requests in roughly old to new order — is crucial for performance in interactive applications. Intuitively, a FIFO queue naturally emulates the aging heuristic — when deques become resumable, they are inserted into the queue in FIFO order. However, it turns out that there are some subtle exceptions. Therefore, in practice, we actually utilize *two* queues to maintain the deque pool in order to implement aging properly.

One exception where the FIFO queue does not implement aging strictly is when a deque d is already in the queue, d becomes both suspended and empty, but d has yet to be removed from the queue (because it has not yet reached the head). Now suppose d becomes resumable again. In this case, d might be handled before deques that became resumable before it, but were added to the queue later. It turns out that this doesn't happen enough to create issues in practice as deques in interactive applications tend to be shallow (i.e., with zero or few stealable items upon suspension), and workers go through deques quickly due to frequent I/O operations. Thus, a few deques can get "lucky" and cut in line in this fashion.

The other exception is due to abandoned deques and is more problematic. This happens when a worker p is working on a lower priority, say k; some higher priority work becomes available; p abandons its active deque at priority k and starts work at a higher priority.[7] We call such a deque an ***immediately resumable*** deque — it was suspended not because it was blocked on a `get`; it was suspended because the worker has to abandon it to go work on a higher priority. At the point of suspension, this immediately resumable deque must be inserted into the queue, if it is not in the queue already. By inserting it at the tail, however, we are effectively "de-aging" the deque, putting it behind any deques that became resumable before the insertion.

To mitigate this issue, we use two queues to implement the centralized pool — a ***mugging queue*** that contains *only* immediately resumable deques that were abandoned, and a ***regular queue*** that operates as described above. If a worker needs to abandon its active

[6] The queue is organized as an array of arrays to allow for concurrent accesses while resizing.

[7] This is an issue only at lower priorities since a worker in the highest priority level will never abandon its active deque to go work on a different level.

deque, it *always* inserts the deque into the mugging queue. A thief trying to find work will always check the mugging queue first and mug the deque found in there before trying to steal or mug deques from the regular queue. Therefore, an immediately resumable deque may be in both queues until it is mugged by a thief, but this is not a problem; the regular queue operates as usual.

Supporting Promptness and Elasticization

The Prompt I-Cilk scheduler tries to approximate prompt scheduling — workers want to work on high priority work if any is available. In order to do so, the Prompt I-Cilk scheduler uses a bitfield to indicate which priority levels currently have work, where index i in the bitfield is set to 1 if priority level i has work. In practice, we use a 64-bit integer to represent the bitfield since it allows us to use the efficient atomic logical operations on x64 to manage the bitfield. In particular, we use fetch-and-or and fetch-and-and to update the bitfields; C++ atomic load with sequential consistent memory ordering to read the bitfield; and the compiler builtin `__builtin_clzll` (e.g., see [4]) to quickly retrieve the bit index for the highest bit set. Although this limits the max number of priority levels to 64 in our implementation, this is more than enough in the applications we examined.

Because of the centralized deque pools, it is easy in Prompt I-Cilk to check whether a given priority level contains work. A worker, when enqueueing a deque into a pool, always sets the corresponding bit. A thief, when encountering an empty pool (which requires checking two queues, as described above), clears the corresponding bit. Since checking the size of the pool and clearing the bit are not atomic, the thief performs double checking — if the pool is empty, it clears the bit, checks the pool again, and resets the bit if the pool is no longer empty, ensuring that the bit should not be left unset for an extensive period if a thief clearing the bit interleaves with an active worker generating new work.

Every worker checks this bitfield frequently — a thief checks this bitfield before attempting a steal; an active worker checks this bitfield at every `spawn`, `sync`, `fut-create`, and `get`. If a worker realizes that it is working at a lower priority level than the highest level with available work, it abandons its active deque in the current level and moves itself to the higher level (the abandoned work remains in the lower priority deque pool and will eventually be resumed). The frequent checking goes against the work-first principle [17]. However, somewhat surprisingly, we found that implementing the promptness with the frequent checking works well — in Section 5 we show that the frequent checking allows the scheduler to react to work quickly and does not incur undue scheduling overhead.

