Priority Scheduling for Interactive Applications

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

Noah Goldstein
Washington University in St. Louis
St. Louis, Missouri
goldstein.n@wustl.edu

Stefan K. Muller
Carnegie Mellon University
Pittsburgh, Pennsylvania
smuller@cs.cmu.edu

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

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

Umut A. Acar
Carnegie Mellon University
Pittsburgh, Pennsylvania
umut@cs.cmu.edu

ABSTRACT

Many modern parallel applications, such as desktop software and cloud-based web services, are service-oriented, long running, and perform frequent interactions with the external world (e.g., responding to user input). We want such interactive applications to provide fast response times because typically at the other end of the external interaction there is a user waiting for a response. Existing parallel platforms designed for multicore hardware do not work well for such interactive applications, because they are designed to maximize throughput (rather than responsiveness). Interactive applications may have a mixture of interactive and compute-intensive tasks occurring concurrently, and the scheduler must be able to discern and prioritize tasks so that tasks which require faster response are prioritized over background tasks.

We present Interactive Cilk, or I-Cilk for short, a task parallel platform designed to schedule such parallel interactive applications. I-Cilk supports a C++-based templated library that allows the programmer to specify priorities for task-parallel code, and the underlying runtime schedules the computation so as to optimize for the response time of high-priority tasks. We show that the scheduling algorithm used by I-Cilk provides provably efficient response times for tasks at all levels of priorities, with better response time to high-priority tasks. We also empirically demonstrate that the scheduling algorithm can be implemented efficiently in practice with low scheduling overhead and provides fast response times for high-priority tasks.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**; *Parallel computing models*; • **Software and its engineering → Software performance**.

SPAA '20, July 15–17, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-6935-0/20/07…$15.00
https://doi.org/10.1145/3350755.3400236

KEYWORDS

scheduling, responsive parallelism, adaptive scheduling, work, span, greedy scheduling, performance bounds, interactive applications, priority inversion

ACM Reference Format:
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 '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3350755.3400236

1 INTRODUCTION

Many modern parallel applications have a mixture of different types of tasks occurring concurrently that may have different requirements for responsiveness: interactive tasks that interact with the external world (e.g., the user) and therefore must provide fast response; and other (possibly compute intensive) tasks that run in the background to support these interactive tasks, but may not need to be quite as responsive. We want such interactive applications to be able to effectively utilize commodity multicore hardware because multicore processors are widely deployed from personal computers to mobile devices to cloud platforms.

In order for the application to be responsive, the scheduler must be able to discern and prioritize tasks that require faster response time over tasks with looser or no responsiveness requirements. Consider for example a modern desktop application such as an email client application; it likely has a graphical user interface (GUI) component that interacts with the user by continuously listening to keyboard or mouse inputs and reacting to them. The user may type in a search string; the email client reacts by performing a compute-intensive search across all the emails in the inbox. From time to time, the email client may trigger background tasks such as compressing existing emails as an archive to save storage space. In this example, the GUI component generates high-priority tasks, since they need to be most responsive. The search constitutes a medium priority task, as it needs to be done to respond to the user but is not as latency-sensitive as the GUI component. Finally, the compression is a low-priority task, as it does not directly interact with the user.

Most scheduling algorithms used by existing task-parallel platforms for multicore hardware do not work well for such interactive applications, because they are designed for throughput-oriented

applications, as opposed to applications with latency-sensitive components that must be responsive. Thus, the scheduler has no notion of priorities and treats all ready tasks as equal — in the above example, if the archiving is ongoing and occupying the entire machine, a GUI task may not get to execute at all. If these schedulers are used for interactive applications, the interactive latency-sensitive components may not execute as promptly as we would wish.

Researchers have begun to address how to best support such interactive parallel applications in task-parallel platforms. Muller et al. [17, 18, 20] describe language extensions to parallel ML [25, 26] to express priorities of tasks. The languages are accompanied by a cost semantics that allows one to bound the response time of high-priority tasks, provided that the computation is ***well-formed*** and does not contain any ***priority inversions***, where a high-priority task may wait for a low-priority task to complete to make progress. For well-formed computations, Muller et al. show that a "prompt" scheduler can provide provably efficient response times for high-priority tasks and bounded execution times for low-priority tasks. The accompanying type system checks for the well-formed-ness of programs; if a program type checks, the resulting computation is guaranteed to be well-formed and the corresponding cost semantics hold.

A shortcoming of the prior work by Muller et al. [17, 18, 20] is that the performance guarantees provided by its cost semantics hold only if a scheduler adheres to the ***prompt scheduling principle*** which requires that the scheduler is both ***greedy or work-conserving*** — no core is idle as long as some work is available to be done — and ***prompt*** — strict prioritization such that no core does low-priority work if high-priority work is available.

No prior work has developed a scheduler that strictly maintains, or even provably approximates, prompt scheduling in all cases, however, and doing so is challenging in practice. For example, strict adherence to the protocol would require extremely frequent preemption (so that high-priority work can immediately preempt lower-priority work) and a global queue (so that priority determinations can be made over all the work in the system). The context switching and contention in such an approach would result in unacceptable overhead. Thus, a practical system must approximate a prompt schedule using coarser-grained preemption and decentralized queues, much like approaches such as randomized work stealing [7–10] approximate work-conserving schedules [11, 13]. The most recent work by Muller et al. [20] describes an implementation based on distributed load balancing that aims to approximate prompt scheduling in common cases, but it does not formally analyze the algorithm. Indeed, in certain cases, their implementation can be shown not to match the theoretical bounds provided by the cost semantics.

In this paper, we present ***Adaptive Priority Scheduling*** (***APS*** for short), a scheduling algorithm that approximates the prompt scheduling principle for scheduling computations with priorities. APS takes inspiration from ***A-GREEDY*** [3], an online two-level scheduling algorithm originally designed for scheduling multiple independent parallel jobs on a single multicore. Like in A-GREEDY, APS assumes that time is broken into a sequence of ***scheduling quanta*** consisting of L time steps and utilizes an adaptive scheduling strategy with two-level scheduling. The top level is a processor allocator that determines how to best assign cores to each priority level for the next quantum. For each priority-level, a ***work-conserving*** scheduler [11, 13] maps available work within the priority level onto the assigned cores during the quantum. APS can be implemented efficiently because it does not follow the prompt scheduling principle strictly and the resource allocator changes the allocation of cores between different priority levels at the granularity of the scheduling quantum of length L and not at every timestep. Quantum length L can be large enough to amortize the scheduling and preemption overheads while approximating prompt scheduling.

Even though APS does not follow the prompt scheduling principle strictly, we show that APS provides provably efficient response time for tasks at all priority levels. To formally state the bounds, we use a cost model similar to the one based on work and span from prior work [7–10]. We model the computation as a ***directed acyclic graph*** (or ***DAG***) where a node denotes a unit-time instruction[1] and an edge denotes the dependence between a pair of nodes. Like most prior work, we assume that each node has out-degree at most two.

Since interactive applications can be long running, we are concerned with bounding the execution time of each task as opposed to the overall execution time, where a ***task*** τ is defined as a sub-DAG with a single source ($source(\tau)$) and a single sink ($sink(\tau)$) and consists of nodes with the same priority level. Given a task τ at priority level ℓ, we define the ***competitive work*** W_τ of τ as the number of nodes in τ and the nodes logically parallel with τ that have the same or higher priority as τ. We say that an ***induced sub-DAG*** for a given task τ is the sub-DAG consisting of τ and all nodes that have a directed path to $sink(\tau)$ and that are not proper ancestors of $source(\tau)$. Then, the ***span*** S_τ of τ is defined as the length of a longest path in the induced sub-DAG of τ. Given the competitive work W_τ and span S_τ of task τ with priority level ℓ, APS with scheduling quantum length L executes τ in time $T(\tau) = O(\frac{W_\tau}{P} + S_\tau + (\ell + k)L \lg P)$, where k is the number of edges between two nodes with different priority levels in the induced sub-DAG of τ and P is the total number of cores in the system. Here, the highest priority level has $\ell = 1$, and lower priorities take on a higher value for ℓ. Thus, a highest-priority task τ has a response time of $T(\tau) = O(\frac{W_\tau}{P} + S_\tau + L \lg P)$, since $\ell = 1$ and $k = 0$ (in a well-formed DAG with no priority inversion, the induced sub DAG of a task with priority level ℓ has only nodes with priority level ℓ or smaller).

To put these bounds into perspective, work by Muller et al. bounds the execution time of a task τ at any priority level on a "prompt" system with P cores to be $O(W_\tau/P + S_\tau)$.[2] Therefore, theoretically, APS incurs an additive overhead of $O((k + \ell)L \lg P)$, which is typically small.

We implemented APS in a Cilk-based task-parallel platform [14] called ***Interactive Cilk*** (or ***I-Cilk*** for short). Even though APS is analyzed assuming a work-conserving scheduler for each priority level, a strict work-conserving scheduler is difficult to implement efficiently. Thus, I-Cilk instead implements a distributed, approximately work-conserving scheduler based on ***work-stealing*** [10] to schedule tasks within a priority level. To provide provable bounds,

[1] This assumption is without loss of generality since a larger node can be represented by a chain of unit-work nodes.
[2] Work W_τ and span S_τ are defined similarly.

APS necessarily assumes that the DAG is well-formed, which is checked by I-Cilk's C++-template based type system [19]. Finally, to support applications that perform I/O operations, I-Cilk provides a library that allows the scheduler to overlap computations with I/O.

