Fairness in Responsive Parallelism

STEFAN K. MULLER, Carnegie Mellon University, USA
SAM WESTRICK, Carnegie Mellon University, USA
UMUT A. ACAR, Carnegie Mellon University, USA and Inria, France

Research on parallel computing has historically revolved around compute-intensive applications drawn from traditional areas such as high-performance computing. With the growing availability and usage of multicore chips, applications of parallel computing now include interactive parallel applications that mix compute-intensive tasks with interaction, e.g., with the user or more generally with the external world. Recent theoretical work on responsive parallelism presents abstract cost models and type systems for ensuring and reasoning about responsiveness and throughput of such interactive parallel programs.

In this paper, we extend prior work by considering a crucial metric: fairness. To express rich interactive parallel programs, we allow programmers to assign priorities to threads and instruct the scheduler to obey a notion of *fairness*. We then propose the *fairly prompt scheduling* principle for executing such programs; the principle specifies the schedule for multithreaded programs on multiple processors. For such schedules, we prove theoretical bounds on the execution and response times of jobs of various priorities. In particular, we bound the amount, i.e., *stretch*, by which a low-priority job can be delayed by higher-priority work. We also present an algorithm designed to approximate the fairly prompt scheduling principle on multicore computers, implement the algorithm by extending the Standard ML language, and present an empirical evaluation.

CCS Concepts: • **Software and its engineering** → **Parallel programming languages**; *Concurrent programming languages*; *Concurrent programming structures*; • **Theory of computation** → Interactive computation.

Additional Key Words and Phrases: Parallelism, Concurrency, Priorities, Cost Semantics, Fairness, Starvation

ACM Reference Format:
Stefan K. Muller, Sam Westrick, and Umut A. Acar. 2019. Fairness in Responsive Parallelism. *Proc. ACM Program. Lang.* 3, ICFP, Article 81 (August 2019), 30 pages. https://doi.org/10.1145/3341685

1 INTRODUCTION

Due to recent advances in hardware, multicore computing has become the rule, rather than the exception, in consumer desktops as well as mobile devices such as tablets, smartphones and even smart watches. These developments have led to increased interest in languages and techniques for writing parallel programs. Many languages, libraries and systems have been developed, including NESL [Blelloch et al. 1994], Cilk [Frigo et al. 1998], Fork/Join Java [Lea 2000], Habanero Java [Imam and Sarkar 2014], TPL [Leijen et al. 2009], TBB [Intel 2011], X10 [Charles et al. 2005], parallel ML [Fluet et al. 2011, 2008; Guatto et al. 2018; Raghunathan et al. 2016; Sivaramakrishnan et al. 2014], and parallel Haskell [Keller et al. 2010; Kuper et al. 2014].

These languages and systems for *cooperative threading* allow the user to express parallelism at an abstract level without directly specifying how to map tasks onto processors, and leave it to a *scheduler* to determine the mapping. The focus of nearly all of this work has been on running a

Authors' addresses: Stefan K. Muller, Carnegie Mellon University, USA, smuller@cs.cmu.edu; Sam Westrick, Carnegie Mellon University, USA, swestric@cs.cmu.edu; Umut A. Acar, Carnegie Mellon University, USA, Inria, France, umut@cs.cmu.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/8-ART81
https://doi.org/10.1145/3341685

compute-intensive job on the cores of a machine as quickly as possible, i.e., maximizing *throughput*. This focus on compute-intensive workloads is partially historical, owing to the traditional use of parallelism in high-performance computing.

Another line of work, largely in the systems community, develops techniques for *competitive threading*. Systems and applications for competitive threading focus on hiding the latency of blocking operations by multiplexing independent threads. This multiplexing is useful even on a uni-processor, especially in interactive applications (e.g., [Blake et al. 2010; Flautner et al. 2000; Gao et al. 2014; Hauser et al. 1993]). Key performance metrics for competitive threading include *responsiveness*—how long each thread takes to respond to a user—and *fairness*—whether the threads receive fair distributions of CPU time (this may mean giving them equal time or dividing CPU time according to the weight or priority of threads).

While each of these two approaches to threading have been researched extensively, they have historically remained as two largely separate lines of research, with competitive threading confined mostly to the systems community and cooperative threading to the parallelism community. The advances in multicore computers challenge this order, because the workloads of multicore and parallel hardware today do not consist of purely computational or purely interactive workloads but also include applications that are interactive and involve compute-intensive parallel tasks. For example, a game might perform compute-intensive tasks to implement visual effects while it also interacts with the user. Recent work by Muller et al. [2017, 2018] has developed programming-language theory for such interactive parallel applications by considering languages that combine interaction and parallelism, and by allowing parallel tasks to have different priorities. Their work allows, for example, a parallel compute-intensive task to run in the "background" concurrently with a high-priority interactive task. They introduce a scheduling principle called *prompt scheduling* that can be used to reason about the response time and throughput of such computations and provide an operational semantics that implements this principle.

The prior work on responsive parallelism [Muller et al. 2017, 2018] shows that it is theoretically possible to achieve both responsiveness and throughput. In this paper, we build on that work to account for *fairness* while maintaining provable guarantees on response time by presenting *fairly prompt scheduling*. Fairness has been the subject of a large body of work in the systems community (e.g. [Demers et al. 1989; Goyal et al. 1996; Waldspurger and Weihl 1994]), where the focus has been on coarse-grained threads and largely excluded parallel workloads. Fairness continues to be challenging to achieve even in production operating systems, especially in the presence of both interactive and computational workloads [Nieh et al. 1994], which have essentially opposing demands from the schedulers.

As an example of a highly parallel application where fairness plays a crucial role, consider a robot path planning system which runs two or more parallel path planning algorithms in a hierarchical fashion [Knepper et al. 2010]. A high-level planner runs infrequently to find an efficient path using a set of waypoints (e.g. which hallways to use to go between rooms in a building) while a low-level planner runs frequently to plan the detailed path to the next waypoint, which may involve navigating around obstacles such as furniture. Each planning algorithm itself is parallel, and so many planning threads must share resources with a high-priority interaction loop that periodically reads sensor values and outputs commands to motors. Such an application could not be encoded in the priority model of prompt scheduling because the two planning computations could starve each other. The application could be coded directly using low-level threading libraries which provide both priorities and fairness, but this would require manually writing complex concurrent code and provide no guarantees on responsiveness.

Intuitively, the guarantee a programmer would desire from a fair scheduler would be a bound on how much the execution time of a low-priority computation can be impacted by higher-priority work in the system. We refer to this increase in execution time as a computation's *stretch*: if a computation takes 4 times longer to execute than it would in isolation, its stretch is 4. Using techniques presented in this paper, the motion planning example above can be programmed easily as three parallel computations (two planners and an interaction loop) running at three priorities (see Section 6.3.1 for specific details of our implementation of the example). If the high- and low-level planners are equally computationally intensive and we wish for the low-level planner to run at 4 times the rate of the high-level planner, the programmer might specify that the high-level planner can tolerate a stretch of 6 while the low-level planner can only tolerate a stretch of 1.5 (both must be stretched somewhat to allow for the other planner and the interaction loop to run).

In this paper, we will more formally define mechanisms for specifying fairness that allow programmers to reason about and appropriately tune the maximum amount of stretch experienced by low-priority computations. We show that this reasoning is sound for all schedulers that follow a scheduling principle we call "fairly prompt". Fairly prompt scheduling extends the prompt scheduling of prior work by also requiring schedulers to devote enough processing time to lower-priority work to achieve the desired fairness (or stretch). Prior systems based on prompt scheduling, in which higher-priority jobs are always executed, cannot guarantee any finite bound on stretch: a series of long-running high-priority jobs could delay a low-priority computation indefinitely.

In addition to presenting and bounding fairly prompt scheduling, we also describe a concrete algorithm based on this principle. We evaluate the algorithm by implementing it as an extension of Standard ML and considering a suite of benchmarks. We also provide a qualitative analysis by considering two larger case studies, including one based on the motion planning example above.

The contributions of this paper include the following.

- The fairly prompt scheduling principle for scheduling interactive parallel programs.
- An offline scheduling theorem showing that an abstract scheduler obeying the fairly prompt principle respects an upper bound on the execution and response time of threads.
- A scheduling algorithm that implements the key ideas of fairly prompt scheduling while also considering practical concerns.
- An implementation of the scheduler and a threading library as an extension to Standard ML.
- A suite of parallel interactive benchmarks and an empirical evaluation.

2 PRELIMINARIES

In this section, we give a brief overview of the threading model considered in this paper, and then review related definitions and notations on which we will build. This section does not discuss extending the models with fairness; that is the subject of Section 3.

2.1 Threading Model

In this paper, we assume a program consists of threads, each of which is assigned a priority ρ. Priorities are drawn from a set R. Priorities are ordered, i.e. we may write $\rho_1 \prec \rho_2$ to indicate that a thread of priority ρ_1 is lower-priority than a thread of priority ρ_2. We write $\rho_1 \preceq \rho_2$ if ρ_1 and ρ_2 may be the same. The order $\preceq$ may be total or partial, i.e., it is not necessary to specify the full ordering of thread priorities.

Threads may depend on each other in two ways, through *spawns* and *joins*. A thread a may *spawn* another thread b; thereafter, thread b will run in parallel with the remainder of thread a. A thread a may also *join* with thread b; the join operation will wait for thread b to complete before thread a proceeds. This model is sufficient to encode many threading models, such as fork-join, async-finish and futures, which are used in real-world parallelism libraries. In particular, the formal model and scheduling algorithm presented in this paper could be considered a blueprint for extending the

PriML language [Muller et al. 2018] with fairness. In PriML, programs are parallelized using a threading construct similar to futures. Each future is assigned a priority. The set of priorities and their ordering (which may be a partial order) is specified by the programmer. However, PriML, as it currently exists, does not support any notion of fairness, scheduling threads strictly in priority order. The threading library we describe in Section 5.3, and the example programs we implemented it (Section 6) are inspired by PriML, but the ideas of this paper are more general and can apply to any language and threading model that is compatible with the assumptions described above.

2.2 A DAG Model for Priorities

In the formal development of this paper, as is standard for such developments, we represent a parallel computation using a directed acyclic graph, or DAG. In such a model, vertices of the DAG represent sequential operations taking a uniform amount of time (e.g. processor instructions) and edges represent dependences between operations. Our model and related notations are built on those of Muller et al. [2018], which we now briefly review.

Formally, a DAG g is a triple $(\mathcal{T}, E_j, E_s)$. The first component is a collection of *threads*, each of which is associated with a thread symbol, for which we use the metavariables a, b, c and variants, and with a priority ρ. An element of $\mathcal{T}$ is written $a \underset{\rho}{\hookrightarrow} \vec{u}$. The thread itself, written $\vec{u}$, is a sequence of vertices $\vec{u} = u_1 \cdot u_2 \cdot u_3 \cdot \ldots \cdot u_n$, all of which must occur in sequence. We write $Prio_g(u)$ to refer to the priority of the thread in g to which u belongs.