At times, thieves may find that there are no bits set in the bitfield. In this case, the thief is currently (perhaps temporarily) not needed by the application. Workers in this situation will, instead of stealing, wait on a global condition variable while the bitfield contains only 0 bits. As soon as an active worker sets the bitfield from zero to non-zero, that worker will broadcast the condition variable to wake up all sleeping workers. These workers then re-enter the scheduler loop and resume working on the highest priority work.

5 EVALUATION OF PROMPT I-CILK

This section empirically evaluates Prompt I-Cilk using three applications, including the Memcached server [2] discussed in Section 3. We evaluate Prompt I-Cilk by comparing it to Adaptive I-Cilk, the-state-of-the-art priority-oriented task-parallel platform. We additionally compare Prompt I-Cilk to two other variants of Adaptive I-Cilk, to evaluate how much of the performance of Prompt I-Cilk may be attributed to each of its key differences from Adaptive I-Cilk. Empirical results indicate that generally Prompt I-Cilk outperforms Adaptive I-Cilk and incurs lower waste.

Experimental Setup and Evaluation Criteria

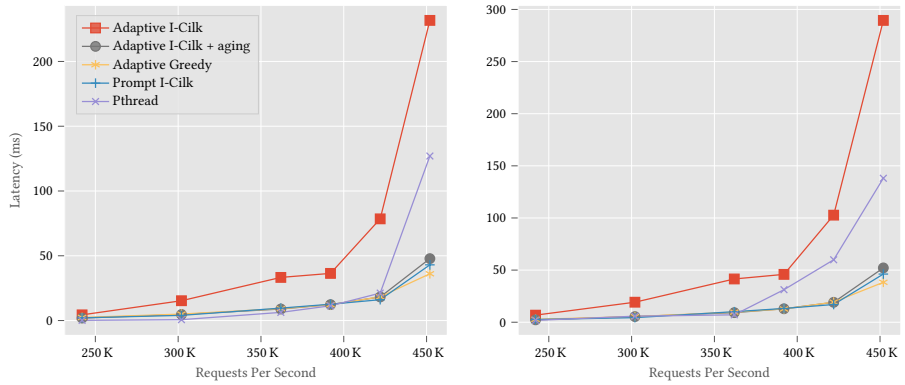
Hardware configuration. We ran experiments on a system with 40 2.40-GHz cores split across 2 Intel Xeon Gold 6148 processors, and 768-GB of main memory. Each processor has a shared (among 20 cores) 27.5-MB L3 cache. Hyperthreading was enabled, and each core had 2 hardware threads. Each core has a 32-KB L1 data cache, a 32-KB L1 instruction cache, and a 1-MB L2 cache. Experiments ran in Linux kernel 5.15, and all benchmarks were compiled with -flto and -O3 using the clang 5.0.0-based Tapir compiler [38].

The Memcached server setup. We have already covered the characteristics of Memcached [2] extensively in Section 3. One of the most important considerations for Memcached is how many requests per second it can process while meeting guarantees on the tail latency, which we will refer to as the ***Quality of Service*** (***QoS***). For Memcached, we adopt the QoS criterion used by Palit et al. [36]: 95% of all client requests should be handled within 10 ms. We use the Memcached driver from [36] to find the maximum number of requests per second (RPS) that the server can handle using the pthreaded version, and run experiments on nearby RPS values. As in [36], we find the maximum RPS that meets the QoS using a binary search on the RPS with a fixed client count (600).

We ran Memcached on 4 cores and used 20 cores for client connections. For the pthreaded version, this configuration means the server creates 4 worker threads, plus the main thread and a few background threads. Even though the pthreaded version oversubscribes, the background threads rarely ran. For all other platforms, this configuration means that the platform creates 4 worker threads plus 4 I/O handling threads (which is based on the design of the prior work on handling I/O futures [40]). We chose to run the Memcached server on this configuration because the Memcached documentation [3] recommends one run the server with only 4 worker threads, as more worker threads would likely create a synchronization bottleneck on the shared cache.