With I-Cilk, we empirically demonstrate the following. First, I-Cilk can provide practically efficient response times for highest-priority tasks and bounded execution times for lower-priority tasks. Second, APS can be implemented efficiently as I-Cilk incurs negligible overhead for utilizing the two-level adaptive scheduling algorithm. Finally, when compared to a vanilla scheduler that does not account for priorities, I-Cilk provides better response times for highest-priority tasks and more stable running times for all tasks.

2 BACKGROUND

In this section we explain how interactive applications which contain tasks with different priorities are modeled and also explain the relationship of APS to A-GREEDY.

Modeling I-Cilk Computations. I-Cilk supports both fork-join parallelism (using `spawn`/`sync` keywords) and futures (using `create`/`get` keywords) to generate parallel tasks. In particular, when a function F invokes another function G with the keyword `spawn`, G may execute in parallel with the continuation of F; however, once F invokes `sync`, control cannot pass beyond it until all locally `spawn`ed functions have returned. On the other hand, if F creates a future by invoking a function H preceded by `create`, `create` returns a future handle which represents the execution of H. H may similarly execute in parallel with the continuation of F, but a `sync` does not act as a barrier for future tasks — one must explicitly invoke `get` on the future handle to wait for H to complete and retrieve its return value.

We now relate these language constructs to the computation DAG model discussed in Section 1. Upon executing a `spawn` or `create` the current node terminates with two outgoing edges: one to the spawned function and one to the continuation of the caller. Upon executing a `sync`, the current node terminates and a new node is generated to represent the continuation after the `sync`. This new node has multiple incoming edges: one from the node terminated with `sync` and one from each previously spawned function. Upon executing a `get`, the current node terminates and a new node is generated to represent the continuation after the `get`. This new node has two incoming edges: one from the last node of the joined future task, and one from the node terminated by `get`. With only `spawn` and `sync`, the execution generates a series-parallel (SP) DAG [28]. With the addition of futures, the DAG can be thought of as a set of SP DAGs connected with additional non-series-parallel edges that arise due to `create` and `get` [1, 27].

In a program written using I-Cilk, tasks have priorities. When one invokes `create`, the future function is always considered a separate task from its caller and has its own priority (potentially different from the caller). This may or may not be the case for `spawn`— the spawned function is considered as a separate task only if it has a different priority than its caller. In terms of the DAG model, one can think of a ***task*** being a SP DAG with a single source and a single sink whose nodes all have the same priority.

I-Cilk incorporates a type checking module that statically checks for priority inversions [19]. For scheduling analysis, we assume that there are no ***priority inversions*** — namely, a node (a `sync` or `get`) with higher priority never waits for a node with lower-priority to finish. Recall that in the bound presented in Section 1, the running time of a high-priority task does not depend on the work of any lower-priority task. However, if a high-priority task can block on a lower-priority task, then the latency of the high-priority task necessarily depends on the latency (and therefore, work) of the low-priority task — therefore, the no priority inversion assumption is essential to prove these bounds.

3 ADAPTIVE PRIORITY SCHEDULING

This section describes APS, the algorithm for running an interactive application with multiple priority levels on a multicore machine with P processors. As mentioned in Section 1, APS takes inspiration from A-GREEDY, an adaptive scheduling algorithm designed for scheduling multiple parallel jobs on a shared platform. Like A-GREEDY, APS uses a two-level scheduler. The top level ***processor allocator*** operates at the granularity of a scheduling quantum L, and decides how many processors to allocate to each priority level for the next quantum. This allocation does not change for the duration of the quantum.

At the second level, each priority level has its own scheduler for mapping its ready nodes onto its currently allocated cores. This scheduler is a ***work-conserving scheduler*** — given p allocated processors for the current quantum, at each step, if at least p nodes are ready, it arbitrarily picks any p and schedules them. Such steps are called ***complete steps*** since all allocated processors are being used to do work. If fewer than p nodes are ready, then it schedules all ready nodes. Such steps are called ***incomplete steps*** since not all processors may be used to do work.

To aid the processor allocation, APS also consists of a ***desire-calculation*** module for each priority level ℓ. This module monitors the scheduler of each priority ℓ during each quantum and calculates the ***desire*** d_q^ℓ — the number of cores to request for priority ℓ tasks (all together) for the next quantum q. Given the desires from each priority, the processor allocator assigns cores to each priority level — it simulates promptness by always assigning available cores to the highest priority-level (up to the limit of its desire) before assigning cores to the lower priority levels.

This desire-calculation algorithm is similar to A-GREEDY [3]. The algorithm uses two performance parameters which can be used to make trade-offs. The first is called the ***efficiency parameter*** δ and the second is a ***responsiveness parameter*** ρ. If the priority ℓ scheduler achieved at least δL complete steps in quantum $q-1$, it decides that the quantum was ***efficient*** for the priority ℓ scheduler; otherwise it declares that the quantum was ***inefficient***. In addition, if the quantum $q-1$ was allocated $p_{q-1}^\ell = d_{q-1}^\ell$ processors, then the quantum is said to be ***satisfied***, otherwise, it is a ***deprived*** quantum. Thus, each scheduler classifies each quantum as 4 possible classifications: efficient-satisfied, efficient-deprived, inefficient-satisfied and inefficient-deprived. As done by A-GREEDY, we use three of these: efficient-satisfied, efficient-deprived and inefficient — we do not care if an inefficient quantum is satisfied or deprived.

Based on this classification and the desire of quantum $q-1$, the desire of quantum q is calculated using the algorithm described in

1 **if** no ready nodes of priority ℓ **then** $d_q^\ell = 0$ **return**
2 **if** $d_{q-1}^\ell = 0$ **then** $d_q^\ell \leftarrow 1$ **return**
3 **if** more than $(1-\delta)L$ incomplete steps in $q-1$
4 **then** $d_q^\ell \leftarrow d_{q-1}^\ell / \rho$ **return** ▷ inefficient
5 **if** $p_{q-1}^\ell = d_{q-1}^\ell$ ▷ efficient and satisfied
6 **then** $d_q^\ell \leftarrow \rho d_{q-1}^\ell$ **return**
7 $d_q^\ell \leftarrow d_{q-1}^\ell$ **return** ▷ efficient and deprived

Figure 1: The desire calculation algorithm for quantum q and priority level ℓ.

Figure 1. The idea is simple. When the previous quantum is inefficient, then we assume that the desire was too high and reduce it. When the previous quantum is efficient and satisfied, we speculate and ask for more processors. When the previous quantum was efficient, but deprived, we used the allocated processors efficiently, but since the allocation was smaller than what we asked, we don't know if our calculation of desire was too low or too high. Therefore, we do not change our demand.

There is one important difference from the original A-GREEDY algorithm. A-GREEDY doesn't consider the possibility of a program having no ready nodes as an unfinished job has at least one ready node. However, in APS, there may be no ready nodes of a particular priority level at certain times. In this case, the scheduler makes its desire 0 and restarts the desire at 1 when new ready nodes appear.

Based on the desire of all the schedulers, the processor allocator allocates p_q^ℓ processors to the scheduler at priority level ℓ for quantum q. The processor allocator gives $p_q^1 = \min\{d_q^1, P\}$ processors to the highest-priority scheduler since this scheduler is scheduling the most latency-sensitive tasks. It then gives $p_q^2 = \min\{d_q^2, P - p^1\}$ processors to the priority 2 scheduler, $p_q^3 = \min\{d_q^3, P - (p_q^1 + p_q^2)\}$ processors to the priority 3 scheduler, and so on.[3] In other words, it satisfies the desire of the priority 1 scheduler first, then tries to satisfy the desire of the priority 2 scheduler, and so on until it runs out of processors.

4 RESPONSE GUARANTEES OF APS

This section describes the analysis of the response time provided by APS. We will consider an individual task τ at priority level ℓ, and we want to show a bound on the responsiveness or latency of the task — the amount of time that the task can be ***active*** or, in other words, the time that can pass between when the source node of the task becomes ready and its sink node is executed by *APS*.

Although APS's desire calculation algorithm is almost identical to A-GREEDY, there are significant differences in the analysis and the results. First, A-GREEDY does not dictate anything about how the processor allocator behaves and provides guarantees for individual jobs based on the behavior of the processor allocator. On the other hand, in APS, the processor allocator tries to simulate promptness by preferentially giving processors to higher priority levels. More significantly, A-GREEDY is designed for independent jobs sharing a multicore — these jobs do not interact and can be analyzed independently. APS is designed for a single interactive application where each second level scheduler is scheduling nodes of a particular priority. Tasks at different priority levels are not independent; for instance, a low-priority task may wait for the completion of a high-priority task. Thus, the performance of these tasks cannot be analyzed independently.

Definitions. There are a few important parameters that play a role in a task's response time: ***Competitive work*** W_τ includes the work of the task itself and any work of the same or higher priority that may execute in parallel with the task. This is all the work that the schedulers at the same or higher priority level can be executing while τ is active. The second is the ***span*** S_τ which is the longest chain of nodes that can hold up the completion of the task. Note that the longest chain may not contain only the nodes of the task itself since nodes of this task may wait on nodes of other tasks especially in the presence of futures. Formally, to define the span, we will define an ***induced DAG*** G_τ for task τ, which consists of all nodes that have a path to the sink node of τ, but do not have a (non-empty) path to the source node of τ. The span S_τ is the longest path in the induced DAG G_τ.