For each thread $u_1 \cdot \ldots \cdot u_n$, there exist edges $(u_1, u_2), \ldots, (u_{n-1}, u_n)$ to indicate the sequential dependences between instructions in the thread. Inter-thread dependences are indicated with additional edges, which are collected in the second and third components of the DAG triple. The set E_j contains edges of the form (a, u) indicating that vertex u waits for thread a to complete. This is a shorthand for the edge (t, u) where t is the last vertex of thread a. The set E_s contains edges of the form (u, a) indicating that vertex u spawns thread a. This is a shorthand for the edge (u, s) where s is the first vertex of thread a.

For two vertices u and u', we write $u \sqsupseteq u'$ if u is an ancestor of u' in a graph, meaning u must execute before u'. If neither $u \sqsupseteq u'$ nor $u' \sqsupseteq u$ then u and u' may execute in parallel.

Scheduling results for responsiveness, in this paper and in prior work, quantify over *well-formed* DAGs. Intuitively, well-formed DAGs do not have *priority inversions*: threads at one priority do not depend on lower-priority work along their critical path. Such a priority inversion would mean that the execution time of a high-priority thread might depend on the amount of lower-priority work, breaking the efficient scheduling guarantees later in this section.

Definition 1. A DAG $g = (\mathcal{T}, E_s, E_j)$ is *well-formed* if for all threads $a \underset{\rho}{\hookrightarrow} u_1 \cdot \ldots \cdot u_n \in \mathcal{T}$, if $u \sqsupseteq u_n$ and $u \not\sqsupseteq u_1$ then $\rho \preceq Prio_g(u)$.

Exactly how to ensure a DAG is well-formed is outside of the scope of this paper. PriML is equipped with a type system that rules out priority inversions, ensuring well-typed programs will produce well-formed DAGs. The threading library we describe in Section 5.3 detects priority inversions at runtime and throws an exception.

Given a number P of processors, a *schedule* is an assignment of vertices to processors over a sequence of time steps such that no vertex is executed before any of its ancestors (we say a vertex is *ready* if all of its ancestors have been executed). The *offline scheduling problem* for a DAG is to construct a short schedule for the DAG. Constructing the optimal schedule is NP-hard in general [Ullman 1975]. As a result, offline scheduling results in the literature generally show that schedules that obey a certain *scheduling principle* come within a constant factor of optimal.

Probably the most famous such result is Brent's Theorem [Brent 1974] which gives a schedule that is 2-optimal for standard DAGs without priorities or responsiveness requirements.

Muller et al. [2017, 2018] proposed the *prompt* scheduling principle, which extends classic ideas like Brent's to prioritized DAGs. A prompt schedule is one that does not execute a thread if there is a higher-priority thread that is ready and not already being executed. Furthermore, it does not leave a processor idle if there remains a ready thread not being executed.

In a prompt schedule of a well-formed DAG, the response time of a thread of priority ρ can be given a bound that does not depend on the amount of work at priorities lower than ρ. Formally, let $a \underset{\rho}{\hookrightarrow} s \cdot \ldots \cdot t$ be a thread. The response time $T(a)$ of a is the number of time steps between when s becomes ready and when t is executed, inclusive. We write W_R to refer to the number of vertices of priorities belonging to a set R. The *competitor work* $W_{\not\prec\rho}(\updownarrow a)$ refers to the number of vertices at priority at least ρ that may occur in parallel with vertices of thread a:

$$W_{\not\prec\rho}(\updownarrow a) \triangleq |\{u \in g \mid u \not\sqsupseteq s \wedge t \not\sqsupseteq u \wedge Prio_g(u) \not\prec \rho\}|$$

The *a-span* $S_a(\updownarrow a)$ is the length of the longest path ending at t consisting of non-ancestors of s. Under these definitions and assumptions, the response time of a thread can be bounded by terms that include only competitors. Intuitively, at each step during which thread a is active, we are either executing P vertices at a's priority or higher, or we are making progress on the a-span.

Theorem 1 (Muller-Acar-Harper). *Let g be a well-formed DAG with a thread $a \underset{\rho}{\hookrightarrow} \vec{u} \in g$. For any prompt schedule of g on P processors,*

$$T(a) \leq \frac{W_{\not\prec\rho}(\updownarrow a)}{P} + S_a(\updownarrow a)$$

3 SCHEDULING WITH FAIRNESS

3.1 The Fairness Criterion

In the introduction, we motivated the idea of fairness using the notion of *stretch*: one may wish to specify that, for example, a computation at a low priority should take no more than four times as long to run as it would if that computation had the machine to itself. In our scheduling principle and the design of our algorithm, we will accomplish the desired stretch by allotting processor resources to priorities according to the stretch specified by the programmer: if a priority ρ is to have stretch $s(\rho)$, the scheduler will ensure that threads at that priority, when available to run, get at least $1/s(\rho)$ of the compute time. Of course, this means that in order for a programmer-defined set of stretch constraints to be feasible, it must be the case that $\Sigma_{\rho \in R} \frac{1}{s(\rho)} \leq 1$. This motivates the concept of the *fairness criterion*, which indicates the fraction of computing resources allotted to each priority. A fairness criterion F is a mapping from priorities to real numbers in the range $[0, 1]$, summing to 1. A fairness criterion F states that computations at a priority ρ should be given the opportunity to use at least a fraction $F(\rho)$ of the available processing time. The reader may note that, as defined above, fairness criteria happen to be discrete probability distributions over priorities, a fact we will take advantage of later when defining a fair scheduling principle.

The fairness criterion is a convenient formalization of fairness for designing the scheduling principle and scheduling algorithm, and so most of the rest of the paper will be framed in terms of the fairness criterion rather than the more intuitive notion of stretch. We will, however, show in Section 3.3 that the connection drawn between the two above is valid: under any fairly prompt scheduler and a fairness criterion F, a computation at priority ρ will experience a stretch of at most $\frac{1}{F(\rho)}$.

3.2 A Fair and Prompt Scheduling Principle

We introduce a new scheduling principle called the *fairly prompt* scheduling principle, which extends the prompt scheduling principle with the notion of fairness criteria developed above. A fairly prompt scheduler is parametrized by a fairness criterion F and is defined by two properties:

- When threads of *all priorities* are present in the system, the scheduler should devote, on average, the fraction $F(\rho)$ of processor cycles to threads at priority ρ.

This property is sufficient to ensure that the execution time of a job at priority ρ is not stretched by more than a factor of $1/F(\rho)$ due to higher-priority jobs, but leaves open the question of what to do when a particular priority is unavailable (i.e., has no threads available in the system). This leads to the second property:

- When a chosen priority is unavailable, this priority should "donate" its unused CPU cycles to the highest available priority [1].

We note that donation of cycles slightly complicates the intuitive relationship between fairness criterion, as described, and stretch. Consider a program with the three priorities (H)igh, (M)edium and (L)ow, with the following fairness criterion.

$$F(\rho) = \begin{cases} 0.7 & \rho = H \\ 0.2 & \rho = M \\ 0.1 & \rho = L \end{cases}$$

Suppose that, for a time, there are no threads of priority H in the system. Under fairly prompt scheduling, all of H's cycles will be donated to M, resulting in a 0.9/0.1 distribution of resources between M and L. If one interprets the fairness criterion F to mean "M should receive twice as many cycles as L", this distribution might seem unfair. However, we consider the goal of our principle to be maintaining *stretch* rather than the relative speeds of different threads, and so the fairness criterion F should instead be read to mean "a computation at M should stretch no more than 5× and a computation at L should stretch no more than 10×. This guarantee *is* upheld by our system. Note that this guarantee would be upheld even if cycles were not donated from inactive priorities; donations can only improve the stretch of threads (in this case, a computation at M experiences a much higher level of service than it expects).

The *fairly prompt* scheduling policy above is flexible enough to encode many application-specific scheduling policies. For example, if $\top$ is the highest priority, we can encode a form of prompt scheduling in which as many processors as possible are devoted to the highest-priority work available, followed by the next-highest, and so on. The fairness criterion for such a policy would be $F(\top) = 1$ and $F(\rho) = 0$ for all $\rho \neq \top$.

For concreteness, we will describe how to construct a fairly prompt schedule given a sequence of time steps and a set of P processors. At each time step, each processor is assigned a priority probabilistically according to the fairness criterion F (which, we have noted earlier, may be thought of as a probability distribution). Processors attempt to execute a ready vertex at their assigned priority. Processors that are unable to execute a vertex at their assigned priority default to the "prompt" policy and execute the highest-priority ready vertex.

The use of probability in the definitions of fairness and fairly prompt schedules merits further discussion. Formal definitions of fairness, for example in the model checking community (e.g., [Francez 1986]), are often stated as properties of infinite executions (e.g., that every process executes infinitely many times over an infinite interval). The notion of fairness described here diverges from such

[1] If priorities form a partial, rather than total, order, "highest available priority" could be interpreted as "any priority ρ such that no priority higher than ρ has available work". In practice, we implement this by performing a topological sort to derive a total order on priorities consistent with the partial order, and using the total order for scheduling purposes.

definitions in two important ways. First, the number of processes and (in general) the lengths of schedules are finite. Second, our notion of fairness places requirements on the relative frequency at which threads execute, as opposed to simply requiring that they execute eventually. Consider the fairness criterion defined above for the priorities $\{H, M, L\}$. For a program that runs for 12 time steps on 2 processors, it is impossible to exactly meet such a fairness criterion because 0.1×24 is not an integral number of processor-steps. Given a finite, discrete number of processors and time steps, we could guarantee that the fairness criterion is met *to within some approximation*, but we instead choose to introduce probabilistic reasoning and to guarantee that the fairness criterion is met exactly *in expectation.* As the number of processors or time steps approaches ∞, we would intuitively expect the distribution of work to converge to the fairness criterion.

3.3 Bounding Response Time

We now bound the response time of threads in fairly prompt schedules. We will do this using two theorems. Theorem 2 bounds the response times of threads in terms of their share of the fairness criterion, that is, it bounds the amount by which a job is "stretched", justifying the intuitive connection between stretch and the fairness criterion. This theorem gives a straightforward bound that does not consider the additional CPU cycles that may be "donated" from other priorities; this makes it useful for thinking about threads at lower priorities that are less likely to receive such donations. Next we show another result, Theorem 3, which shows how the response times of higher-priority threads can benefit from donated cycles.

In showing both results, we need not develop new cost metrics because, intuitively, fairness doesn't cause high-priority threads to depend on any work they otherwise wouldn't. Fairness only "inflates" the response time bounds to account for the fraction of cycles devoted to lower-priority work according to the fairness criterion. We introduce a convenient shorthand for summing the fairness criterion over a set of priorities:

$$F(R') \quad \triangleq \quad \sum_{\rho \in R'} F(\rho)$$

We will also use the notation $\not\prec \rho$ to denote the set $\{\rho' \in R \mid \rho' \not\prec \rho\}$.