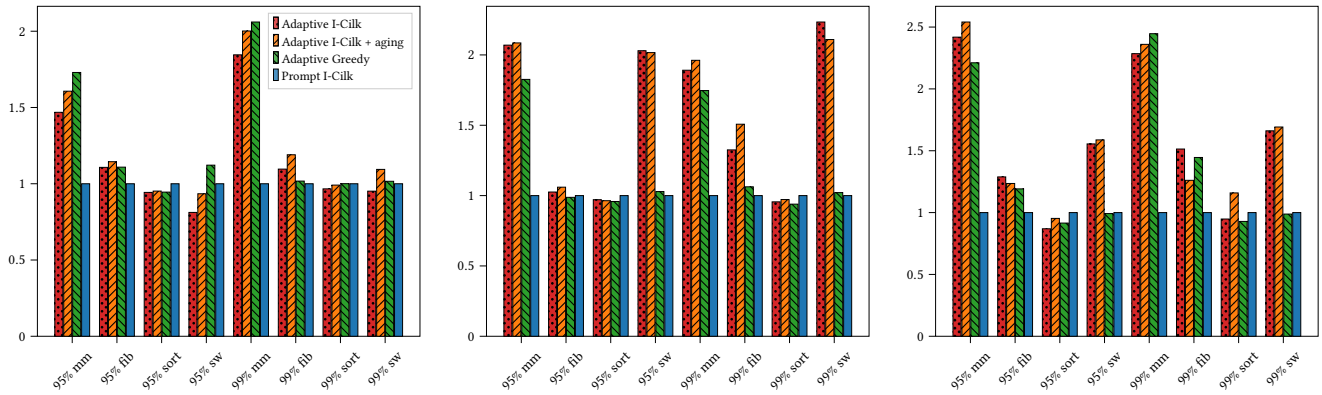
The email and job servers setup. The email and job servers were used to evaluate Adaptive I-Cilk [41], and they are specifically designed to test out priority-oriented task-parallel platforms, where each application consists of multiple tasks with multiple priority levels. We acquired the applications from the public release of Adaptive I-Cilk [39], but modified them slightly to ensure that the amount of the work done in each run is the same. Typically, if in an interactive application the input arrives with nondeterministic timing, then the workload generated may differ depending on the internal server state when the input arrives. To enable more consistent timing and a fair comparison between different platforms, we made changes to the client request generations and how the servers worked to ensure consistent workload generation.

The email server benchmark simulates a multi-user email server and supports multiple operations, from highest to lowest priorities



95th Percentile Latency 99th Percentile Latency

Figure 3: The latencies of Memcached implemented with different schedulers. For Adaptive I-Cilk, Adaptive I-Cilk plus aging, and Adaptive Greedy the data points are drawn from the runtime parameter configuration with the best 99th percentile latency for the given RPS. Adaptive I-Cilk uses 5 different sets of parameters, while Adaptive I-Cilk plus aging and Adaptive Greedy both use 4 different sets of parameters.



3 RPS 4 RPS 5 RPS

Figure 4: The 95th and 99th percentile latencies of the job benchmark normalized to the latencies of Prompt I-Cilk, with Prompt I-Cilk also shown for reference. Tasks are shown in order of highest to lowest priority. Adaptive I-Cilk uses 3 different sets of parameters, and its variants use 2 different sets of parameters.

— a) send emails (`send`), b) sort emails (`sort`), and c) two operations with equal priority: compress emails (`compress`); and print emails (`print`), which requires decompressing the emails. We ran the email server on 4 cores and used 20 cores to simulate client connections.

The job server simulates a server that performs shortest-job-first scheduling, so shorter jobs get higher priorities, from highest to lowest: a) matrix multiplication (`mm`), b) Fibonacci (`fib`), c) sort (`sort`), and d) Smith-Waterman (`sw`). Because all these jobs are parallel tasks, we ran the job server on 20 cores.

Variants of Adaptive I-Cilk. We additionally implemented two variants of Adaptive I-Cilk to gauge how much of the performance of Prompt I-Cilk may be contributed by its key differences from Adaptive I-Cilk. The first one is Adaptive I-Cilk plus aging, which adds a per-worker aging heuristic to Adaptive I-Cilk; that is, each worker additionally maintains a queue of resumable deques in resumption order, such that thieves stealing from this worker can work on these deques according to the aging order. This allows for a per-worker approximation of the aging heuristic. The second one is Adaptive Greedy, which still uses the two-level scheduling system as in Adaptive I-Cilk but replaces the bottom-level scheduler with the work-stealing scheduler in Prompt I-Cilk, i.e., it uses a centralized deque pool and steals without randomization, and therefore approximates aging better than Adaptive I-Cilk plus aging.