We will assume that our interactive application has ***no priority inversions*** — formally, this means that the induced DAG G_τ of task τ at priority level ℓ has no nodes of lower priorities (priorities with levels $> \ell$). Therefore, higher priority tasks do not wait for the completion of lower priority nodes. In addition, we will define k as the number of edges that go between nodes of different priority levels in G_τ. For tasks with $\ell = 1$ (highest priority), $k = 0$ by definition since their induced DAGs can only contain nodes with $\ell = 1$. Using these definitions, we will prove the following theorem:

Theorem 4.1. *A task τ with priority level 1 (the highest priority level) has a response time of at most $\frac{W_\tau}{\delta P} + O(\frac{S_\tau}{(1-\delta)} + L \lg P)$. A task τ with priority level $\ell > 1$ (a lower priority level) has a response time of at most $(1+\rho) \cdot \frac{W_\tau}{\delta P} + O(\frac{S_\tau}{(1-\delta)} + (\ell + k)L \lg P)$.*

Potential Function. We will prove the theorem using an amortized analysis with a potential function. For every node $u \in G_\tau$, we define the weight $w(u)$ of the node u as the length of the longest path from u to the sink of τ. We say that a ready node u has a potential $\Phi(u) = 2^{2w(u)}$; nodes which are not ready or have already executed have no potential.[4] For task τ, the potential of the task at time t is $\Phi_\tau(t) = \sum_{u \in G_\tau} \Phi(u)$.

The following Lemma follows from the fact that the longest path in the induced DAG G_τ is of length S_τ.

Lemma 4.2. *The total potential of the task when the task becomes active is at most 2^{3S_τ}; the final potential when the task ends is 0 and the potential of a task never increases.*

Proof. A task becomes active when its source node becomes active. The potential of the source node is at most 2^{2S_τ} since the source is part G_τ and thus can have weight at most S_τ. In addition, some other nodes u which are part of the induced subdag G_τ may be ready when the source becomes active and all of these can have potential at most 2^{2S_τ} for the same reason. However, all these nodes must be part of other futures which will eventually (recursively, possibly) join with τ — otherwise, there cannot be a path from u

[3] In practice, if the total desire of all levels is smaller than P, the remaining cores can be allocated arbitrarily — it does not impact the analysis.

[4] Since we consider a single task τ at a time, we omit it from the definition, even though the same node may be part of induced DAGs of multiple tasks and have potential with respect to all those tasks.

to the sink of τ. Since only two futures can participate in a join at a time, and the joins must eventually occur on some chain of length at most S_τ, the total number of such nodes that can join before sink of τ is 2^{S_τ}. Therefore, the total potential of all these nodes together is at most 2^{3S_τ} when the task becomes active. When the task completes, all of the nodes in its induced subdag have completed — therefore, the potential is 0.

At any point, the potential only changes when a node, say u finishes executing, and enables either 0, 1 or 2 children. It is easy to check that even if it enables two children, say v and v', the reduction in potential is $2^{2w(u)} - 2^{2w(v)} - 2^{2w(v')} \geq 2^{2w(u)} - 2^{2w(u)-1} - 2^{2w(u-1)} \geq \Phi(u)/2$. Therefore, the potential always reduces. □

Counting Efficient-Satisfied and Inefficient Quanta. We want to amortize certain quanta against others during certain time intervals, but we first need the following Lemma which is indirectly proved for A-GREEDY; here we state and prove it directly.

Lemma 4.3. *Each inefficient quantum q with desire d_q can be mapped to a prior efficient and satisfied quantum with desire d_q/ρ for the same scheduler such that no two inefficient quanta are mapped to the same efficient and satisfied quantum.*

Proof. Consider any inefficient quantum q with desire d_q. We go backward in time to find the nearest efficient and satisfied quantum q' with desire d_q/ρ and we say that inefficient quantum q is mapped to efficient and satisfied quantum q'. We can always find such a quantum since the desire cannot increase to d_q without there being a prior efficient and satisfied quantum with desire d_q/ρ. The only way another quantum q'' could be mapped to the same quantum q' is if q'' had desire d_q and was between q and q', so that q' is the closest efficient and satisfied quantum for both q and q''. However, if such a q'' existed, then the desire would have reduced to d_q/ρ after q'' and we would need another efficient and satisfied quantum q''' after q'' with desire d_q/ρ so that the desire can increase to d_q again, which is a contradiction since q''' is closer to d_q than q'. □

We now show how to amortize certain quanta against others during certain time intervals in the following lemma. Consider a period of time from t_1 to t_2 and say $I^\ell(t_1, t_2)$ is the number of inefficient quanta within this period for priority ℓ and say $ES^\ell(t_1, t_2)$ is the number of efficient-satisfied quanta.

Lemma 4.4. *Consider a scheduler at priority ℓ and say that the desire of this scheduler is never 0 between time t_1 and t_2. Say $I^\ell(t_1, t_2)$ is the number of inefficient quanta between time t_1 and t_2 and $ES^\ell(t_1, t_2)$ is the number of efficient and satisfied quanta within the same time. (We ignore partial quanta at the beginning and end of the period). Then, $I^\ell(t_1, t_2) \leq ES^\ell(t_1, t_2) + \lg_\rho P$ and $ES^\ell(t_1, t_2) \leq I^\ell(t_1, t_2) + \lg_\rho P$.*

Proof. The two statements are somewhat symmetric — we consider the first one. From Lemma 4.3, we can find a mapping from inefficient quanta to efficient and satisfied quanta. There are two cases: Either q's mapped quantum q' is after time t_1 and therefore within the interval t_1 to t_2 — there can be only $ES^\ell(t_1, t_2)$ such quanta. Or this mapped quantum is before t_1 and we will now argue that there can be at most $\log_\rho P$ such quanta. We know that there were no other inefficient quanta with desire d_q between q' and q and therefore, no other inefficient quanta with desire d_q between t_1 and quantum q. Therefore, there can be only one inefficient quantum with desire d_q which cannot be mapped to an efficient and satisfied quantum after time t_1. There are only $\log_\rho P$ different possibilities of desire since desire increases and decreases multiplicatively by factor ρ. Hence there can only be $\log_\rho P$ inefficient quanta that cannot be mapped. Considering both cases, we have $I^\ell(t_1, t_2) \leq ES^\ell(t_1, t_2) + \lg_\rho P$.

We can do a similar sort of mapping from efficient and satisfied quanta to inefficient quanta. We can uniquely map an efficient and satisfied quantum q with desire d_q to the closest inefficient quantum q' later in time with desire ρd_q— unless q is the last quantum with desire d_q in the time period (or ever). Again there are only $\log_\rho P$ quanta that cannot be mapped, giving us $ES^\ell(t_1, t_2) \leq I^\ell(t_1, t_2) + \log_\rho P$. □

4.1 Response time of tasks with priority 1

To build intuition, we will begin by analyzing the response time of highest priority tasks (level 1). We first bound the number of efficient-deprived quanta for the priority 1 scheduler while priority 1 task τ is running. Note that the priority 1 scheduler is only deprived when it has a desire larger than P.

Lemma 4.5. *The total number of efficient and deprived quanta for priority 1 scheduler during the execution of τ is at most $W_\tau/(\delta LP)$*

Proof. During deprived quanta, the priority 1 scheduler has P processors. Since they are also efficient, at least δL steps are complete steps where all P processors are doing ready work. Thus, the work done during these quanta is at least δLP. Since all work done by the priority 1 scheduler while τ is executing is part of competitive work W_τ, we can have at most $W_\tau/(\delta LP)$ such quanta. □

We next show that each inefficient quantum reduces the potential of task τ.

Lemma 4.6. *During any incomplete step of the priority 1 scheduler, the potential of all currently active priority 1 tasks decreases by a factor of 2. Thus, during an inefficient quantum for priority 1 scheduler, the potential of a currently active priority 1 task τ decreases by a factor of $2^{(1-\delta)L}$.*

Proof. While a task τ is active, some node(s) in its G_τ must be ready and these nodes are priority 1 nodes (there are no priority inversions). During an incomplete step for the priority 1 scheduler, all priority 1 ready nodes and, therefore, all ready nodes $u \in G_\tau$ are executed. A particular ready node u enables at most two children, say v and v'. After u executes, the potential due to v and v' is $\Phi(v) + \Phi(v') = 2^{2w(v)} + 2^{2w(v')} \leq 2 \times 2^{2w(u)-2} = \Phi(u)/2$. Thus, after an incomplete step, the potential due to all ready nodes $u \in G_\tau$ reduces by at least a factor of 2, reducing Φ_τ by at least a factor of 2. The potential decrease for inefficient quanta follows since they contain at least $(1 - \delta)L$ incomplete steps. □

The following lemma follows from Lemmas 4.6 and 4.2.

Lemma 4.7. *The total number of inefficient quanta during the execution of a priority 1 task τ is at most $3S_\tau/((1 - \delta)L)$.*

Proof. When a task τ starts executing, the total potential of the task is at most 2^{3S_τ}. (It can be lower since some nodes $u \in G_\tau$ may already have been executed before the task starts if they are part of other tasks.) In every inefficient quantum, it reduces by a factor of $2^{(1-\delta)L}$. Therefore, if the total number of inefficient quanta before the task finishes is I, then $2^{3S_\tau}/2^{I(1-\delta)L} < 1$. Solving for I gives us the number of inefficient quanta. □

Combining with Lemma 4.4 gives us the following corollary since the priority 1 scheduler's desire is never 0 while τ is active.

Corollary 4.8. *The total number of efficient and satisfied quanta during the execution of a high-priority task τ is at most $3S_\tau/((1-\delta)L) + \log_\rho P$.*

Lemmas 4.5, 4.6 and Corollary 4.8 gives us the result stated in Theorem 4.1 for priority 1 tasks.