We also note that the bounds in this section are given as expected values. Since fairly prompt schedules are defined probabilistically, their analysis is also probabilistic and it is not possible to give exact bounds.

First, we show that the response time of a thread at priority ρ is "stretched" by no more than $1/F(\rho)$ relative to its expected execution time if the system contained only threads at priority ρ. The intuition behind this result is that, from the perspective of a thread (or collection of threads) at one priority ρ, a fairly prompt schedule on P processors appears the same as a greedy schedule (one that always executes ready threads when possible) on $P \cdot F(\rho)$ processors. In this way, the result is very much like the offline scheduling bound of Arora et al. [2001], which concerns *multiprogrammed environments* in which only a fraction of the total processors may be available to a parallel computation at any given step. Our bound must include the thread's competitor work at priority ρ, but otherwise the bound and proof are quite similar to those of Arora et al.

Theorem 2. *Let g be a well-formed DAG. Let $a \underset{\rho}{\hookrightarrow} s \cdot \ldots \cdot t \in g$ be such that for all $u \in g$ where $u \sqsupseteq t$ and $u \not\sqsupseteq s$, it is the case that $\mathit{Prio}_g(u) = \rho$. For any fairly prompt schedule on P processors,*

$$E[T(a)] < \frac{1}{F(\rho)} \frac{1}{P} \left(W_{\{\rho\}}(\ddagger\, a) + S_a(\ddagger\, a) \cdot (P - 1) \right) + 1$$

Proof. Consider the portion of the schedule from the step in which s becomes ready (exclusive) to the step in which t is executed (inclusive). At each step, each processor samples the fairness criterion independently, so we may model each step as P “processor-steps” happening in sequence, in which one processor samples the fairness criterion and attempts to execute a vertex. At each processor-step, collect a token from the relevant processor *if it is assigned to priority* ρ at that step. If the processor successfully works on a vertex of priority ρ at that step, place a token in the “work” bucket; otherwise, place a token in the “idle” bucket. Let B_w and B_l be the numbers of tokens in the work and idle buckets, respectively, immediately after a is executed. We will bound B_w and B_l separately. Each token in B_w corresponds to work done at priority ρ, so $B_w \leq W_{\{\rho\}}(‡\, a)$.

We now bound B_l by $S_a(‡\, a) \cdot (P-1)$. For this part of the proof, we work in whole steps, not processor-steps. Let step 0 be the step after s is ready, and let $Exec(j)$ be the set of vertices that have been executed at the start of step j. Consider a step j in which a token is added to B_l. For any path ending at t consisting of vertices of $g \setminus Exec(j)$, the path starts at a vertex of priority ρ that is ready at the beginning of step j. By the fair principle, this vertex must be executed in step j, so the length of the path decreases by 1 and so $S_a(g \setminus Exec(j+1)) = S_a(g \setminus Exec(j)) - 1$. Therefore, the maximum number of steps in which B_l increases is $S_a(g \setminus Exec(0))$, and so $B_l \leq S_a(g \setminus Exec(0)) \cdot (P-1)$ since at most $P-1$ processors can be idle while thread a is active. Let $⇞s \triangleq g \setminus \{u \neq s \mid u \sqsupseteq s\}$. Because $⇞s \supset g \setminus Exec(0)$, any path excluding vertices in $Exec(0)$ is contained in $⇞\, s$, and $S_a(g \setminus Exec(0)) \leq S_a(⇞\, s)$, so $B_l \leq S_a(⇞s) \cdot (P-1) = S_a(‡\, a) \cdot (P-1)$.

Let $B = B_w + B_l$. By construction, once B tokens have been placed, a has finished executing. Let T_{ps} be a random variable representing the number of processor-steps required to place B tokens. Whether a token is placed at each processor-step can be modeled by a geometric distribution with $p = F(\rho)$, and so $E[T_{ps}] = \frac{B}{F(\rho)}$. The number of *steps* to place B tokens is $\left\lceil \frac{T_{ps}}{P} \right\rceil < \frac{T_{ps}}{P} + 1$. By monotonicity and linearity of expectation we have

$$E[T(a)] < \frac{E[T_{ps}]}{P} + 1 = \frac{B}{P \cdot F(\rho)} + 1 \leq \frac{1}{P \cdot F(\rho)} \Big(W_{\{\rho\}}(‡\, a) + S_a(‡\, a) \cdot (P-1) \Big) + 1$$

□

The above result considers a collection of threads at a single priority in isolation. This result is useful as an upper bound on the execution time of long-running computations because, as long as work at that priority is always available, these threads are guaranteed to receive at least a fraction $F(\rho)$ of the processor cycles. The result is less useful for jobs that may idle, and therefore give up some of their CPU time, but also benefit from donations from other priorities. Theorem 3 bounds the expected response time of such a thread a at priority ρ. Thread a may run on cycles devoted to priority ρ, but it may also run on cycles devoted to higher priorities if work at those priorities is unavailable. Thus, consider the processor cycles which are intended to be devoted to priorities $\rho' \not\prec \rho$. At each such cycle, if there are any threads ready at a priority not less than ρ, one will be executed on this cycle: either the processor will choose a thread at priority ρ' if one is available, or if not, it will execute the highest-priority thread available which, by assumption, has priority not less than ρ. Thus, for cycles in this category, the schedule is greedy with respect to work at priority not less than ρ. In expectation, a fraction $F(\not\prec \rho)$ of cycles fall into this category. So the response time for thread a is simply inflated to account for the fact that only $\frac{1}{F(\not\prec \rho)}$ of cycles are being spent on this work in expectation.

Theorem 3. *Let $g \ni a \underset{\rho}{\hookrightarrow} \vec{u}$ be a well-formed DAG. For any fairly prompt schedule on P processors,*

$$E[T(a)] < \frac{1}{F(\not\prec \rho)} \left(\frac{W_{\not\prec \rho}(‡\, a)}{P} + S_a(‡\, a) \right) + 1$$

Proof. Let s and t be the first and last vertices of a, respectively. Consider the portion of the schedule from the step in which s becomes ready (exclusive) to the step in which t is executed (inclusive). At each step, collect a token from every processor *attempting* to work on vertices of priorities in $\not\prec \rho$. If the processor is attempting to work at a priority not less than ρ, but is unable to, place a token in the "low" bucket. If it attempting to work at a priority not less than ρ and succeeds, place a token in the "high" bucket. Let B_l and B_h be the final number of tokens in the low and high buckets, respectively, when a finishes executing. Each token in B_h corresponds to work done at priority not less than ρ, and thus $B_h \leq W_{\not\prec\rho}(\ddagger a)$. We may bound B_l by $P \cdot S_a(\ddagger a)$ using a technique similar to that in the proof of Theorem 2 (and identical to that of Theorem 4 of [Muller et al. 2018]); we omit the details.

The argument that $E[T(a)] < \frac{1}{P \cdot F(\not\prec\rho)}(B_h + B_l) + 1$ proceeds as in the proof of Theorem 2. □

4 ALGORITHM

The fairly prompt scheduling principle described in Section 3 does not immediately lead to a practical algorithm for scheduling threads: its analysis assumes that the DAG is known before execution (which is not feasible in practice) and making scheduling decisions globally requires high synchronization overhead. In this section, we present an algorithm that is designed around the fairly prompt principle's guidelines for ensuring fairness and promptness: processors should choose priorities according to the fairness criterion, attempt to work at the chosen priority if it is available and otherwise work at the highest available priority. Dividing processor time according to the fairness criterion ensures that each priority will have enough time to meet its desired stretch and donating unused cycles to the highest available priority ensures that high-priority interactive tasks receive the best possible service without harming fairness. To achieve this combination of fairness and promptness in practice, we divide the run of a program into *rounds*. At the beginning of each round, the processor picks a *primary priority* by sampling the fairness criterion. The scheduler attempts to work on threads at the primary priority until a specified interval, called the scheduling *quantum*, has passed. At this point, the processor chooses a new primary priority, thus beginning a new round of scheduling.

To meet the promptness requirement, when a processor's primary priority is unavailable, it temporarily defaults to working on threads of the highest locally available priority. We use the term *current priority* to refer to the priority at which a processor is currently working, which may differ from the primary priority if it is donating time.

Once a processor's current priority has been established, in order to be fairly prompt, the scheduler must work on a thread at the current priority if one is available. This requirement is typically known as the *greedy* principle and is vital to ensuring that the desired stretch of a computation is obeyed: if a thread is available at a processor's current priority but the processor is for some reason unaware of it and mistakenly donates a cycle, the task's stretch increases. Achieving this in practice while keeping scheduling overhead low requires ensuring that threads at each priority are balanced across processors. For this task, we use ideas from the *sender-initiated private deques* variant of *work stealing* [Acar et al. 2013]. In this algorithm, each processor executes threads which are stored in a processor-local set. Work stealing algorithms migrate tasks from busy processors to idle ones. The algorithm we use is *sender-initiated* in that processors periodically attempt to send extra threads to processors which are idle.

4.1 Parameters and Definitions

Before we give a more detailed description of the scheduling algorithm, we introduce some notations and definitions used in the description. In particular, the way threads are treated in the algorithm deserves additional explanation.

```
type bank
insert : bank * thread -> unit
remove : bank -> thread
split : bank -> thread_set
```

Fig. 1. The thread bank interface.

Threads. The units of work handled by the scheduler are *threads*. Each thread has an associated priority, which is accessible with the function `priority`. Threads may be executed with the function `execute`, which runs the thread until it terminates, spawns, or suspends. Executing a thread returns a set of zero, one, or two threads which have become enabled (zero if the thread terminates or suspends waiting for an unavailable resource, one if it can be rescheduled immediately, and two if it spawns a new thread).

In the description of the algorithm, we assume that threads return within a short, bounded amount of time. In practice, we enforce this with periodic interrupts (described in more detail in Section 5.2).

Parameters. The scheduler is parametrized by the number of processors, P; the number of priorities, R; the length of a round, `roundQuantum`; the time interval between work sending ("dealing") attempts, `dealInterval`; and a fairness criterion, `fc` which is a probability distribution over priorities as described in Section 3.

Processor-local state. Each processor has access to several local variables which store the state of the scheduler at that processor. The variable `nextRound` is used to record the time at which the processor should switch to a new primary priority. The variable `nextDeal` records the time at which the processor should next attempt to deal threads to another processor. The variable `primaryPriority` records the primary priority of the current round, and `currentPriority` records the priority at which the processor is actually working. Each processor maintains a set `ioThreads` of I/O-blocked threads which have resumed since the last iteration of the scheduling loop. We will not discuss the mechanism by which these sets are populated, but we follow the method described in [Muller and Acar 2016].