Like Adaptive I-Cilk, both variants still require setting runtime parameters. For all of these Adaptive variants of schedulers, we always perform the runtime parameter sweep for each benchmark and present numbers with the runtime parameters that have the best tail latency.[8] This means the timing data shown for each RPS for a given benchmark may not use the same runtime parameters.

Latency Comparison

Figure 3 shows the 95th and 99th percentile latencies of handling requests in the Memcached server for the implementations using pthreads, Prompt I-Cilk, and variants of Adaptive I-Cilk. We only show the 95th and 99th percentile latencies because the plots for the average and 50th percentile latencies look very similar to the 95th percentile plot. Prompt I-Cilk, Adaptive I-Cilk plus aging, and Adaptive Greedy have similar performance to the pthreaded version, and outperform the pthreaded version at higher RPS, especially for the 99th percentile latency. Adaptive I-Cilk, on the other hand, has

[8] For email and job servers, we used an average of 95th and 99th percentile latencies; for Memcached, we used the 99th percentile latency.

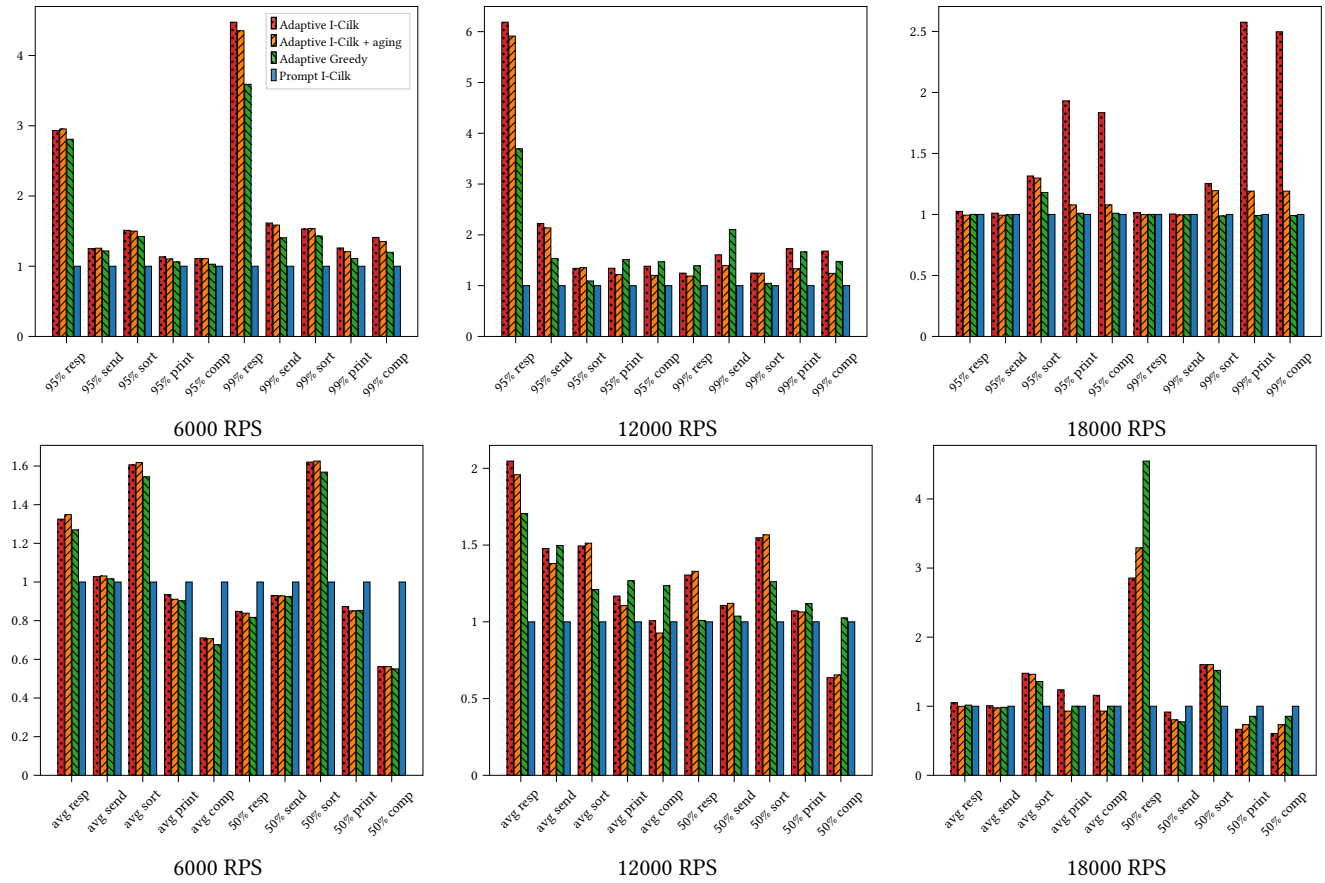


Figure 5: The top row shows the the 95th and 99th percentile latencies of the email benchmark normalized to the latencies of Prompt I-Cilk, with Prompt I-Cilk also shown for reference. The bottom row shows the average and median latencies similarly. Tasks are shown in order of highest to lowest priority. Print and comp are at the same priority level. The Adaptive I-Cilk variants use 3 different sets of parameters.

much higher latencies, creating an obvious gap between Adaptive I-Cilk and Prompt I-Cilk. This result indicates that the aging heuristic is really the crucial difference here. Interestingly, Adaptive Greedy slightly outperforms Prompt I-Cilk at the highest RPS, indicating that the promptness incurs slightly more overhead for Memcached at high RPS.

Figure 4 shows the 95th and 99th percentile latencies of the job server implemented in Prompt I-Cilk and the Adaptive variants of I-Cilk, with the latencies normalized to the latencies of Prompt I-Cilk. The server workload is configured to be low at 3 RPS, decent at 4 RPS, and high at 5 RPS. We only show the 95th and 99th percentile latencies, but the plots for the average and 50th percentile latencies look similar, and Prompt I-Cilk across the board outperforms other variants. The job server contains more parallelism — each job instance created by the server is a traditional task-parallel job. Even in this setting the scheduling mechanisms in Prompt I-Cilk seem to work well, namely aging and promptness.