4.2 Response time of tasks with priority 2

Analyzing the response time of lower priority tasks is more challenging for the following reasons.

1. Lemma 4.5 depends on the fact that when the priority 1 scheduler has an efficient and deprived quantum, all P are allocated to it and are doing competitive work. When a priority 2 scheduler has an efficient and deprived quantum, its allocation is not necessarily P since some processors may have been allocated to the priority 1 scheduler. Therefore, an analog of Lemma 4.5 does not directly hold.
2. The induced DAG G_τ of a priority 2 task τ can have both priority 1 and 2 nodes and some of the potential of τ belongs to priority 1 nodes. Therefore, an analog of Lemma 4.6 does not hold directly — an inefficient quantum for the priority 2 scheduler does not reduce the potential of an active priority 2 task if most of its potential is in the priority 1 nodes (which may not execute during this quantum). Therefore, we must consider both priority 1 and 2 schedulers when analyzing a priority 2 task.

When analyzing a priority 2 scheduler, we must consider the quantum classifications of both priority 1 and 2 schedulers — giving us 9 types of quanta. However, we will divide them into two types of quanta. The first type — we call it ***type A*** — is when either the priority 1 or the priority 2 scheduler is efficient and deprived. We will bound the number of type A quanta using the competitive work of τ. The second type — called ***type B*** — is when neither is efficient and deprived and therefore priority 1 and 2 schedulers are either efficient and satisfied or inefficient during these quanta. We will bound the number of type B quanta as a function of the span S_τ. Note that if either scheduler has desire 0, the other scheduler can still completely classify the quantum. This covers all cases since, while τ is active, one or the other scheduler must have ready nodes.

Lemma 4.9. *The total number of type A quanta while a priority 2 task τ is executing is at most $(1+\rho)W_\tau/(\delta LP) + \log_\rho P$.*

Proof. We consider a few sub-cases of type A quanta.

Priority 1 scheduler was efficient and deprived: Say there were X such quanta and the total priority 1 work done during these quanta was W_1 (no lower priority work executed since the priority 1 scheduler got all cores). During each quantum, at least δLP priority 1 work was done; therefore, $W_1 \geq X\delta LP$.

Priority 1 scheduler was efficient and satisfied and priority 2 scheduler was efficient and deprived: Say there were Y such quanta and say W_2 priority 1 and 2 work was done during these quanta. Again all P processors were allocated jointly to the two schedulers and both schedulers were efficient. Therefore, again, we have $W_2 \geq Y\delta LP$.

Considering just these two cases, all the work done in these quanta is competitive work for τ. Therefore, we have $W_\tau \geq W_1 + W_2 \geq (X+Y)\delta LP$. Therefore, we have $X + Y \leq W_\tau/(\delta LP)$. We have one more case.

Priority 1 scheduler was inefficient and priority 2 scheduler was efficient and deprived: This is the complicated case. Consider one such quantum q and suppose the desire of the priority 1 scheduler during this quantum was d_q^1. The allocation for the priority 1 scheduler during quantum q was $p_q^1 = d_q^1$. Therefore, the allocation to the priority 2 scheduler was $p_q^2 = P - d_q^1$ (since it was deprived, it got all remaining cores). Since the priority 2 scheduler was efficient, the total work done during that quantum was at least $W_q^2 \geq \delta L p_q^2 = \delta L(P - d_q^1)$.

Now we look for the previous quantum q' when the priority 1 scheduler was efficient and satisfied with desire and allocation exactly $p_{q'}^1 = d_{q'}^1 = d_q^1/\rho$. As argued in Lemma 4.4, we can always find such a quantum and there is a unique mapping from q to q'. We have two cases: First, q' occurred after τ became active. The total priority 1 work done during q' is at least $W_{q'}^1 \geq \delta L a_{q'}^1 = \delta L d_q^1/\rho$ and all this work is part of competitive work W_τ since it is executed by a higher priority scheduler while τ is active. Therefore, we have $W_{q'}^1 + W_q^2 \geq \delta L d_q^1/\rho + \delta L(P - d_q^1) \geq \delta LP/\rho$. All this work was part of W_τ since it occurred while the task was active and was at a higher or the same priority. Second, q' may have occurred before τ started executing — by logic similar to Lemma 4.4, there can be at most $\log_\rho P$ such quanta while τ was executing. Say there were Z such quanta (where priority 2 scheduler is efficient and deprived and priority 1 scheduler is inefficient) — we would get $Z \leq W_\tau\rho/(\delta LP)$. We get $X + Y + Z \leq (1+\rho)W_\tau/(\delta LP) + \log_\rho P$ type A quanta. □

Now we must consider type B quanta. We will use the potential function to bound these quanta using the span of τ. The fundamental reason this analysis is complicated is as follows: A priority 2 task can have both priority 1 and 2 nodes in its G_τ — therefore, some of the nodes that contain the potential of this task are being executed by each scheduler, potentially. If both priority 1 and 2 schedulers were inefficient in some quantum q, then we could use an argument similar to the one in Lemma 4.6 to show that the potential decreased for the priority 2 task τ during that quantum. However, if only one of the two schedulers is inefficient (and the other is efficient and satisfied) or if both schedulers are efficient and satisfied, then we cannot say anything about the total decrease in potential.

Therefore, we divide the priority 2 task potential into two components — one consisting of priority 1 nodes and one consisting of priority 2 nodes. That is, $\Phi_\tau^1(t) = \sum_{\text{priority 1 node } u \in G_\tau} \Phi(u)$ and $\Phi_\tau^2(t) = \sum_{\text{priority 2 node } u \in G_\tau} \Phi(u)$. The total potential $\Phi_\tau(t) = \Phi_t^1(\tau) + \Phi_t^2(\tau)$.

Now, we hope to make an argument similar to the one in Lemma 4.6 to say that if, for instance, the priority 1 scheduler had an inefficient quantum, then Φ_τ^1 will decrease by a large fraction in

that quantum. Unfortunately, this is not always true. Even though Φ_τ always decreases (as shown in Lemma 4.2), it is not true that either $\Phi^1_t(\tau)$ or $\Phi^2_t(\tau)$ always decreases. In fact, sometimes $\Phi^1_\tau(t)$ may become 0 if there are no priority 1 ready nodes from G_τ at time t and then increase again later when some priority 2 node executes and enables a priority 1 child. The same thing can happen to $\Phi^2(\tau)$. In other words, potential sometimes gets transferred from $\Phi^2(\tau)$ to $\Phi^1(\tau)$ and vice versa, even though their sum never increases.

So, we first state a Lemma about how many times this potential transfer can happen — in other words, how many times can Φ^1_τ or Φ^2_τ increase while τ is active. The proof of the following lemma relies on the fact that there are at most k edges between nodes of different priority levels in G_τ and the potential of priority 1 can increase only if a priority 2 node enables a child node of priority 1 (and vice versa).

Lemma 4.10. *The total number of times Φ^1_τ or Φ^2_τ can increase is at most k.*

Proof. The only time Φ^1_τ can increases is when a priority 2 node $u \in G_\tau$ executes and enables a child node v which is a priority 1 node. By an argument similar to Lemmas 4.2 and 4.6, if a priority 1 node enables a priority 1 node, the potential only decreases since child nodes have much lower potential than their parents. There is a symmetric argument for priority 2 potential increase. Since there are only k edges between nodes of different priorities in G_τ, this increase can occur at most k times. □

The quantum during which a potential transfer occurs is called a ***transfer quantum***. We divide τ's execution into ***phases*** between these transfer quanta — the first phase starts when τ is created and ends when the first potential transfer happens. All subsequent phases begin when the previous phase ends and ends at the next potential transfer or when τ finishes executing. Therefore, there are at most k transfer quanta while τ is executing and at most $k+1$ phases.

We now consider each phase individually. Consider a phase Q that begins at time t_1 and ends at time t_2. We want to bound the number of type B quanta during this phase (not counting the transfer quanta that begin and end the phase). The potential of task τ at time t_1 is $\Phi_\tau(t_1) = \Phi^1_\tau(t_1) + \Phi^2_\tau(t_1)$ and at time t_2 is $\Phi_\tau(t_2) = \Phi^1_\tau(t_2) + \Phi^2_\tau(t_2)$. From Lemma 4.2, we have $\Phi_\tau(t_2) \le \Phi_\tau(t_1)$.

Lemma 4.11. *Say there are $I^2 \ge 0$ inefficient quanta for priority 2 scheduler during the phase Q and $I^1 \ge 0$ inefficient quanta for the priority 1 scheduler during the phase Q. Then $\Phi_\tau(t_2) \le \Phi_\tau(t_1)/(2^{(1-\delta)L\min\{I^1,I^2\}})$. Therefore, $\min\{I^1, I^2\} \le \frac{\lg\Phi_\tau(t_1) - \lg\Phi_\tau(t_2)}{(1-\delta)L}$.*

Proof. Since there was no potential transfer during the phase, every inefficient quantum decreased the priority 2 potential Φ^2_τ by a factor of $2^{(1-\delta)L}$ by the same logic as Lemma 4.6. Therefore, I^2 inefficient quanta reduced Φ^2_τ by a factor of $2^{(1-\delta)LI^2}$ — implying $\Phi^2_\tau(t_2) \le \Phi^2_\tau(t_1)/2^{(1-\delta)LI^2}$. Note that in this case, the potential Φ^2_τ may become 0 at some point during the quantum — however, the above inequality trivially holds in that case and the potential cannot increase again during the quantum since it is not a transfer quantum. Using a similar argument, we have $\Phi^1_\tau(t_2) \le \Phi^1_\tau(t_1)/2^{(1-\delta)LI^1}$. Therefore, we have $\Phi_\tau(t_2) = \Phi^1_\tau(t_2) + \Phi^2_\tau(t_2) \le \Phi^1_\tau(t_1)/2^{(1-\delta)LI^1} + \Phi^2_\tau(t_1)/2^{(1-\delta)LI^2} \le (\Phi^1_\tau(t_1) + \Phi^1_\tau(t_2))/2^{(1-\delta)L\min\{I^1,I^2\}}$

More algebra gives the bound on $\min\{I^1, I^2\}$. □

The following lemma follows from Lemma 4.4.