Finally, each processor maintains local collections of threads which we call *thread banks*. The interface for thread banks is shown in Figure 1. Each processor uses R thread banks, one for each priority. When new threads are spawned, they are inserted into the appropriate bank. Threads are removed from banks to be executed locally. When a processor deals work to another processor, it splits its bank and sends the resulting set of threads.

4.2 The Scheduler Loop

The core of the scheduling algorithm is the scheduling loop. The loop runs on each processor and, on each iteration, selects a thread to run and runs it until it terminates. Note that an iteration of the scheduling loop is different from a round: a single round will likely encompass many iterations of the loop.

Pseudocode for the scheduling loop of a single processor is shown in Figure 2. On each iteration of the loop, the processor performs the following.

(1) If it is time to start a new round, draw a new primary priority and set the next switch time (lines 18-20).
(2) Handle any threads previously suspended on I/O operations which have resumed since the last iteration (lines 21-24).

```
 1 // Parameters
 2 int P // number of processors
 3 int R // number of priorities
 4 time roundQuantum
 5 time dealInterval
 6 fairness_criterion fc
 7
 8 scheduleLoop(p): // p = identifier of processor
 9   // Processor-local state
10   bank banks[R]
11   time nextRound
12   time nextDeal
13   priority primaryPriority
14   priority currentPriority
15   thread_set ioThreads
16
17   repeat:
18     if now() > nextRound:
19       nextRound := now() + roundQuantum
20       primaryPriority := fc.random()
21     for t in ioThreads:
22       insert(banks[priority(t)], t)
23       refuseWork(p, banks, priority(t))
24     ioThreads := {}
25     if now() > nextDeal:
26       nextDeal := now() + dealInterval
27       dealAttempt(p, banks, currentPriority)
28     t := getWork(p, banks, primaryPriority)
29     if t = NULL:
30       t := getWork(p, banks, findHighestPrio(banks))
31     if t != NULL:
32       currentPriority := priority(t)
33       enabled := execute(t)
34       for t' in enabled:
35         insert(banks[priority(t')], t')
36         refuseWork(p, banks, priority(t'))
```

Fig. 2. Scheduler loop pseudocode.

(3) Attempt a deal if it is time to do so (lines 25-27).
(4) Attempt to get work at the primary priority (line 28).
(5) If the previous step was unsuccessful, determine the highest locally available priority and get work at it (lines 29-30).
(6) If a thread is found, execute it and handle any new threads it enables (lines 31-36).

Whenever new work at a priority `r` is added to the thread banks (steps 2 and 6 above), `refuseWork` is called to signal that this processor has work at priority `r` so that another processor will not attempt to send it work at this priority.

```
//Global (shared) state
mailbox mailboxes[P, R]

dealAttempt(p, banks, prio):
  q := select random processor
  m := mailboxes[q, prio]
  if tryClaim(m):
    send(m, split(banks[prio]))

refuseWork(p, banks, prio):
  threads := close(mailboxes[p, prio])
  for t in threads:
    insert(banks[prio], t)
```

Fig. 3. Load balancing operations.

```
type mailbox
tryClaim : mailbox -> bool
send : mailbox * thread_set -> unit
open : mailbox -> unit
close : mailbox -> thread_set
```

Fig. 4. The mailbox interface.

```
getWork(p, banks, prio):
  refuseWork(p, banks, prio)
  t := remove(banks[prio])
  if t = NULL:
    open(mailboxes[p, prio])
    return t
```

Fig. 5. Pseudocode for retrieving local threads.

Two main elements of the scheduling algorithm, and several auxiliary functions called but not defined in the pseudocode, require further discussion. First, we describe the load balancing strategy, including the implementations of `dealAttempt` and `refuseWork`. Next, we detail how processors find work locally, including the implementations of `getWork` and `findHighestPrio`.

4.3 Load Balancing and Mailboxes

In order to accomplish load balancing, we require busy processors to periodically attempt to *deal* (send) threads to other processors. Pseudocode for the `dealAttempt` function is shown in Figure 3. When attempting a deal, a processor p picks another processor q at random and checks if q is accepting threads at p's current priority. If so, it splits its bank of that priority and sends the resulting threads to processor q. Sending and receiving threads is facilitated by $P \cdot R$ shared mailboxes, where each processor has one mailbox for each priority. The interface for mailboxes is shown in Figure 4. Each mailbox is either closed (unavailable to receive threads), open (available to receive threads), claimed (waiting to receive threads), or full (holding a set of threads). Before sending threads to a mailbox, the mailbox must first be successfully claimed, which only succeeds if the mailbox is open and not already claimed. This guarantees that if multiple processors attempt to concurrently send threads to the same mailbox that only one succeeds.

The function `refuseWork`, defined in Figure 3, is used by a processor to (a) check for threads dealt by other processors, and (b) signal that the processor is no longer accepting work at a particular priority. Refusing work is accomplished by closing the mailbox and adding any returned threads to the appropriate thread bank.

4.4 Thread Banks and Local Scheduling

The main work of a processor in our algorithm is retrieving threads from the local thread bank at the current priority, executing them and returning newly spawned threads to the appropriate thread banks. Once a priority is selected, the task of finding work at that priority is done by the auxiliary function `getWork`, shown in Figure 5. The function uses `refuseWork` to retrieve work (if any) at that priority from the appropriate mailbox, and add it to the thread bank. Next, a thread from the thread bank is selected with `remove` and returned, to be executed by the current processor. If no such thread is available, the function re-opens the mailbox and returns `NULL`.

Selection of threads in `remove` *and* `split`. The choice of which threads to execute locally and which to send to other processors can have a substantial impact on performance. For scheduling algorithms based on work stealing, it is advantageous to prefer working locally on "young" (recently spawned) threads, thus leaving "old" threads available to be sent for load-balancing. This preference amortizes the cost of sending work (because older threads tend to transfer larger amounts of work) and utilizes existing data locality amongst threads [Acar et al. 2002, 2013; Arora et al. 2001; Blumofe and Leiserson 1999]. We therefore specify that `remove` returns the youngest thread, and `split` returns a set of oldest threads in the bank.

In Section 5.2, we describe in more detail how our implementation tracks the age of threads and efficiently ensures that appropriate threads are chosen. In particular, we do not implement thread banks with standard double-ended queues, or *deques*. Deques rely upon LIFO behavior at the bottom (young end) of the deque in order to maintain the age relationship of threads efficiently. However, our algorithm has two cases in which a thread inserted into a thread bank has no guaranteed age relationship with the existing threads in the bank: (1) when a thread spawns work at a different priority than its own, and (2) when a previously I/O-blocked thread resumes. In both cases, a general insertion (not just a "push") is required.

Finding the highest available priority. One final detail is the implementation of the function `findHighestPrio` which returns the highest priority at which the corresponding bank is not empty. Note that this is entirely a local notion: the function `findHighestPrio` does not search for the highest available priority across all processors, but rather only looks for the highest available priority amongst the banks stored locally at the current processor.

A naïve implementation of `findHighestPrio` could simply inspect every bank, beginning with the highest priority used in the program. We describe a more optimized implementation in Section 5.2.

4.5 Relationship Between the Scheduling Principle and Algorithm

In designing the algorithm of this section, we were guided by the fairly prompt scheduling principle of Section 3. Our goal was to develop a scheduler that, in practice, approximates the responsiveness guarantees of an offline fairly prompt schedule. We show this through the implementation in Section 5 and the empirical evaluation of Section 6. We leave to future work a proof of the specific guarantees of the algorithm. Instead, in the rest of this section, we describe, at the level of intuition, why we would expect our scheduling algorithm to meet the fairly prompt scheduling bounds in common cases. We also briefly study a case in which the algorithm's behavior might not match the theoretical bound.

To understand the behavior of the algorithm at a high level, assume that the number of threads at each priority significantly exceeds the number of processors. Under this assumption, we may model the system as $|R|$ separate instances of work stealing running in a time-sharing fashion according to the fairness criterion. For intuition as to why such a system should achieve fairness and responsiveness, consider the important result of Arora et al. [2001] that shows that if a randomized work stealing scheduler runs a computation with work W and span S concurrently with other processes on a machine with P processors and is given an average of P_A processors by the operating system, it runs in time $O\left(\frac{W}{P_A} + \frac{SP}{P_A}\right)$. This result suggests that the instance of work stealing for priority ρ should run in time $O\left(\frac{W}{PF(\rho)} + \frac{S}{F(\rho)}\right)$, which begins to look like the bound of Theorem 3. Note that the above intuition is not the basis of a proof, in part because modeling the scheduler as $|R|$ separate instances of work stealing breaks down when threads spawn threads at other priorities and when priorities "donate" spare CPU cycles to other priorities.

There is one important case in which not all priorities have many threads available, contrary to our assumption. Suppose that a priority ρ is given a sizable fraction of the fairness criterion F, but

```
signature PRIORITY =
sig
  type t
  val top : t
  val bot : t

  val new : unit -> t
  val new_lessthan : t -> t -> unit

  val installDist : (t -> int) -> unit
end
```

```
signature SCHEDULER =
sig
  type task
  type handle
  val insert : Priority.t * task -> handle
  val suspend :
       (Priority.t * task -> unit) -> unit
end
```

Fig. 6. Simplified priority and scheduler interfaces

has only one thread available. This thread will be in the thread bank of one processor. Processors do not consider whether they have threads of a priority in their thread banks when choosing a primary priority, and so *each* processor will assign ρ as its primary priority during approximately $F(\rho)$ of its rounds. However, only the one processor that owns the ρ-priority thread will actually work on this thread, and so priority ρ will, in practice, receive only a fraction $F(\rho)/P$ of cycles. We do not expect this to be a performance issue in practice. In general, well-structured parallel computations have many more threads than processors. High-priority interactions may use only a small number of threads, but we have seen empirically that the performance of such interactions is fairly robust to receiving only small percentages of CPU time (see Figure 9).

5 IMPLEMENTATION

We have implemented the scheduling algorithm on top of a parallel extension to the MLton compiler for Standard ML, which was developed as part of Spoonhower's PhD thesis [Spoonhower 2009]. This compiler was a good base for our implementation for several reasons. MLton is a mature, whole-program optimizing compiler which produces fast sequential code. The parallel extension to MLton was developed a number of years ago, but is being actively maintained and improved [Guatto et al. 2018; Raghunathan et al. 2016]. Furthermore, MLton makes it easy to develop custom schedulers written directly in Standard ML with the use of its built-in library for user-level threads. This library does not provide scheduler implementations, but allows programs to represent paused computations and switch between them at runtime.

The main scheduler code consists of approximately 600 lines of ML code. The thread bank implementation is approximately an additional 75 lines. The remainder of this section describes the implementation details of the scheduler, as well as various optimizations we implemented for better practical performance.