First, let's examine the aging mechanism. The aging mechanism really kicks in when the number of deques within a priority level gets large. In the job server, unlike in Memcached, each request generates a parallel task and thus can fully utilize more cores. Consequently, the higher priority tasks (i.e., `mm` and `fib`) will take up more cores, leaving the lower-priority-level tasks unattended to for longer periods. We observed that the average number of deques (across quanta) is high for lower priority tasks (i.e., `sort` and `sw`), especially at 4 and 5 RPS (on the order of hundreds per level). The aging mechanism really matters for providing better latencies in this setting, as demonstrated by the superior performance of Adaptive Greedy over other Adaptive variants for `sort` and especially `sw` at 4 and 5 RPS. We don't see this phenomenon at 3 RPS because the number of deques is low across priority levels.

Second, the promptness in Prompt I-Cilk really helps. Generally, we see that the latency improvement of Prompt I-Cilk over other variants is the most pronounced for high server loads and for the higher priority levels, which makes intuitive sense for this workload. Because the job server contains many parallel jobs, the promptness of Prompt I-Cilk will cause it to work on the highest priority work available, whereas in other Adaptive variants the highest priority level will take time to both ramp up its worker counts when the tasks come online and then take time to ramp back down when the tasks complete. This slower ramp up allows the lower priority level

		Wasted (seconds)		*Running (seconds)*	
Benchmark	RPS	Adaptive I-Cilk	Prompt I-Cilk	Adaptive I-Cilk	Prompt I-Cilk
memcached	242K	2.8 (0.4x)	7.7	966.0 (1.0x)	966.6
	362K	1.3 (0.3x)	3.7	966.3 (1.0x)	966.1
	452K	0.5 (0.2x)	2.2	966.3 (1.0x)	966.2
job	3	125.5 (9.5x)	13.2	3787.0 (1.0x)	3805.9
	4	72.5 (49.7x)	1.5	5755.9 (1.0x)	5852.7
	5	65.2 (43.2x)	1.5	7097.9 (1.0x)	7285.6
email	6K	128.8 (4.8x)	26.8	152.2 (0.9x)	170.4
	12K	116.6 (5.0x)	23.4	318.3 (0.9x)	350.5
	18K	140.8 (12.3x)	11.5	512.6 (0.9x)	554.6

Figure 6: Wasted and running time, in seconds, accumulated across all processors. Running time is the processing time incurred by scheduling overhead and user code. The values in parentheses for Adaptive I-Cilk are the ratio of Adaptive I-Cilk to Prompt I-Cilk.

jobs to get some work done in the mean time, but the slower ramp down also means more wasted cycles, which we will examine in more detail later in this section.