Lemma 4.12. *The total number of type B quanta during a phase Q that goes from time t_1 to time t_2 is at most $2\frac{\lg\Phi_\tau(t_1) - \lg\Phi_\tau(t_2)}{(1-\delta)L} + \log_\rho P$.*

Proof. Recall that, by Lemma 4.4, we know that during the phase, the total number of efficient and satisfied quanta is at most $ES^2 \le I^2 + \log_\rho P$ for the priority 2 task and $ES^1 \le I^1 + \log_\rho P$. In addition, the total number of type B quanta is at most $\min\{I^1 + ES^1, I^2 + ES^2\}$ since both schedulers have to inefficient or efficient and satisfied during these quanta. Therefore, the total number of type B quanta is at most $2\min\{I^1, I^2\} + \log_\rho P$ giving us the required bound. □

We can now bound the type B quanta for task τ — the proof uses Lemmas 4.2 and 4.12 to argue about initial potential and change in potential.

Lemma 4.13. *The total number of type B quanta across the entire execution of the priority 2 task is $O(S_\tau/((1-\delta)L) + (k+1)\log_\rho P)$.*

Proof. We know that the beginning potential of the task when it is created is at most 2^{3S_τ} and the final potential right before the task ends is less than 2. Say that there are $X \le k$ phases for task τ. Say there are B_i type B quanta during phase i and phase i begins at time t_{i-1} and ends at time t_i. From Lemma 4.12, we know that $B_i \le 2\frac{\lg\Phi_\tau(t_{i-1}) - \lg\Phi_\tau(t_i)}{(1-\delta)L} + \log_\rho P$. Therefore, if we add B_i's over all phases, we get

$$\begin{aligned}
\sum_{i=1}^{X} B_i &\le \sum_{i=1}^{X} 2\frac{\lg\Phi_\tau(t_{i-1}) - \lg\Phi_\tau(t_i)}{(1-\delta)L} + \log_\rho P \\
&\le 2\frac{\lg\Phi_\tau(t_0) - \lg\Phi_\tau(t_X)}{(1-\delta)L} + X\log_\rho P \\
&\le 2\frac{\lg 2^{3S_\tau}}{(1-\delta)L} + X\log_\rho P \\
&\le 2\frac{3S_\tau}{(1-\delta)L} + (k+1)\log_\rho P
\end{aligned}$$

In addition, we have $k+2$ transfer quanta, but that term is subsumed by the last term. □

Adding type A and type B quanta and multiplying by L gives us the bound on response time for priority 2 tasks as stated in Theorem 4.1. Intuitively, it turns out that the analysis of type B quanta does not change at all; however, for type A quanta, we get an additional $\ell\log_\rho P$ quanta for the following reason. Consider the proof of Lemma 4.9 and look at the third case where priority 1 quanta are inefficient and priority 2 quanta are efficient and deprived. For a scheduler at priority ℓ, in the worst case, all schedulers of level $< \ell$ may be inefficient — if all of these quanta can be mapped to prior quanta that started after τ, then like in the proof, we can account for them using competitive work. However, if any of them cannot be mapped, then we get an additive factor of $\log_\rho P$, giving us a total additive factor of $(\ell-1)\log_\rho P$.

4.3 Generalizing to priority level ℓ

Now we consider some arbitrary priority level. Again, we can divide quanta into two categories — type A quanta are those where some scheduler with priority $x \leq \ell$ is efficient and deprived and type B quanta are quanta where all schedulers with priority $x \leq \ell$ are either inefficient or efficient and satisfied.

It turns out that the analysis of type B quanta does not change as we increase the priority level ℓ. In particular, again, we can divide the potential into ℓ different components and the number of transfer quanta is still bounded by k. In each phase (time between transfer quanta), we can easily generalize Lemma 4.12 to show the same bound on the decrease in potential and then use it to show Lemma 4.13. In other words, increasing priorities does not increase the bound on type B quanta.

Type A quanta are a different beast since the bound depends on ℓ — but only in the last term.

Lemma 4.14. *The total number of type A quanta while a priority ℓ task τ is executing is at most $(1+\rho)W_\tau/(\delta LP) + \ell \log_\rho P$.*

Proof. Recall that if a scheduler with priority level x was efficient and deprived, no scheduler with priority $> x$ gets any processors. First, let us consider the simple case: All schedulers which got any processors were efficient (either satisfied or deprived). This includes all cases where the scheduler at priority level $x \leq \ell$ was efficient and deprived and all schedulers with priority level $< x$ were efficient and satisfied. (In Lemma 4.9, this includes the first two cases.) Say there were A such quanta and W_1 work was done over all these quanta on priority $1-\ell$ tasks. Since all P processors were allocated to priorities $\leq \ell$ and the quanta were efficient, we have $W_1 \geq \delta LA$. Since all this work is competitive work of τ, we get $A \leq W_\tau/(\delta L)$.

Now we must consider the case where some quanta were inefficient and some quanta were efficient — consider one such quantum q. By definition of type A quanta — some scheduler with priority level $x \leq \ell$ is efficient and deprived during q. All schedulers with priority level $< x$ are either efficient and satisfied or inefficient — the worst case is that they are all inefficient and we will consider that case. Consider the inefficient quantum q for priority level $y < x$ and say that the allotment of that quantum was $p^y_q = d^y_q$ (since there were processors left over for lower priority tasks). From Lemma 4.3, we can find a previous efficient and deprived quantum $q'(y)$ with allotment $p^y_{q'(y)} = d^y_{q'(y)} = d^y_q/\rho$. We can find a similar mapping for all priority levels $y < x$ with inefficient quanta (though the $q'(y)$ may be different for each y).

There are two cases: For all y, this $q'(y)$ occurred after task τ became active. In this case, we know that $\sum_{y<x} p^y_{q'(y)} = \sum_{y<x} d^y_q/\rho$ Therefore, the total work done by the priority $1-x-1$ schedulers during these mapped quanta (which are efficient) is at least $W_{q'} \geq \delta L \sum_{y<x} d^y_q/\rho$. In addition, the work done by the scheduler at priority level x, which is efficient and deprived, is at least $W_q \geq \delta L(P - \sum_{y<x} d^y_q)$. Adding these together, we get $W_q + W_{q'} \geq \delta LP/\rho$. If there are B such quanta, and all this work is competitive work of τ (since it was all executed by a higher or equal priority scheduler while τ was active), we get $B \leq \rho W_\tau/(\delta L)$.

The final case to consider is that for some y, the mapped quantum $q'(y)$ occurred before task τ became active. In this case, we cannot argue that the work done during $q'(y)$ was competitive work. However, from Lemma 4.4, we know that we can find at most $\log_\rho P$ such quanta for each priority level $< x$. Therefore, the total number of quanta of this type is $(\ell-1)\log_\rho P$.

Adding all cases gives us the bound on type A quanta. □

Since the bound on type B quanta does not change, combining Lemmas 4.14 and 4.13, and multiplying with L gives us the bound stated in Theorem 4.1.

5 EMPIRICAL EVALUATION

This section empirically evaluates I-Cilk, which implements APS. We evaluated I-Cilk using two microbenchmarks and three moderately sized application benchmarks (that range from $1.1k$ to $1.5k$ lines of code). The microbenchmarks are written such that they can be configured to generate different workloads to allow us to better evaluate different aspects of the scheduler. The application benchmarks consist of richer workloads that simulate real-world interactive applications.

Implementation of I-Cilk. I-Cilk extends Cilk-F, a Cilk dialect that uses ***proactive work stealing*** [24] to schedule futures. I-Cilk incorporate latency-hiding I/O support as described in [23], a priority type system [19], and an adaptive processor allocator. We don't discuss the I/O support and the type system, as they are not the focus of this work.

I-Cilk implements a two-level adaptive scheduling strategy as described in Section 3. The top-level master scheduler adaptively allocates workers (surrogates of processing cores) to priority levels. The second level uses an extension of proactive work stealing to schedule tasks within a priority level. Each worker periodically updates its utilization based on how long it spent working versus looking for work to do. The master is a thread that sleeps for the duration of a quantum (L), collects utilization reported, calculates the core utilization at each priority level, computes the desire of each priority level, and allocates workers to each level. If there are left-over cores not allocated based on desires, the runtime assigns half of the left-over cores to each priority level, starting from the highest level to the lowest until it runs out of cores. The master alerts a worker to switch priority by setting a per-worker flag, checked at each strand boundary (i.e., at a `spawn`, `sync`, `create`, and `get`) or at a user-specified yield via a runtime library call,[5] and switches to its new priority level if the flag is set.

In classic work stealing, each worker has one deque storing its work items. In I-Cilk, since each worker may work on different priority levels throughout execution, each worker ends up having multiple pools of deques, one for each priority level, and like in Cilk-F, there can be multiple deques within each pool a worker can generate additional deques when it needs to switch level or when it generates a new task with a different priority level from its own.