5.1 Scheduler Interface

The low-level interfaces for the priority library and scheduler are shown in Figure 6 (with some details simplified for space reasons). The priority interface provides a type of priorities, a pre-defined top and bottom priority, and facilities for creating new priorities and defining the ordering between them. It also provides the function `installDist` allowing the programmer to install a fairness criterion, specified as a function from priorities to integers (the integers are summed over all priorities and normalized to 1.0). The scheduler interface provides low-level control over the scheduling queues; a programmer would not interface with this directly, but rather through higher-level threading libraries built on top of this interface. The operation `insert` is a wrapper around the corresponding queue operation. A thread can also call `suspend f` to yield control back to the scheduler. Before scheduling another thread, the scheduler calls the provided function `f` (which may, e.g., add the thread back to the queue for later rescheduling).

5.2 Implementation Details and Optimizations

Thread banks and fork potential. As described in the previous section, we wish to choose which threads in a bank to execute locally and which to send to other processors based on their "age" in the computation DAG. To accomplish this in practice, we implement thread banks as doubly-linked lists, sorted in increasing order of *fork potential*, inspired by a related concept used by Acar et al. [2013]. Each thread v has an associated fork potential $f(v) = 2^{-d(v)}$ where $d(v)$ is the fork-depth of thread v in the dependency graph (where the initial thread has depth 0, and every other thread has depth $1 + \max_u d(u)$ for its predecessors u). Fork potential is a good approximation of a thread's "size" in that a thread with high fork potential is more likely to spawn additional work. We implement the thread bank `split` operation by selecting the set of threads at the "back" of the bank (those with highest fork potential) which comprise at least $\frac{1}{4}$ of the total fork potential of the bank. Since banks typically remain small in well-behaved parallel programs (e.g. at most 100 threads), this is done simply with a linear scan through the bank. The `remove` operation simply pops the thread of smallest fork potential, and `insert` searches for the appropriate place in the list in order to maintain sorted order. Note that threads often spawn (or enable) other threads at the same priority, in which case we know that the new thread has smaller fork potential than any other thread in the corresponding bank and may be simply pushed onto the front of the list. Thus, in the common case, insertions are quite fast. The expense of maintaining the order of the thread bank is incurred only by splits and out-of-order insertions, which are much rarer in well-behaved programs. Pathological programs that violate these assumptions (e.g., by spawning many more threads than expected or frequently spawning threads at other priorities) would artificially inflate the work and (because the scheduler operations are sequential) span of the computation by the extra time spent in the scheduler. The response time and fairness bounds would degrade proportionally.

Finding the highest priority. To implement `findHighestPrio` efficiently, we use a heuristic which avoids unnecessary linear scans through the priorities. Each processor maintains its highest currently available priority in a global variable, the *top-priority cell.* The function `findHighestPrio` first checks the bank of the top priority to see if it has work. Processors update their own top-priority cell when they insert work at higher priorities, and atomically update other processors' cells on deals. If a processor executes its last thread at its top priority, the value in the cell will become out of date and the next check may require a full linear scan.

Interrupts. The scheduling algorithm of Section 4 assumes that the user code returns to the scheduler frequently in order to ensure that round switches and deals happen properly. Most well-written parallel code is fairly fine-grained, i.e. sequential computations do not run for long without spawning or synchronizing. However, to ensure proper load balancing and responsiveness even in the presence of long-running sequential computations, we use periodic interrupts delivered by interval timers to return control to the scheduler. When a thread is interrupted to return to the scheduler, the scheduler pushes the continuation of the running thread onto the thread bank so that it may be continued at a later time (possibly immediately if a priority switch does not occur). We determined experimentally that these interrupts should be at a finer granularity than the round-switch quantum. Specifically, our implementation performs interrupts at 1ms intervals and switches rounds at 5ms intervals. We set the deal interval to an even finer granularity, 100

```
signature THREAD =
sig
    exception IncompatiblePriorities
    type 'a t
    val spawn  : (unit -> 'a) -> Priority.t -> 'a t
    val join   : 'a t -> 'a
end
```

Fig. 7. Signatures for the threading library.

microseconds; deals will occur at this frequency when the program code frequently returns to the scheduler (coarser-grained code will still perform deals at the interrupt interval).

Requests. Sender-initiated algorithms such as ours perform well in situations where work on a small number of processors must be distributed. This is useful for distributing high-priority computations, which may be generated at one processor and need to be balanced onto the others. Sender-initiated algorithms are less useful when most processors have work and so most randomly targeted deal attempts will fail. We improve our implementation's performance in this case by adding an element of receiver-initiated algorithms in the form of *requests*. Each processor has a request cell. Processors which are idle at a particular priority may request work at that priority by writing their processor ID and the requested priority into a random processor's request cell. When a busy processor deals work, it will first check its request cell. If a request is present, it will attempt to deal work to the requesting processor instead of targeting randomly.

5.3 Programmer-Facing Threading Library

As discussed earlier in this section, programmers interact with a higher-level thread library implemented using the low-level operations exposed by the scheduler (Figure 6). We have implemented a threading library that exposes a threading construct similar to futures. As shown in Figure 7, the library exposes a first-class type `'a t` of threads with the return type `'a`. The operation `spawn f p` spawns a new thread at priority `p` to run the function `f`. The operation `join t` waits for thread `t` to finish and returns its return value. If blocking on `t` would create a priority inversion, it raises the exception `IncompatiblePriorities`. We will not dwell here on the implementation details of the threading library, as they are relatively straightforward.

6 EVALUATION

We evaluated the implementation against three criteria:

(1) (Fairness) Does the stretch experienced by low-priority jobs approximately match what would be expected from the fairness criterion under Theorem 2?
(2) (Promptness) When appropriate, does the scheduler execute high-priority work quickly?
(3) (Speedup) When focusing solely on parallel computational tasks, does the scheduler achieve good parallel speedup?

The last item, speedup, has not been a focus of this paper thus far, but is still important. We have defined stretch as the execution time of low-priority parallel tasks relative to their execution time if they were executed alone. A good scheduler should therefore achieve good speedups on purely computational tasks *and* still stay within the expected stretch when these tasks are combined with work at other priorities.

We perform this evaluation on a set of benchmarks combining interactive applications with standard parallel algorithms. At the end of the section, we describe two larger applications that show the power and flexibility of the system described in this paper.

6.1 Experimental Setup

All experiments were run on a machine with 1TB of main memory and 72 Intel Xeon cores running at 2.4GHz. We used up to 70 cores, reserving two for the experiment scripts and other background processes. When running on P cores, the Parallel MLton runtime pins P system-level worker threads to P cores. In the version of Parallel MLton we use, garbage collection is sequential and stops all processors. Because the time spent on GC is thus orthogonal to the scalability of our scheduling algorithm, we subtract the time spent on GC (typically under 150ms per run) for all reported *execution* times. We *do not* subtract GC time from response times.

Evaluation framework. For testing our interactive benchmarks, we use a driver program, written in C, which shares pipes and network connections with the benchmark application. The driver simulates interactions as specified by an "event trace" consisting of a sequence of interactive actions, encoded in a domain-specific language. Our interactive benchmarks respond to inputs with console output. When the driver initiates an event, it adds an entry in a buffer with the start time and expected response. A separate thread in the driver reads the output of the program and checks it against the buffer entries. When a line of output matches the expected response for an event, the thread records the time difference and uses it to compute the average response time.

6.2 Benchmarks

To gain a controlled experimental setting for doing a thorough quantitative evaluation of the scheduler, we use a set of *orthogonal benchmarks*, in which computations at different priorities do not affect each other, allowing us to tune them independently.

Each of our orthogonal benchmarks consists of some combination of *compute kernels* and *interactive kernels*. Each of the kernels operates independently of the others, and they may all be run at any priority. All of the compute kernels and one of the two interactive kernels are themselves parallel. The set of kernels was chosen to cover both predictable workloads, where the parallel structure is relatively static, and unpredictable workloads, where the parallel structure is determined dynamically during computation. For the compute kernels, we also consider both compute-intensive and memory-intensive workloads.

The remainder of this section is structured as follows. In Sections 6.2.1 and 6.2.2, we describe the interactive and compute kernels we use in the orthogonal benchmark evaluation. The first experiment we describe (Section 6.2.3) evaluates how well our scheduler preserves the desired fairness guarantees by comparing the stretch of low-priority jobs to the expected stretch. We can only compare to the theoretically optimal stretch rather than the performance of another system because we are not aware of another fair cooperative threading system that would provide an apples-to-apples comparison. The next experiment (Section 6.2.4) gives a more direct comparison to two cooperative parallel systems, both of which are also based on Standard ML, but designed for more specific use cases. This experiment focuses on the response time of high-priority tasks and the overall speedup of low-priority parallel computation, and places our performance in the context of that of comparable systems. This experiment shows that our very general system can compete with more specialized ones (i.e. that one is not paying a high price for this generality when one is not using it). Finally, in Section 6.2.5, we compare this paper's approach to two well-known cooperative threading systems, Cilk and Go. These systems are not designed for interactive applications, so these results are not meant to be a direct apples-to-apples comparison, but rather they highlight the need for specialized schedulers for handling interactive workloads.

6.2.1 Interactive Kernels. The two interactive kernels are Terminal echo (abbreviated "Term" below) and Network echo (abbreviated "Net").

Terminal echo. The kernel repeatedly accepts a line of text on standard input and echoes it back to the user on standard output. This kernel is sequential and predictable.

Network echo. The kernel listens for network connections in a main, single-threaded loop. For each connection, the loop spawns a new thread to handle this client. Each such thread interacts with the client as in terminal echo above, accepting a line of text over the socket and echoing the text on standard output. This kernel is parallel and unpredictable, since the parallel structure depends on the incoming connections.

6.2.2 Compute Kernels. There are five compute kernels: Fibonacci (abbreviated "Fib"), Unbalanced Tree Search ("UTS"), Dense Matrix Multiplication ("DMM"), Breadth-First Search ("BFS") and Sample Sort ("Sort").

Fibonacci. A naturally parallel recursive algorithm computes the 45^{th} Fibonacci number. While this is not an efficient algorithm for computing Fibonacci numbers, it generates many parallel tasks quickly with little to no allocation, making it an excellent representation of a computation-intensive, predictable parallel workload.

Unbalanced Tree Search (UTS). UTS [Olivier et al. 2006], designed as an adversarial test of load balancing, explores a tree where, at each node, the number of children is randomly selected from a geometric distribution. This results in a high degree of imbalance, stressing the scheduler's distribution of work. The tree we use has depth 11 and average branching factor 4.0. Since our random number generation is fairly inexpensive, we add a small sequential Fibonacci computation at each node to increase the work. This benchmark is highly unpredictable and computation-intensive.

Dense Matrix Multiplication (DMM). This is a parallel, recursive implementation of Strassen's algorithm for matrix multiplication, which we use to multiply two 2048×2048 matrices. This benchmark is still more compute-intensive than memory-intensive, and also results in a fairly predictable parallel structure, but is less artificial than Fibonacci.