The top row of Figure 5 shows the 95th and the 99th percentile latencies of the email server implemented with Prompt I-Cilk and the Adaptive variants of I-Cilk. We also show the the average and median latencies in the bottom row because, unlike other benchmarks, the shape of the average and median latencies do not resemble that of the higher percentiles.

The email server is an interesting application to study, as the data does not tell a story as clean as the job server. Let's examine the aging heuristics first. In the email server, we observed that the lower priority levels (i.e., `print` and `compress`) contain large numbers of deques at only the highest load, 18K RPS. Thus, we see that the aging heuristics really help in that setting. At the 6K RPS, because the number of deques is low, the aging heuristics do not do much, and the different variants perform comparably. However, at 12K RPS, we see Adaptive Greedy perform better for high priority tasks but worse at lower priority tasks. It turns out that, at 12K RPS in particular, the data points we chose to show for Adaptive Greedy used a runtime parameter that ramps up and down the core counts more aggressively compared to other variants, leading to better latencies at higher priority tasks but also higher waste, which resulted in lower priority tasks getting delayed. At 6K RPS we don't see this pattern because the best data points across variants happen to use the same runtime parameters.

At the 95th and 99th percentile, the promptness clearly helps as Prompt I-Cilk outperforms the other variants. For the 50th percentile, the Adaptive variants outperform Prompt I-Cilk at 6K RPS as well as for the lowest priority task, `comp`, at 12K RPS. Prompt I-Cilk still provided better or comparable average latencies for these data points, however, which indicates that Prompt I-Cilk provides more stable running times (i.e., lower variance).

Waste and Scheduling Overhead

To evaluate how well Prompt I-Cilk manages waste and its scheduling overhead, we additionally collected the waste times and running times for Adaptive I-Cilk and Prompt I-Cilk. The ***waste*** corresponds to time workers spent looking for and failing to find work. For Prompt I-Cilk, it also includes time a worker spent putting itself to sleep and waking up when the bitfield goes from non-zero to zero and vice versa. The ***running time*** corresponds to time workers spent doing useful work or scheduling overhead, such as successful steals, mugging, and in the case of Prompt I-Cilk, time spent checking the bitfield or checking if it needs to push its deque onto the queue while active.

Across benchmarks, Prompt I-Cilk incurs slightly higher running times, but makes up for it by reducing waste.

For the Memcached server and the job server, where the concurrency and parallelism are abundant, Prompt I-Cilk does not particularly incur higher scheduling overhead compared to Adaptive I-Cilk, which suggests that the extra scheduling overhead incurred in Prompt I-Cilk while a worker is active are not overly cumbersome. Moreover, compared to Adaptive I-Cilk, for the job server it seems to be managing waste a lot better.

The email server benchmark, on the other hand, creates sequential tasks and tasks with low parallelism in bursts. Such a workload causes Prompt I-Cilk to incur slightly higher waste compared to the job server, and slightly higher running time compared to both of the other benchmarks. The savings on waste still outweighs the higher running times compared to Adaptive I-Cilk however. Interestingly, Prompt I-Cilk incurs its highest waste at 6K RPS for email. As the server load gets higher (in this case, higher concurrent requests), it is able to manage waste better.

6 RELATED WORK

In Sections 1 and 2, we have discussed the most closely related work, namely Adaptive I-Cilk, in detail. In addition, there is extensive work, going back many decades, on work-stealing for classical throughput oriented parallel applications; here we will focus primarily on some related work for interactive applications.

As mentioned in Section 1, task parallelism for interactive applications is a relatively recent research topic, but has been considered from the perspective of type systems and cost models [29–32]. Most of this work focuses on designing type systems that can detect priority inversions in order to ensure that, if scheduled using a prompt scheduler, the application gets appropriate performance guarantees. This work is synergistic and orthogonal with our work because Prompt I-Cilk is meant to be a practically efficient approximation of a prompt scheduler. Some of this work [29, 30, 32] implements a work-stealing scheduler with private deques in an extension of parallel ML [43, 44]; however, this scheduler is mainly designed to be proof of concept for expressiveness of the type system and not really evaluated for performance on larger applications. Singer et al. (authors of Adaptive I-Cilk) [41] proved theoretical performance guarantees which do not apply directly to Adaptive I-Cilk; however, the authors argue that a design similar to Adaptive I-Cilk, but with a greedy scheduler at the bottom-level is *almost prompt*.