Experimental Setup. We empirically evaluate I-Cilk by comparing its performance against that of Cilk-F. To enable fair comparison, we have extended Cilk-F with the same latency-hiding I/O support. Our experiments ran on a computer with 2 Intel Xeon Gold 6148 processors, each with 20 2.40-GHz cores. Each core has a 32-kB L1 data and 32-KB L1 instruction cache, and a private 1

[5] This yield functionality is an optimization for programs with long strands, allowing the programmer to indicate to the runtime when it is safe to interrupt.

MB L2 cache. Hyperthreading was enabled, and each core had 2 hardware threads. Each processors has a 27.5 MB shared L3 cache, and there are 768 GB of main memory. I-Cilk and all benchmarks were compiled using the Tapir compiler [21] (based on clang 5.0.0), with -O3 and -flto. Experiments ran in Linux kernel 4.15.

For the choices of runtime parameters, we expect δ (efficiency parameter) to range between 0 and 1, and is likely close to 1, and ρ (responsiveness parameter, or how fast to grow the desire) to range between 1 and 2. For all evaluations, we have tested several sensible configurations of runtime parameters. Unless stated explicitly (i.e., when testing the sensitivities to paratmers), we show the configuration that best benefits the high-priority tasks. We discuss how runtime parameters may impact execution time at the end.

5.1 Evaluation of Microbenchmarks

We use two microbenchmarks to answer the following questions. First, does I-Cilk appropriately prioritize tasks in the order of their priorities? Second, how much overhead does the two-level adaptive scheduling strategy incur in I-Cilk? Finally, how do the changes in the scheduling parameters of I-Cilk (i.e., scheduling quantum length L, responsiveveness parameter ρ, and utilization parameter δ) impact execution times of tasks at different priority levels? To answer these questions, we utilized the microbenchmarks to generate different workloads and compare the average execution times of tasks at each priority level when running on I-Cilk versus Cilk-F.

Microbenchmark. The first microbenchmark, `fib-ep`, computes the 42nd fibonacci number with a serial base case of 2, which has ample parallelism. The second microbenchmark, `fib-rp`, similarly computes the 44th fibonacci number but with a serial base case of 40, thus restricting the number of parallel strands to 8 for each priority level. In both microbenchmarks, fibonacci computations are spawned for high (H), medium (M), and low (L) priority in succession. The serial fibonacci base case in I-Cilk uses yield calls to check if the worker needs to switch priorities.

Since `fib-rp` does not have much parallelism, it is an adversarial workload for I-Cilk. Each task cannot fully utilize all the cores, so the desire for each priority level will oscillate between two different values, as falling slightly below the parallelism or overshooting and causing low utilization.

Prioritization of Tasks. We compare the runtime of tasks at each priority level running on I-Cilk and Cilk-F. Here, we choose the optimal runtime parameters for I-Cilk (sensitivity to parameters evaluated later). Figure 2 shows the execution times for tasks at each level, with overhead and standard deviation shown in parentheses. The overhead is computed by comparing to **`fib-ideal`**, a single instance of the fib computation running on Cilk-F (as opposed to three contending for cores). I-Cilk appears to prioritize tasks appropriately. For `fib-ep`, I-Cilk across the board executes H and M tasks faster with a slight degradation for the L task. For `fib-rp`, the H task oscillates between desires that either undershoot its parallelism level or overshoot. When it overshoots, it takes cores away from M and L tasks yet doesn't utilize them fully. Cilk-F does not have the same issue as it treats all tasks equally. Consequently, I-Cilk experiences higher degradation compared to Cilk-F, but not by too much.

The lower overhead and standard deviation on the M task in I-Cilk indicate that the priority relationship between M and L is also being respected. To verify this, we ran the same I-Cilk microbenchmarks with just the H and M tasks (not shown) and saw that the execution times for the H and M tasks were similar to that shown in Figure 2.

Overhead of the Adaptive Scheduling Strategy. We can look at the overhead of the H task of I-Cilk to gauge the overhead for adaptive scheduling. From the results in Figure 2, we found the overhead be low. For `fib-ep` the overhead is also minimal for M and L tasks; they complete in the time it would take to execute 2 and 3 fib computations respectively. This reflects the fact that the medium priority work has to wait for the high priority work to complete, and low similarly has to wait for the medium priority work.

Sensitivity to Responsiveness Parameter ρ. When there is ample parallelism, as in `fib-ep`, larger ρ reduce the execution time at every level because it is most efficient to be aggressive about growing the desire; it also benefits lower priorities because the H tasks back off their desire more quickly. When there is little parallelism, as in `fib-rp`, the correct values are less obvious. For the H task, it is important that ρ is large enough that the desire can grow quickly; however, it is in some cases even more important that the floor of the high priority desire can reach exactly the number of parallel strands in the H tasks to avoid oscillating between too many and too few processors. In Figure 3, this is the case only for a ρ of 1.2 or 2, and the overhead on H task of `fib-rp` reflects this. For M and L tasks, however, their execution times suffer with ρ of 2 because H task wastes more processing cycles (low utilization when it overshoots the desire) and deprives the M and L tasks.

Sensitivity to Scheduling Quantum Length L. Figure 4 shows the microbenchmark execution times of I-Cilk compared to `fib-ideal` with different L. As L increases there is less overhead seen on the one core execution. For parallel executions, a longer L means that it takes longer for a priority level to reach the desired number of cores (especially evident in the H task). The M task also suffers slightly, as it takes longer for the H task to give up its cores once it's done executing; this is especially evident with `fib-rp`, whih has a low parallelism H task can waste cores due to low utilization.

5.2 Evaluation of Application Benchmarks

Our application benchmarks are designed to simulate real-world interactive applications. These benchmarks have much richer characteristics compared to the microbenchmarks in that they generate tasks of variable sizes and have different mixtures of I/O operations and computations. The first bench `job` most closely resemble traditional task-parallel workloads but incorporates priorities; it simulates a job scheduler that schedules compute-heavy workloads. The second bench, `email`, simulates an email application that has a non-negligible amount of I/O operations, but also a healthy amount of computation. The last bench, `proxy`, simulates a proxy server that has very little background computation and a high I/O-to-compute ratio. These benchmarks are nondeterministic even when we fix the inputs (e.g., the sequence of requests sent by a client), as inputs arrive with nondeterministic timing.

The Job Server. The `job` server generates parallel jobs using a Poisson process at random intervals and schedules them. Priorities

	fib-ep					fib-rp	
	T_1	T_4	T_8	T_{16}	T_{20}	T_{16}	T_{20}
H	11.61 (1.00×, 0.0%)	5.98 (2.06×, 27.7%)	2.24 (1.54×, 35.9%)	1.27 (1.74×, 21.4%)	0.95 (1.64×, 27.2%)	0.95 (1.22×, 20.8%)	0.78 (1.00×, 0.5%)
Cilk-F M	23.22 (2.00×, 0.0%)	8.05 (2.77×, 7.1%)	3.67 (2.52×, 12.1%)	2.07 (2.84×, 7.8%)	1.73 (2.96×, 1.3%)	1.10 (1.42×, 19.9%)	1.04 (1.34×, 20.0%)
L	34.83 (3.00×, 0.0%)	8.62 (2.97×, 2.5%)	4.36 (2.99×, 0.0%)	2.18 (2.99×, 0.06%)	1.74 (2.98×, 1.1%)	1.43 (1.85×, 11.1%)	1.14 (1.47×, 14.1%)
H	12.53 (1.08×, 0.0%)	3.13 (1.08×, 0.2%)	1.57 (1.08×, 0.1%)	0.79 (1.08×, 0.4%)	0.63 (1.08×, 0.3%)	0.87 (1.13×, 0.2%)	0.87 (1.12×, 0.3%)
I-Cilk M	25.05 (2.16×, 0.0%)	6.24 (2.15×, 0.1%)	3.13 (2.15×, 0.1%)	1.57 (2.15×, 0.1%)	1.26 (2.15×, 0.1%)	1.14 (1.47×, 0.6%)	0.87 (1.12×, 0.1%)
L	37.52 (3.23×, 0.0%)	9.36 (3.22×, 0.1%)	4.69 (3.22×, 0.0%)	2.34 (3.22×, 0.0%)	1.88 (3.22×, 0.1%)	1.68 (2.17×, 2.1%)	1.27 (1.63×, 0.2%)

Figure 2: Execution times of `fib-ep` and `fib-rp` (seconds) run using vanilla Cilk-F and I-Cilk with $\delta = 0.9$ and $L = 1ms$. `fib-ep` was run with $\rho = 2$, `fib-rp` with $\rho = 1.2$. Overhead, relative to `fib-ideal`, and standard deviation are in parentheses.

		2 Jobs/s			3 Jobs/s			4 Jobs/s			5 Jobs/s		
		Avg.	95%	Jobs/s	Avg.	95%	Jobs/s	Avg.	95%	Jobs/s	Avg.	95%	Jobs/s
Cilk-F	mm	595	143	1.99	658	137	2.77	100	224	3.88	260	560	4.25
	fib	71	123	2.16	77	128	3.10	102	187	3.71	227	541	4.41
	sort	104	170	1.76	115	187	2.05	146	294	2.42	341	790	2.19
	sw	204	400	2.27	330	732	3.16	538	1072	3.61	1462	2154	4.12
I-Cilk	mm	41	68	2.17	41	62	2.79	44	84	3.78	44	74	5.00
	fib	75	109	1.82	87	178	3.11	109	253	4.24	126	268	4.88
	sort	107	206	1.99	152	382	2.79	188	552	2.97	318	858	3.52
	sw	328	624	2.44	492	1095	2.96	19961	35984	2.86	56881	76684	0.99

Figure 5: The average (Avg.) and 95 percentile (95%) flow time of different types of computations running on `job`, listed from highest to lowest priorities. All times are reported in milliseconds. The Job/s reports the throughput (how many instances per seconds executed). Times for I-Cilk were collected with $\rho = 2$, $L = 500\mu s$, and $\delta = 0.9$.