Breadth-First Search (BFS). This benchmark performs a breadth-first search of a randomly generated graph with 15M nodes and 240M edges. The graph is stored using a compact, array-based adjacency list representation. This benchmark is memory-intensive, with a somewhat unpredictable parallel structure based on the graph structure.

Sample Sort. Our implementation of sample sort is based on that of Blelloch et al. [2010], and is highly optimized to make maximum use of parallel processing and hierarchical caches. As a benchmark, it is memory-intensive but with a more predictable structure than BFS. The benchmark sorts a random sequence of length approximately 128M.

6.2.3 Measuring Fairness. The primary feature that sets our scheduler apart from existing scheduling algorithms is its ability to guarantee a pre-defined stretch for low-priority computations based on the given fairness criterion. This is the feature we evaluate in our first experiment. For this experiment, we use a set of benchmarks with three priorities. The top priority runs an interactive kernel, and the bottom priority runs a compute kernel. The middle priority acts as a time sink. It runs a very large, massively parallel computation which is not expected to finish or idle during the experiment. It will therefore collect any cycles donated by the top priority, allowing us to view the

Table 1. Execution times and stretch (T_{70}/T_{70}^b) for three-priority benchmarks, run with 50 interactions per second. Each benchmark has a large computation running at medium priority.

			50-0-50		50-25-25	
High prio.	Low prio.	T_{70}^b (s)	T_{70} (s)	Stretch	T_{70} (s)	Stretch
Terminal	Fibonacci	0.40	0.93	2.31	2.00	4.96
	UTS	0.40	0.85	2.13	2.03	5.13
	DMM	0.53	1.31	2.47	3.23	6.11
	BFS	0.65	2.55	3.93	6.13	9.46
	Sample Sort	1.01	2.80	2.77	6.09	6.03
Network	Fibonacci	0.40	0.90	2.23	2.02	4.99
	UTS	0.39	0.90	2.32	1.97	5.06
	DMM	0.50	1.33	2.66	3.49	6.95
	BFS	0.73	2.47	3.39	6.15	8.43
	Sample Sort	1.01	2.77	2.74	6.50	6.42
Expected Stretch				2.00		4.00

low-priority computation in isolation. Each benchmark runs until the low-priority computation completes, and then terminates.

We ran each combination of compute and interaction kernels with three different fairness criteria, written in the form "*H*-*M*-*L*", where H is the percentage of cycles devoted to the high priority, M to the middle priority and L to the low priority. The expected stretch of the low-priority computation for each fairness criterion is therefore $100/L$. We ran each experiment on 70 cores and the driver program interacted with the interactive kernel 50 times per second. For this evaluation, we ran each benchmark with the fairness criterion 0-0-100 (i.e. devoting all cycles to the low-priority computation) to measure the baseline for computing stretch, which we denote T_{70}^b. We then ran them with the fairness criteria 50-0-50 and 50-25-25, aiming for stretches of 2 and 4, respectively, while keeping the cycles allocated to the top priority constant.

Table 1 gives the baseline T_{70}^b and the 70-core execution time (T_{70}) in seconds and the stretch for 50-0-50 and 50-25-25. The ratios between the actual and desired stretch are shown graphically in Figure 8 (note that, by definition, the expected and desired stretch for 0-0-100 are identical). If the scheduler is being perfectly fair, these ratios should be 1.0. For the compute-bound benchmarks (Fibonacci, UTS and DMM), the ratios stay very close to 1. For the memory-bound benchmarks, the ratios go as high as 2.5. We do not expect perfect fairness because context switching between threads, which is necessary for a fair scheduler, always introduces some overhead not accounted for by our theoretical model, which assumes that scheduling decisions and allocation can be performed instantaneously. This overhead includes the overhead of preemption itself, as well as other effects like disrupting temporal locality. These increase the time necessary to complete the tasks, which in turn impacts their stretch. The effect on locality, in particular, may explain why the memory-bound benchmarks suffer more overhead.

Furthermore, we would expect that the impact of small context-switching overheads would increase the smaller a computation's fairness criterion (in the extreme limit, think of the overhead of context-switching to a thread for just one quantum and then doing other work for a long period of time). This explains why the computation deviates more from perfect fairness under a 50-25-25 criterion than a 50-0-50 one.

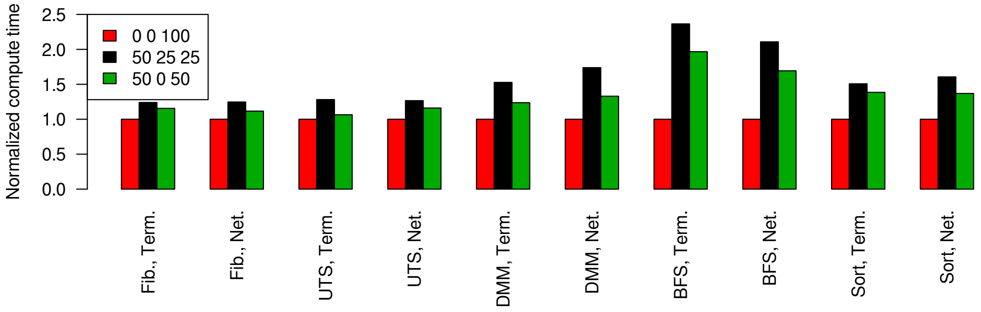

Fig. 8. Normalized execution times for the low-priority computation, calculated as $LT_{70}/100T_{70}^{b}$.

Table 2. Parallel speedup of computation-only benchmarks on our scheduler and Spoonhower Work Stealing (SWS), as well as the relative performance of our scheduler. Higher is better.

	4 procs			32 procs			70 procs		
Comp.	SWS	Ours	$\frac{\text{Ours}}{\text{SWS}}$	SWS	Ours	$\frac{\text{Ours}}{\text{SWS}}$	SWS	Ours	$\frac{\text{Ours}}{\text{SWS}}$
Fib.	2.9×	2.6×	0.91	16.8×	15.3×	0.91	33.5×	29.7×	0.88
UTS	2.3×	2.5×	1.08	13.6×	13.6×	1.00	26.6×	24.6×	0.93
DMM	1.6×	2.1×	1.29	8.7×	11.6×	1.32	14.0×	22.2×	1.59
BFS	2.5×	2.4×	0.98	19.0×	14.3×	0.75	37.5×	19.4×	0.52
Sort	4.4×	4.4×	0.99	29.6×	26.5×	0.89	62.6×	50.0×	0.80

6.2.4 Measuring Scalability and Response Times. In the next set of experiments, we compare our scheduler to two existing cooperative parallelism systems based on MLton, in order to place the response times and speedups of our scheduler in the context of results for similar approaches. The first comparison system is Spoonhower's original work stealing scheduler, on which our implementation is built [Spoonhower 2009]. This scheduler is quite effective at balancing purely computational tasks, and so makes a good baseline for comparing the scaling of computation times. The second comparison is with the responsive scheduler of [Muller et al. 2017], which was also based on Spoonhower's framework and schedules interactive workloads to optimize for both computation and response time, but only for programs with two priorities.

For the comparison to Spoonhower Work Stealing (SWS), we use benchmarks consisting of only a compute kernel (because SWS was not designed to handle interactive programs). Table 2 shows the parallel speedup for each compute kernel on both our scheduler and SWS. Each speedup shown is with respect to the *serial baseline*, a sequential version of the same compute kernel compiled with unmodified MLton. That is, the speedup is calculated as $\frac{T_s}{T_P}$, where T_s is the computation time of the serial baseline and T_P is the computation time of the parallel benchmark on P cores. The table also shows the ratio between the speedup of our algorithm and that of SWS: a ratio of 1.0 indicates that the two are equal, and higher values show an improvement of our algorithm over SWS. For smaller numbers of cores, our scheduler is highly competitive with Spoonhower on most benchmarks, and even performs substantially faster on DMM. At 70 cores, our performance begins to decline for the BFS and Sample Sort benchmarks, indicating that there is still some work to be done to improve the scalability of our work stealing algorithm for these memory-intensive computations.

Table 3. Parallel speedup and response time of two-priority, winner-take-all benchmarks on our scheduler and Muller-Acar-Harper (MAH). Lower is better for response time, higher is better for speedup.

	1 proc resp. time (ms)			70 proc Speedup		
Comp.	MAH	Ours	$\frac{\text{Ours}}{\text{MAH}}$	MAH	Ours	$\frac{\text{Ours}}{\text{MAH}}$
Terminal interaction						
Fib.	2.6	2.6	1.03	25.5×	30.3×	1.19
UTS	2.6	2.6	0.99	15.7×	22.9×	1.46
DMM	2.8	2.5	0.91	18.7×	21.1×	1.13
BFS	199.6	2.7	0.01	12.4×	19.1×	1.54
Sort	13.8	2.5	0.18	34.8×	49.0×	1.41
Network interaction						
Fib.	2.6	2.5	0.98	24.5×	32.1×	1.31
UTS	2.6	2.6	0.98	16.3×	21.0×	1.29
DMM	2.6	2.5	0.97	19.8×	20.9×	1.05
BFS	9.7	2.6	0.27	11.4×	19.2×	1.68
Sort	3.5	2.6	0.73	33.7×	49.3×	1.47

Promptness requires that when work at the primary priority is unavailable, the processor works on the highest available priority. We show this using benchmarks with one high-priority interactive kernel and one low-priority compute kernel, run with a winner-take-all policy (the high priority interaction is always chosen as the primary priority and the low priority computation is only run when no interactive thread is ready) and a driver that interacts with the benchmark at 50 interactions per second. Because there is relatively little high-priority work and the policy is winner-take-all, we would expect to see the large majority of CPU cycles spent on computation, but see the scheduler quickly switch to handling interaction when it occurs.

Table 3 shows the parallel speedups and response times for each benchmark, along with the same measurements for the winner-take-all scheduler of prior work (MAH). The serial baseline used for the speedups included only the sequential version of the compute kernel, again compiled with unmodified MLton. In all benchmarks, our scheduler matches or out-performs the specialized scheduler in both response time and speedup. Furthermore, the performance of our scheduler in these experiments is comparable to that of Table 2, indicating that very few cycles (only those that are necessary to complete the interaction) are devoted to the interaction, as required by promptness.

The experiments above show that, without priorities and fairness requirements, the scaling and response times of our scheduler are usually competitive with prior approaches. In order to show that these properties are not unduly harmed by the addition of more priorities and fairness requirements, we return to the three-priority benchmarks of the previous subsection. Table 4 shows the 70-core speedup for the 0-0-100 fairness criterion and the *normalized speedups* for the other two fairness criteria. The normalized speedup is the parallel speedup multiplied by the expected stretch. This normalization accounts for the fact that the scheduler is not expected or allowed to devote all of its processing time to the low-priority task being measured. The maximum normalized speedup in all cases is 70× on 70 cores. The speedups are lower than those for the purely computational benchmarks reported in Table 2, but not drastically so, indicating a modest performance penalty for using interaction and priorities. The response times for the benchmarks are also reported in Table 4 for the 50-0-50 and 50-25-25 criteria. Response times for 0-0-100 are quite poor: they range

Table 4. Normalized speedups ($100T_s/LT_{70}$) and response times for three-priority benchmarks, run with 50 interactions per second. Each benchmark has a large computation running at medium priority.