Interactive applications are conventionally written with asynchronous I/Os with event-driven programming. Runtime mechanisms have been proposed to allow for a synchronous I/O interface while providing the performance of asynchronous I/O [5, 21]; this work is concerned only with I/O performance and not load balancing. Researchers [18, 47] have also tried to incorporate work stealing into event-driven applications to allow for parallel execution but they are not concerned with the priorities of tasks.

Much work on interactive applications is in the context of web services, focusing on resource management to ensure quality of service (QoS) guarantees on responsiveness [9, 20, 23]. In web

searches, a single search can generate multiple queries, where the response time is dictated by the tail latency contributed by a few long queries. In this context, each query is treated as a parallelizable job, and multiple scheduling mechanisms have been proposed to selectively or incrementally parallelize certain queries to reduce the tail latency [20, 24, 25]. Often, interactive web services run in a multiprogrammed environment, where batch jobs are co-located with latency-sensitive web services to increase CPU utilization. Various scheduling techniques have been proposed for such systems to meet QoS while keeping the CPU utilization high [7, 11–13, 16, 26, 27, 33–35, 37, 46]). This line of work is largely orthogonal and complementary to ours — one can imagine using these systems to manage resources in a multiprogrammed environment with some latency-sensitive task-parallel applications written to run on Prompt I-Cilk.

7 CONCLUSION AND FUTURE DIRECTIONS

This paper shows that well-engineered task-parallel platforms can provide good performance for real-world interactive applications while simplifying their construction. It's interesting that Prompt I-Cilk outperforms the state-of-the-art priority-oriented task-parallel platform by going against the conventional wisdom on work-stealing, such as the importance of randomization and the work-first principle. We believe this is because the characteristics of parallel interactive applications differ considerably from traditional task-parallel applications, and thus different considerations apply.

There are several directions of future work. First, Prompt I-Cilk implements an approximation of prompt schedulers, but we haven't characterized its worst case performance theoretically; it would be interesting to do so. Second, real-world interactive applications are complex and use many features, e.g. locks and condition variables, which must be handled better if task-parallelism is to become the new way these applications are written. Finally, it would be interesting to test Prompt I-Cilk on additional real-world benchmarks.

8 ACKNOWLEDGMENTS

We thank Noah Goldstein and York Liu for their work with Adaptive I-Cilk that provided valuable feedback on implementation of the interface, as well as the work they started on porting benchmarks. We also thank Xiaojian Xu for her aid in writing scripts to plot the results of our benchmarks. The work in this paper was funded in part by the National Science Foundation under grants CCF-1910568, CCF-1943456, CCF-2107280, and CCF-2216971.