	$\rho = 1.2$	$\rho = 1.5$	$\rho = 1.75$	$\rho = 2$
H	0.64 (1.10×)	0.63 (1.08×)	0.63 (1.08×)	0.63 (1.08×)
fib-ep M	1.26 (2.15×)	1.25 (2.15×)	1.25 (2.15×)	1.25 (2.15×)
L	1.88 (3.22×)	1.88 (3.22×)	1.88 (3.22×)	1.88 (3.22×)
H	0.87 (1.12×)	1.09 (1.40×)	1.00 (1.29×)	1.04 (1.33×)
fib-rp M	0.87 (1.12×)	0.98 (1.26×)	1.16 (1.49×)	1.27 (1.63×)
L	1.27 (1.64×)	1.41 (1.82×)	1.38 (1.77×)	1.37 (1.77×)

Figure 3: Execution time of `fib-ep` and `fib-rp`, in seconds, run in I-Cilk with various ρ values on 20 processors with $L = 1ms$. Overhead (in parentheses) is relative to `fib-ideal`.

	$L = 100us$	$L = 500us$	$L = 1ms$	$L = 10ms$
H	13.18 (1.13×)	12.57 (1.08×)	12.53 (1.08×)	12.50 (1.08×)
fib-ep (T_1) M	26.37 (2.27×)	25.13 (2.16×)	25.05 (2.16×)	24.97 (2.15×)
L	39.11 (3.37×)	37.64 (3.24×)	37.52 (3.23×)	37.40 (3.22×)
H	0.63 (1.08×)	0.63 (1.08×)	0.63 (1.08×)	0.66 (1.12×)
fib-ep (T_{20}) M	1.26 (2.16×)	1.25 (2.15×)	1.25 (2.15×)	1.27 (2.18×)
L	1.89 (3.23×)	1.88 (3.22×)	1.88 (3.22×)	1.88 (3.24×)
H	0.87 (1.12×)	0.88 (1.13×)	0.87 (1.12×)	0.89 (1.15×)
fib-rp (T_{20}) M	0.87 (1.13×)	0.87 (1.13×)	0.87 (1.12×)	0.92 (1.19×)
L	1.28 (1.65×)	1.27 (1.64×)	1.27 (1.64×)	1.28 (1.65×)

Figure 4: Execution times, in seconds, running with I-Cilk using various quantum lengths (L) for `fib-ep` ($\rho = 2$) on 1 and 20 processors, and `fib-rp` ($\rho = 1.2$) when run on 20 processors. Overhead (in parentheses) is relative to `fib-ideal`.

are assigned based on the smallest-work-first principle, so jobs with the smallest work (i.e., one-core execution time) are assigned the highest priority. Such a scheduling policy is designed to minimize the average ***flow time***, time elapsed between when a job is generated and when it finishes executing, of jobs [6]. Four types of parallel jobs are used (from highest to lowest priority): a) matrix multiplication (`mm`, $n = 1024$), b) fibonacci (`fib`, $n = 36$), c) merge sort (`sort`, $n = 1.1e7$), and d) Smith-Waterman (`sw`, $n = 1024$).

We ran `job` on 20 cores with different L and ρ. At 2, 3, 4, and 5 jobs per second, the machine utilization is about 50%, 70%, 95%, and > 95%, respectively. Compared to Cilk-F, I-Cilk prioritizes tasks with higher priority. When the machine is not heavily loaded, I-Cilk provides higher throughput than Cilk-F because it implements smallest-work-first using priorities. I-Cilk compares favorably over Cilk-F regardless of the choice of L and ρ.

As the machine gets more loaded, I-Cilk prioritizes tasks at higher priority at the expense of the lowest priority tasks (e.g., `sw`). This generally translates to higher throughput for higher priority tasks but lower throughput for lower priority ones, and the overall system throughput may be lower than Cilk-F as a result. Here, the choice of L and ρ matters, and we have shown data with paramters that prioritize higher-priority tasks. With a longer L and/or a lower ρ, it would take longer for the high-priority tasks to gain processing cores and in turn benefits the lower priority tasks. Thus, a longer L and/or a lower ρ can lead to higher flow time for high-priority tasks but also higher overall system throughput.

The Email Client. The `email` bench simulates a multi-user shared email client. It contains five priority levels (highest to lowest): a) loop that handles user requests, b) a component to send emails, c) a component to sort emails, d) two equal-priority components: one to compress emails and one to uncompress and print emails, e) a loop to periodically check if there are uncompressed emails (due to print) that need to be compressed and trigger the compress component.

We ran `email` on 10 cores, using the other 30 cores to simulate clients connecting to `email`. Figure 6 shows the results with different client configurations with each client sending 1500 requests, except for the 30 client configuration where each client sends 2000 requests to allow for a longer execution time.

Due to space limitations, we only show data for one configuration of runtime parameters. Based on our evaluations however, I-Cilk uniformly provides shorter response time and send time (the two highest priority tasks) compared to Cilk-F regardless of the choice of L and ρ. When the number of clients are moderate (90 or less), I-Cilk also provides better sort time. As the number of clients become large (120), I-Cilk sacrifices the lowest priority tasks (periodic check of compression).

That means Cilk-F may be doing more work (the two lowest priority tasks) than I-Cilk, because `email` running on I-Cilk can skip checks from time to time if the loop does not get scheduled within the timer period. As a result, the duration of compress tasks is also higher on I-Cilk.

Similar to the observation in `job`, a longer L and a lower ρ means that it takes longer for the high-priority tasks to gain processing cores which leads to overall longer latency for higher priority tasks (e.g., response and send) but still outperforms that of Cilk-F.

The Proxy Server. The `proxy` server requests websites on behalf of clients, hiding the requestor's IP address, and sends the

		30 Users			60 Users			90 Users			120 Users		
		Avg	95%	99%	Avg	95%	99%	Avg	95%	99%	Avg	95%	99%
Cilk-F	resp	486.00	344.58	7433.00	2248.61	3050.62	18597.10	8223.66	7649.56	31378.00	18080.90	9894.68	37980.10
	send	501.84	374.74	7938.14	2287.18	3282.31	19415.30	8112.99	8113.83	33816.70	16724.80	10260.00	40010.80
	sort	2.40	4.39	19.33	6.94	10.14	41.64	19.87	18.82	73.07	43.68	23.12	85.63
	print	3.85	34.88	40.82	6.98	36.11	50.01	16.69	40.07	59.63	29.23	42.34	68.49
	comp	13.45	13.67	40.29	13.42	13.65	39.86	13.40	13.55	39.82	13.42	13.48	40.63
I-Cilk	resp	46.91	125.11	500.72	70.53	289.13	874.64	95.65	519.80	1273.61	122.47	679.27	1552.71
	send	392.37	1608.09	4893.00	497.92	1799.58	5324.97	499.97	1822.39	4714.04	502.78	1724.76	4415.29
	sort	3.38	9.66	30.43	6.61	19.43	59.31	9.16	24.53	67.57	10.08	31.41	81.18
	print	4.13	35.59	50.45	4.16	19.24	48.99	6.03	23.36	94.31	6.42	21.76	68.79
	comp	127.36	326.18	484.74	128.84	295.28	340.15	137.72	319.51	625.98	121.99	280.24	360.99

Figure 6: Response times of `email` tasks, listed from highest to lowest priorities. The `resp` reports the times elapsed between when a client sends a request to when `email` reacts to the request by generating a computation. The `send`, `sort`, and `print` report the respective times elasped between when a client sends the given request to when the corresponding task completes. The `compress` reports the time elapsed to perform a particular compression task. We report the average times (Avg), the 95^{th} percentile (95%), and the 99^{th} percentile (99%) for all categories. The times for `resp` and `send` are in microseconds. The times for `sort` are in microseconds per message. The rest are in milliseconds. Times for I-Cilk were collected with $\rho = 2$, $L = 500\mu s$, and $\delta = 0.9$.