			0-0-100	50-0-50		50-25-25	
High prio.	Low prio.	T_s (s)	Speedup	Speedup	RT (ms)	Speedup	RT (ms)
Terminal	Fibonacci	11.67	28.9×	25.0×	4.7	23.3×	8.4
	UTS	5.61	14.1×	13.3×	4.2	11.0×	6.1
	DMM	7.48	14.2×	11.4×	4.6	9.3×	6.3
	BFS	8.43	13.0×	6.6×	3.3	5.5×	6.1
	Sample Sort	34.67	34.3×	24.8×	22.2	22.8×	35.4
Network	Fibonacci	11.67	28.9×	25.9×	5.3	23.2×	7.0
	UTS	5.61	14.4×	12.4×	3.9	11.4×	6.6
	DMM	7.48	14.9×	11.2×	4.7	8.6×	6.8
	BFS	8.43	11.6×	6.8×	3.5	5.5×	6.1
	Sample Sort	34.67	34.3×	25.1×	6.2	21.3×	18.0

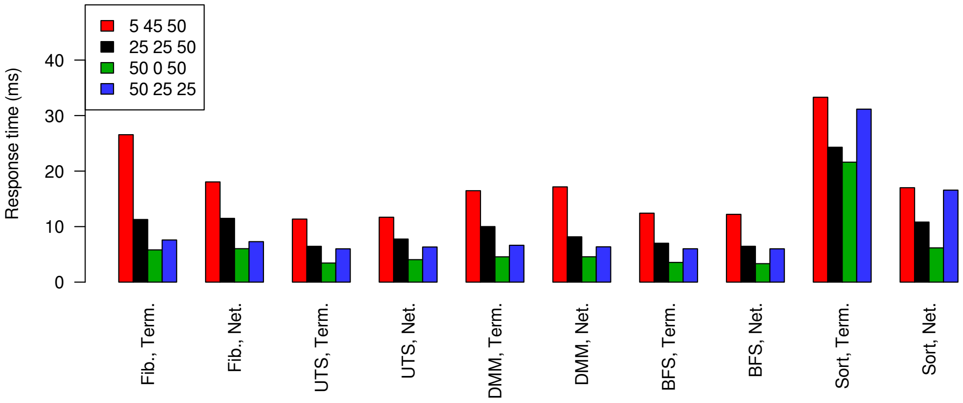

Fig. 9. Response times for 50 interactions per second on three-priority benchmarks.

as high as 800ms, and in many cases, events are dropped entirely, as is to be expected when no cycles are devoted to the interaction (though, because of promptness, idle processors will work on high-priority tasks). Response times are otherwise quite good, remaining well under 5ms except for the very intensive Sample Sort computation.

Finally, we show how many cycles are needed to guarantee responsive interaction For this experiment, we also measured the response times for fairness criteria 25-25-50 and 5-45-50. The results for 50 terminal interactions per second are shown in Figure 9. As expected, mean response times are higher when very few cycles are devoted to interaction, though they remain under 50ms, showing that for applications where responsiveness is not critical, few cycles need to be devoted to handle infrequent bursts of interaction. We repeated this experiment for 10 and 100 interactions per second with broadly similar results, showing that the scheduler is able to scale well with interaction rate. As these results are similar to those shown in Figure 9, we omit them.

Table 5. Computation and response times of the Fibonacci-network benchmark using our system, Cilk and Go. Lower numbers are better.

Procs		Ours	Cilk	Go
1	Computation time Response time	74.6 s 2.6 ms	∞ —	29.5 s 3723.1 ms
4	Computation time Response time	18.8 s 2.6 ms	14.7 s 660.3 ms	8.4 s 744.6 ms
8	Computation time Response time	9.7 s 2.6 ms	6.6 s 217.0 ms	4.4 s 385.0 ms
32	Computation time Response time	2.9 s 2.5 ms	1.7 s 0.1 ms	1.6 s 227.8 ms
70	Computation time Response time	1.5 s 2.7 ms	0.8 s 0.1 ms	1.0 s 140.7 ms

6.2.5 Comparison to Cilk and Go. To show the practicality of our approach, we compare our algorithm against two widely used systems for fine-grained parallelism, Cilk and Go. This comparison shows the cost of our approach relative to well-engineered throughput-oriented systems and, since neither Cilk nor Go provides the ability to prioritize threads, demonstrates the benefit of priority scheduling for response time. These results are not meant to be taken as a direct comparison: there are many extra variables when comparing between approaches based on different languages and systems that provide different sets of features.

Cilk [Frigo et al. 1998] extends C with primitives for creating and synchronizing lightweight threads. We use the version of the Intel Cilk Plus scheduler that is integrated with gcc version 6.4.0. Go [Go Authors 2018] provides the ability to perform function calls, called goroutines, asynchronously. Our comparisons use Go version 1.6.2.

We performed the comparison to Cilk and Go using a benchmark that runs the Fibonacci compute kernel in parallel with the Network interactive kernel. To ensure that the Cilk and Go versions of the benchmark ran for long enough to accurately measure response time, we increased the Fibonacci input to 48 and computed the result modulo a large number (to not overflow the integer data type). The results are shown in Table 5 for 1, 4, 8, 32 and 70 cores. Exact results are not shown for Cilk on one core because Cilk does not preempt blocked program threads, so blocking on I/O will block the entire processor. The response time of the Cilk code on fewer cores is quite poor. Connections are accepted quickly because the code that accepts connections is running in a tight loop on an essentially-dedicated processor. However, when this loop spawns a new Cilk thread to handle an incoming connection, that thread must wait to be stolen by an idle core—the scheduler has no way to prioritize these threads over Fibonacci threads. The response times improve with more cores because of the greater chance of the interactive thread being scheduled at any point in time. The Fibonacci computation scales well, but not as well as Cilk computations typically scale, because as soon as an interaction thread migrates to a processor, that processor will not perform any more computation due to the lack of latency hiding. The computation in Go scales better (relative to the typical performance of the two languages) because it hides latency, but its response times are again significantly worse than ours because Go does not explicitly prioritize interactive threads. As with Cilk, response times improve on more cores, but not to the same extent because interaction threads will never have an entire processor dedicated to them.

6.3 Applications

The previous subsection focused on small benchmarks that provide for controlled quantitative evaluation. We now consider larger, more complex applications. These applications demonstrate the flexibility and scalability of our approach, but are not as amenable to precise quantitative evaluations, and are presented more as qualitative case studies.

6.3.1 Motion Planning. We consider a motion planner for a robot which uses two planning algorithms organized in a hierarchical fashion [Knepper et al. 2010]. A collection of threads at low priority runs a parallel version of the A* search algorithm to compute a coarse-grained plan consisting of a series of landmarks leading toward the goal. The A* algorithm maintains a priority queue of nodes which it expands in order. The details of how we parallelize the algorithm are not central to the paper, but at a high level, we split the queue when it grows past a certain threshold and create a new thread to process part of the queue, capping the total number of threads by the number of cores. Because local decisions of which node to expand may not be globally optimal, this algorithm achieves parallelism at the cost of optimality.

At medium priority, several instances of the Rapidly exploring Random Trees (RRT) algorithm [LaValle 1998] compute detailed paths to the nearest landmark. Each instance of RRT is sequential, but deploying several instances simultaneously and taking the best result leads to more optimal plans. Finally, a high-priority interaction loop polls the paths and attempts to execute the most efficient one.

The motion planner differs from our orthogonal benchmarks in several significant ways: it exhibits substantial computation (not just small bursts of interaction) at multiple priorities, and fairness is crucial to balance the long- and short-term planners. In addition, the planner is not orthogonal: the interaction loop interacts closely with both the A* and RRT threads. While we are able to observe speedup from our parallel version of A*, and some improvement in planning by using multiple simultaneous RRT computations, parallelizing search algorithms is an area of research in itself, and so this application is not yet suitable for a thorough quantitative analysis. We will, however, report some qualitative results based on our experience running the planner on various numbers of processors and under various fairness criteria.

We tested our planner in the context of a simulated environment using a map from a standard set of 2-dimensional motion planning benchmarks taken from video games [Sturtevant 2012]. In keeping with the ratio suggested by Knepper et al. [2010], we ran the motion planning simulation with a fairness criterion that assigns four times as many cycles to the short-term planner as to the long-term planner. The interaction loop was given the same number of cycles as the short-term-planner. This corresponds to a stretch of 6 for the long-term planner and a stretch of 1.5 for the short-term planner.

Under this fairness criterion, the robot generally moves smoothly toward the goal without stopping frequently to replan. When the stretch of the short-term planner is increased (e.g. the stretches of the two planners are equal), this planner does not run quickly enough and the robot pauses frequently. Conversely, when the stretch of the long-term planner is increased, the robot moves smoothly but becomes stuck easily if it drifts off of the plan at all, because the long-term planner is starved and not easily able to correct the plan. When all cycles are allocated to the interaction loop, both planners are starved and the robot does not make progress.

6.3.2 Photo Viewer. Our photo viewer application, derived from a benchmark from prior work [Muller et al. 2017], displays JPEG images from a local folder, and allows the user to scroll forward and backward, and to jump to images out of order. When a new image is selected, the next 8 images are decoded by low-priority threads. The interaction with the user occurs in a high-priority foreground

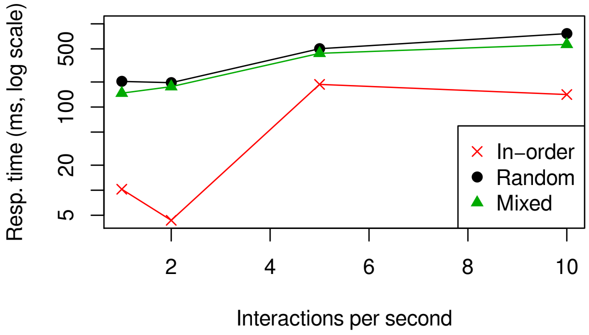

Fig. 10. Response time results for the photo viewer.

thread. When the user selects a new image (either by scrolling or jumping), the foreground thread checks if the image has already been decoded and either displays the decoded image or decodes it and displays it immediately.

The third-party libraries we use for decoding and rendering images are not thread-safe, and so we are only able to run the photo viewer on one core, using threads for latency hiding rather than parallelism. Still, latency hiding can be quite beneficial in this benchmark, if the thread waiting for user input properly yields to the background decoding threads, and these background threads properly yield to each other. Thus, even on one core, this benchmark provides a good test of our scheduler's promptness. Because high-priority events (responding to user input and decoding immediately-requested images) are rare, a winner-take-all strategy is appropriate.