REFERENCES

[1] 2002. *libevent – an event notification library.* https://libevent.org/ Accessed in January 2023.
[2] 2009. *Memcached.* https://memcached.org/ Accessed in July 2019.
[3] 2019. *ConfiguringServer.* https://github.com/memcached/memcached/wiki/ConfiguringServer#threading Accessed in November 2020.
[4] Year not available. Other Built-in Functions Provided by GCC. https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html. Accessed in January 2023.
[5] Saulo Medeiros de Araujo, Kiev Santos da Gama, Nelson Souto Rosa, and Silvio Lemos Meira. 2014. Afluentes Concurrent I/O Made Easy with Lazy Evaluation. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 279–287.
[6] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* (2001), 115–144.
[7] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.* 34, 4, Article 11 (Dec. 2016), 39 pages.
[8] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *JACM* 46, 5 (1999), 720–748.
[9] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.
[10] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. 2006. Lock-Free Dynamically Resizable Arrays. In *Principles of Distributed Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 142–156.
[11] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, Houston, Texas, USA, 77–88.
[12] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, Salt Lake City, Utah, USA, 127–144.
[13] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. 2007. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*. IEEE Computer Society, Washington, DC, USA, 25–38.
[14] Steven Feldman, Carlos Valera-Leon, and Damian Dechev. 2016. An Efficient Wait-Free Vector. *IEEE Transactions on Parallel and Distributed Systems* 27, 3 (2016), 654–667.
[15] Keir Fraser. 2004. *Practical lock-freedom.* Ph. D. Dissertation. Computer Laboratory, University of Cambridge, Cambridge, Massachusetts. No. UCAM-CL-TR-579.
[16] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. *Caladan: Mitigating Interference at Microsecond Timescales.* USENIX Association, USA.
[17] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. ACM, 212–223.
[18] Fabien Gaud, Sylvain Genevès, Renaud Lachaize, Baptiste Lepers, Fabien Mottet, Gilles Muller, and Vivien Quéma. 2010. Efficient Workstealing for Multicore Event-Driven Systems. In *2010 IEEE 30th International Conference on Distributed Computing Systems*. 516–525.
[19] Robert H. Halstead, Jr. 1985. Multilisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS* 7, 4 (Oct. 1985), 501–538.
[20] Md E Haque, hun Yong Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S McKinley. 2015. 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)*. 161–175.
[21] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. 2011. AC: Composable Asynchronous IO for Native Languages. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 903–920.
[22] Institute of Electrical and Electronic Engineers. 1996. Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]. IEEE Standard 1003.1, 1996 Edition.
[23] Myeongjae Jeon, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. 2013. Adaptive Parallelism for Web Search. In *ACM European Conference on Computer Systems (EuroSys)*. 155–168.
[24] Myeongjae Jeon, Saehoon Kim, Seung-Won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. 2014. Predictive parallelization: taming tail latencies in web search. In *ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 253–262.
[25] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. 2015. Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search. In *WSDM*. 7–16.
[26] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, Portland, Oregon, 450–462.
[27] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, Porto Alegre, Brazil, 248–259.
[28] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. ACM, 267–276.
[29] Stefan K. Muller, Umut A. Acar, and Robert Harper. 2017. Responsive Parallel Computation: Bridging Competitive and Cooperative Threading. In *Proceedings*

of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017). ACM, Barcelona, Spain, 677–692.

[30] Stefan K. Muller, Umut A. Acar, and Robert Harper. 2018. Competitive Parallelism: Getting Your Priorities Right. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Functional Programming (ICFP '18)*. ACM, St. Louis, MO, USA, 95:1–95:30.

[31] Stefan K. Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. 2020. Responsive Parallelism with Futures and State. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. ACM, 557–591.

[32] Stefan K. Muller, Sam Westrick, and Umut A. Acar. 2019. Fairness in Responsive Parallelism. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming* (to Appear) *(ICFP '19)*. ACM, Berlin, Germany.

[33] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, Paris, France, 237–250.

[34] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, San Jose, CA, 219–230. http://dl.acm.org/citation.cfm?id=2535461.2535489

[35] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[36] T. Palit, Yongming Shen, and M. Ferdman. 2016. Demystifying cloud benchmarking. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 122–132.

[37] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341.

[38] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, Austin, Texas, USA, 249–265.

[39] Kyle Singer. 2020. https://github.com/wustl-pctg/I-Cilk. Accessed in January 2023.

[40] Kyle Singer, Kunal Agrawal, and I-Ting Angelina Lee. 2020. Scheduling I/O Latency-Hiding Futures in Task-Parallel Platforms. In *Proceedings of the Symposium on Algorithmic Principles of Computer Systems (APoCS)*. SIAM, 147–161.

[41] Kyle Singer, Noah Goldstein, Stefan K. Muller, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. 2020. Priority Scheduling for Interactive Applications. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 465–477.

[42] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. 2019. Proactive Work Stealing for Futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. ACM, New York, NY, USA, 257–271.

[43] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. 2008. Space Profiling for Parallel Functional Programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, Victoria, BC, Canada, 253–264.

[44] Daniel John Spoonhower. 2009. *Scheduling Deterministic Parallel Programs*. Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University.

[45] Chaoran Yang and John Mellor-Crummey. 2016. A Wait-Free Queue as Fast as Fetch-and-Add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, Barcelona, Spain.

[46] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, Tel-Aviv, Israel, 607–618.

[47] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert T. Morris, and David Mazières. 2003. Multiprocessor Support for Event-Driven Programs. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*. USENIX Association, San Antonio, TX. https://www.usenix.org/conference/2003-usenix-annual-technical-conference/multiprocessor-support-event-driven-programs