		36 Clients			72 Clients			108 Clients			144 Clients		
		Avg	95%	99%	Avg	95%	99%	Avg	95%	99%	Avg	95%	99%
Cilk-F	resp	103.78	74.61	341.17	231.44	79.61	529.18	343.11	80.69	591.68	414.18	80.69	580.92
	hit	4.94	3.74	14.85	10.26	3.95	22.71	14.92	4.00	25.33	17.87	3.99	24.85
	miss	2.01	2.30	60.18	2.12	3.04	67.79	1.98	1.51	66.05	1.94	1.45	60.14
	stat	0.17	0.38	0.61	0.17	0.38	0.46	0.19	0.25	0.50	0.18	0.27	0.40
I-Cilk	resp	61.06	65.47	241.95	180.40	73.80	386.48	176.21	76.26	450.83	323.13	76.86	504.60
	hit	3.16	3.36	10.72	8.14	3.72	16.75	7.97	3.83	19.44	14.09	3.85	21.70
	miss	2.46	3.14	67.78	3.42	3.50	66.50	4.41	4.36	67.21	3.70	4.28	67.68
	stat	1.05	0.55	36.20	0.11	0.25	0.33	1.31	0.32	46.90	8.24	37.55	345.85

Figure 7: Response times of `proxy` tasks listed from highest to lowest prorities. The `resp` reports the time elapsed between when a client sends a request to when `proxy` reacts to the request. The `hit` reports $\mu s/byte$ (μs is microsecond) for responding with a website already in cache; the `miss` reports $\mu s/byte$ for those not in cache. The `stat` reports $\mu s/cacheSize$ for collecting cache statistics. Times for I-Cilk were collected with $\rho = 1.2$, $L = 500\mu s$, and $\delta = 0.9$.

websites back to the client. A concurrent hashtable is used to cache the websites once it's fetched. There are four priority levels (highest to lowest): a) accepting new client connections and handling requests from clients already connected (and response immediately if the requested site is in already the cache); b) on a cache miss, fetching content from the site and storing it in cache; c) logging statistics about sites requested.

Results from running `proxy` are shown in Figure 7. In general, I-Cilk schedules the tasks such that it favors response time of high priority tasks (hit) over lower priority ones. A ρ value of 1.2 seems to work the best, as the high priority tasks in `proxy` have little parallelism.

Discussion. While it is true that the optimal parameter values will depend on the application characteristics, and the programmer should empirically evaluate a few configurations within the range if she wants the best performance possible, based on our empirical evaluation a few default values can work well based on the observations that follow. First, δ (the efficiency parameter) does not seem to make much difference as long as it's on the high-end of the range (e.g., > 0.75) because its value is used as a threshold to qualify whether a quantum is efficient. Second, for L (quantum length), generally something like 500 microseconds or 1 millisecond works well regardless of application characteristics because it should be long enough to amortize the cost of adaptive scheduling, but short enough that it can react to changes in parallelism within each priority level. For an application that lacks parallelism, lower ρ (responsiveness parameter) values (e.g., 1.2) seems to work well, whereas for an application that has ample parallelism, higher values (e.g., 2) seem to work well.

6 RELATED WORK

Priority Scheduling. Prior work by Muller et al. [17, 18, 20] provided a type system, the corresponding cost semantics, and the principle that a well-formed computation (from a program that type checked) can be scheduled with their stated execution time bounds as long as the scheduler used is prompt. This prior work does not describe a provably efficient online scheduling algorithm, however. Even though the prior work has an implementation, their implementation is not prompt (a strictly prompt scheduler would not be practical due to synchronization overhead), nor do they analyze the implementation to show any formal approximation of promptness.

In this work, we remove the assumption of strict promptness from our theoretical scheduling algorithm. Instead, our theoretical algorithm approximates promptness while being practical to implement and we provide an analysis that explicitly bounds the waste due to this approximation. In other words, our theoretical result provides an online scheduling algorithm with the same asymptotic performance bounds as Muller et al. Our scheduler approximates promptness in the same spirit as how a work-stealing scheduler approximates work conservation.

Beyond work by Muller et al., there is little work in the context of task parallelism. When priorities have been studied in the context of task parallelism, they have been used as heuristics to improve

the throughput of computations (e.g., searches where the ordering of branches can improve performance) [15, 29, 30], rather than for responsiveness.

Priorities are also used for scheduling in Operating Systems (see an overview in [22]) and real-time systems (see a survey in [12]). In OS, the scheduling entities are usually heavy-weight persistent threads (e.g., POSIX threads [16]) and the scheduling quantum lengths are longer. In this work, the scheduling entity is a task, which is much lighter weight with a shorter scheduling quantum length. These differences necessarily lead to different mechanisms for ensuring responsiveness. Moreover, there are seldom any theoretical bounds on response time within OS threads.

In real-time scheduling, priorities are used in a different manner. Typically in a real-time system, the set of jobs is fixed *a priori*, with known period (how quickly a job generates a task), known deadline (when a task must complete), and a known upper bound on the work of a task. These tasks tend to be independent of each other and do not interact (although they could share resources through the use of locks). The scheduler must provide an *a priori* guarantee of meeting each deadline and priorities are used as a scheduling mechanism to meet these deadlines.

Adaptive Scheduling. Our scheduling framework is based on the adaptive scheduling strategy A-GREEDY [3], which is designed for a very different context (scheduling independent parallel jobs in a multiprogrammed environment). A-GREEDY focuses on providing an algorithm for the second-level scheduler to best determine the desired number of cores for a job to request assuming an adversarial top-level scheduler. Without considering the top-level scheduler and how it assigns cores, our second-level scheduler essentially utilizes the same algorithm as A-GREEDY to determine the desired number of cores to request. However, the analysis of our algorithm is very different from that of A-GREEDY due to two key differences. First, we have a top-level scheduler that accounts for priority when making core assignments. Second, in our context, tasks from different priority levels are not independent, whereas A-GREEDY considers independent parallel jobs.

A-GREEDY [3] assumes a second-level greedy scheduler. A similar adaptive scheduling framework for scheduling independent parallel jobs has been developed called A-STEAL [4, 5], where the analysis considers work stealing [7–10] as the second-level scheduler. A similar adaptive scheduling algorithm using work-stealing has also been used to optimize for power and energy for parallel tasks [2]. In our framework, incorporating work stealing into the analysis will be complex since we would need to bound the steal overheads for different priority levels (a possible direction for future work).

7 CONCLUSION

In this paper, we described a practically efficient and theoretically sound algorithm for scheduling task-parallel interactive applications on multicore platforms. Experiments indicate that the algorithm performs well in practice, and has low scheduling overheads and good response times for high-priority tasks even as the load increases. There are several directions of future work. First, our analysis assumes a work-conserving scheduler for every priority level, while our implementation uses an almost work-conserving work-stealing scheduler which performs better in practice. We want to analyze work-stealing directly as part of a two-level scheduling system. Second, the scheduling overhead of our system increases as ℓ (priority level) and k (number of edges between two nodes with different priority levels) parameters of the task increase — it would be nice to design a scheduler which did not depend on these parameters. Finally, the type system we use restricts the kinds of dependences we can have in the task in order to check for priority inversions. We would like to generalize this type system to allow a richer set of dependences.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable feedback. This research was supported in part by the National Science Foundation under grant numbers CCF-140894, CCF-1629444, CCF-1527692, CCF-1725647, CCF-1733873, CCF-1901381, and CCF-1910568.

REFERENCES

[1] Kunal Agrawal, Joseph Devietti, Jeremy T. Fineman, I-Ting Angelina Lee, Robert Utterback, and Changming Xu. 2018. Race Detection and Reachability in Nearly Series-Parallel DAGs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. New Orleans, Louisiana.

[2] K. Agrawal and S. Gilbert. 2018. The Power to Schedule a Parallel Program. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 182–193.

[3] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. 2006. Adaptive Task Scheduling with Parallelism Feedback. In *Proceedings of the Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[4] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. 2006. An Empirical Evaluation of Work Stealing with Parallelism Feedback. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. Lisboa, Portugal.

[5] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. 2007. Adaptive work stealing with parallelism feedback. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. ACM, San Jose, California, USA, 112–120.

[6] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. 2016. Scheduling Parallel DAG Jobs Online to Minimize Average Flow Time. In *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '16)*. Society for Industrial and Applied Mathematics, Arlington, Virginia, 176–189.

[7] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread Scheduling for Multiprogrammed Multiprocessors. In *10th Annual ACM Symposium on Parallel Algorithms and Architectures*. 119–129.

[8] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* (2001), 115–144.

[9] Robert D. Blumofe and Charles E. Leiserson. 1994. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*. 356–368.

[10] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *JACM* 46, 5 (1999), 720–748.

[11] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (April 1974), 201–206.

[12] Robert I. Davis and Alan Burns. 2011. A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *ACM Comput. Surv.* 43, 4, Article 35 (Oct. 2011), 44 pages. https://doi.org/10.1145/1978802.1978814

[13] R. L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* 17, 2 (March 1969), 416–429.

[14] Cilk Hub. 2018. opencilk.org. (2018). Accessed in July 2019.

[15] Shams Imam and Vivek Sarkar. 2015. Load Balancing Prioritized Tasks via Work-Stealing. In *Proceedings of the 21st International European Conference on Parallel and Distributed Computing (Euro-Par '15)*. Vienna, Austria, 222–234.

[16] 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. (1996).

[17] 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.

[18] 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.

[19] 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, to appear.

[20] 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.

[21] 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.

[22] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2013. *Operating System Concepts (9^{th} Edition)*. Wiley.

[23] 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*. Society for Industrial and Applied Mathematics, 147–161.

[24] 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*. ACM, Washington, District of Columbia, 257–271.

[25] 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.

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

[27] Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. 2019. Efficient Race Detection with Futures. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, Washington, District of Columbia, 340–354.

[28] Jacobo Valdes. 1978. *Parsing Flowcharts and Series-Parallel Graphs*. Ph.D. Dissertation. Stanford University. STAN-CS-78-682.

[29] Martin Wimmer, Daniel Cederman, Jesper Larsson Träff, and Philippas Tsigas. 2013. Configurable Strategies for Work-stealing. *CoRR* abs/1305.6474 (2013). arXiv:1305.6474 http://arxiv.org/abs/1305.6474

[30] Martin Wimmer, Daniel Cederman, Jesper Larsson Träff, and Philippas Tsigas. 2013. Work-stealing with Configurable Scheduling Strategies. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, Shenzhen, China, 315–316.