In our tests of the photo viewer, we directed the driver program to request a series of 10 images, out of 100 total images, each of which is 4.3MB in size and takes approximately 225ms to load from file and decode. Figure 10 shows response times for four interaction rates (1, 2, 5 and 10 requests per second) and three types of traces: in-order access, random access and a mix of the two. The y-axis is the response time on a log scale, measured from when the driver requests the image to when the decoded image is displayed on the screen. For in-order traces where requests are made slowly enough to use pre-decoded images, response times are within a handful of milliseconds. These response times show that the interaction thread properly donates its spare cycles to decoding, and quickly wakes up and responds to interaction when necessary. As expected, the benefit of pre-decoding images decreases when viewing the photos in a random order or scrolling faster than the viewer can decode.

7 RELATED WORK

Scheduling techniques for optimizing throughput and responsiveness, and ensuring fairness have been studied extensively in several communities including parallel computing, operating systems, and queueing theory. In this section, we discuss the closely related work and its relation to this paper and refer the interested readers to excellent surveys and books (cited below) on these areas for broader discussions.

Parallel Scheduling. Parallel schedulers aim to minimize the completion time of a parallel computation. Many results have been shown for parallel scheduling, including on runtime [Acar et al. 2018, 2013; Agrawal et al. 2014; Arora et al. 2001; Blumofe et al. 1996; Blumofe and Leiserson 1999; Brent 1974; Burton and Sleep 1981; Eager et al. 1989; Frigo et al. 1998; Greiner and Blelloch 1999; Halstead 1985; Muller and Acar 2016; Tardieu et al. 2014; Ullman 1975], space usage [Blelloch et al. 1999;

Blumofe and Leiserson 1998; Narlikar and Blelloch 1999], cache utility [Acar et al. 2002; Blelloch et al. 2011; Blelloch and Gibbons 2004; Chowdhury and Ramachandran 2008], and granularity control [Acar et al. 2018, 2016]. Nearly all of these schedulers implement the greedy scheduling principle [Brent 1974; Eager et al. 1989], which requires keeping processors as busy as possible, but differ in the precise assignments of tasks to processors. All these schedulers can be viewed as *cooperative* schedulers because they don't treat threads as competing for the same resources but schedule them cooperatively with the goal of maximizing throughput. Other work has shown that scheduling based on priorities [Imam and Sarkar 2015; Wimmer et al. 2013, 2014] can improve throughput in some applications.

The primary focus of parallel scheduling, and the papers above, has been to maximize throughput or speedup. Recent work [Muller et al. 2017, 2018] extends traditional parallel scheduling to maximize responsiveness of interactive threads. That work proposed the prompt scheduling principle, which specifies the assignment of threads to processors, but does not present a scheduling algorithm for generating the assignment efficiently. In this paper, we extend prompt scheduling with fairness. We also present and evaluate a scheduling algorithm that implements the new scheduling principle.

Operating Systems Community. Scheduling has been studied extensively in the operating systems community, where criteria such as responsiveness, CPU utilization, and throughput are all considered. A classic book [Silberschatz et al. 2005] presents a comprehensive overview of this work. More recently, there has been significant interest in making operating systems work well on multicore machines [Baumann et al. 2009; Boyd-Wickizer et al. 2008] by reducing contention within the OS and by distributing resources to jobs so that they can run effectively. The goal of OS scheduling differs from ours, because OS schedulers operate at the level of system threads, rather than the lightweight user-level threads that we consider. Dependencies between processes, a major focus of parallel scheduling, are also less central to OS scheduling.

Fairness in particular has been a focus of research in the systems community. Lottery scheduling [Waldspurger and Weihl 1994] and fair queueing (e.g., [Demers et al. 1989; Goyal et al. 1996] are two mechanisms that, like our scheduler, schedule threads or jobs fairly given assigned weights. In the case of lottery scheduling, these weights are the distribution of tokens ("tickets") to individual threads, not to entire priority classes as in our system. Fair queueing models also accept weighted jobs, but schedule jobs deterministically based on their weight. Goyal et al. [1996] propose a mechanism for scheduling hierarchical classes of jobs, in which each class has a weight. At each non-leaf node of the hierarchy, a fair decision is made on which of the node's children to schedule based on their weights. Our scheduling algorithm could be seen as an example of such a hierarchy where the classes are priorities. All of this work focuses on scheduling coarser-grained jobs and so uses very different scheduling strategies from our fine-grained model. The Linux kernel's "completely fair scheduler", for example, shares our goal of minimizing a process's deviation from an "ideal" execution time. It does so using per-process data structures that record metadata about prior execution times. This metadata allows the scheduler to use heuristics such as boosting the priority of tasks that have yielded some of their time by sleeping, but the technique would not scale to the magnitude of threads generated by fine-grained parallel programs. Furthermore, we are not aware of any efforts to bound the theoretical performance and fairness of the scheduling algorithms and heuristics used in operating systems.

Real-Time Systems Community. Scheduling is also an important concern in real-time computing. Traditionally, the real-time scheduling community focused on scheduling sequential tasks, and on taking advantage of parallelism at the task level. Recently there has been more interest in scheduling tasks that are themselves parallel [Li et al. 2014, 2016, 2017; Mattheis et al. 2012; Saifullah et al. 2014a, 2013]. Most of this work, however, considers highly structured (e.g., synchronous dataflow) computations. For example, Saifullah et al. [2014b] consider scheduling a set of real-time tasks where each task is a parallel computation. The idea is to decompose the task's dependency graph into a set of sequential pieces which can then be executed separately. This approach requires prior knowledge of the parallel structure of the task. Another prevailing assumption is that tasks arrive independently according to some process. Our work differs from real-time scheduling in that we consider fully general and online computations. In particular, we make no assumption on the structure of the jobs and don't assume that they are known ahead of time.

Queueing Theory Community. In queueing theory, there has been long history of work on scheduling for responsiveness of jobs. Harchol-Balter's book [2013] presents a comprehensive overview. This work typically assumes that *a continuous stream* of *independent* jobs arrive online according to some stochastic process and aims to minimize the response time between when a job arrives and completes. Although the goal of minimizing the response time is shared with our work, the arrival assumptions do not fit parallel interactive applications, where work is created by a program and dependencies between threads matter. In this community, jobs are also usually assumed to be sequential.

8 CONCLUSION

We present techniques for writing parallel programs that allow threads to be assigned arbitrary priorities and appropriate requirements on stretch, and propose a scheduling principle, called *fairly prompt* scheduling, for executing such programs. We have proven bounds for the execution and response times of threads under fairly prompt schedules. In particular, these bounds show that fairness prevents low-priority jobs from being starved or unduly delayed.

Our implementation of a scheduling algorithm based on work stealing and the empirical evaluation of this implementation show that the priority model and scheduler are robust and general enough to express a variety of benchmarks that mix interactive and compute-intensive tasks. This class of benchmarks broadens the traditional benchmarking suites with interactivity. The measurements show that the algorithm performs well in practice, successfully maintaining the specified fairness, while ensuring responsiveness of interactive tasks, and maximizing throughput of compute-intensive ones. Moreover, for applications that use some or none of our priority and fairness facilities, the scheduler is competitive with existing schedulers specialized to those domains, indicating that one doesn't pay for the features of our more general scheduler if one isn't using them. The results thus suggest that it is possible to combine two prevailing models of threading, cooperative and competitive threading, efficiently under one roof.

ACKNOWLEDGMENTS

This research was partially funded by the National Science Foundation. The authors would like to thank Samuel Yeom, Peter Elliott and the anonymous reviewers for their helpful feedback.

REFERENCES

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2002. The data locality of work stealing. *Theory of Computing Systems (TOCS)* 35, 3 (2002), 321–347.

Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. 769–782.

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling Parallel Programs by Work Stealing with Private Deques. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*.

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2016. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *Journal of Functional Programming (JFP)* 26 (2016), e23.

Kunal Agrawal, Jeremy T. Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. 2014. Provably Good Scheduling for Parallel Programs That Use Data Structures Through Implicit Batching. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '14)*. 84–95.

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* 34, 2 (2001), 115–144.

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. 29–44.

Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. 2010. Evolution of Thread-level Parallelism in Desktop Applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. 302–313.

Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2011. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11)*. 355–366.

Guy E. Blelloch and Phillip B. Gibbons. 2004. Effectively sharing a cache among threads. In *SPAA*. https://doi.org/10.1145/1007912.1007948

Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. 1999. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM* 46 (March 1999), 281–321. Issue 2.

Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low Depth Cache-oblivious Algorithms. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. ACM, New York, NY, USA, 189–199. https://doi.org/10.1145/1810479.1810519

Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. 1994. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel Distrib. Comput.* 21, 1 (1994), 4–14.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel and Distrib. Comput.* 37, 1 (1996), 55 – 69.

Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. Comput.* 27, 1 (1998), 202–229.

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46 (Sept. 1999), 720–748. Issue 5.

Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. 43–57.

Richard P. Brent. 1974. The parallel evaluation of general arithmetic expressions. *J. ACM* 21, 2 (1974), 201–206.

F. Warren Burton and M. Ronan Sleep. 1981. Executing functional programs on a virtual tree of processors. In *Functional Programming Languages and Computer Architecture (FPCA '81)*. ACM Press, 187–194.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*. ACM, 519–538.

Rezaul Alam Chowdhury and Vijaya Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. In *Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, New York, NY, USA, 207–216. https://doi.org/10.1145/1378533.1378574

A. Demers, S. Keshav, and S. Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *Symposium Proceedings on Communications Architectures &Amp; Protocols (SIGCOMM '89)*. ACM, New York, NY, USA, 1–12.

Derek L. Eager, John Zahorjan, and Edward D. Lazowska. 1989. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computing* 38, 3 (1989), 408–423.

Kristián Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge. 2000. Thread-level Parallelism and Interactive Performance of Desktop Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. 129–138.

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2011. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming* 20, 5-6 (2011), 1–40.

Matthew Fluet, Mike Rainey, John H. Reppy, and Adam Shaw. 2008. Implicitly-threaded parallelism in Manticore. In *ICFP*. 119–130.

Nissim. Francez. 1986. *Fairness*. Springer US, New York, NY.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. 212–223.

C. Gao, A. Gutierrez, R. G. Dreslinski, T. Mudge, K. Flautner, and G. Blake. 2014. A study of Thread Level Parallelism on mobile devices. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on.* 126–127.

The Go Authors. 2018. The Go Programming Language Specification. (Feb. 2018). https://golang.org/ref/spec#Go_statements

Pawan Goyal, Xingang Guo, and Harrick M. Vin. 1996. A Hierarchial CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI '96).* ACM, New York, NY, USA, 107–121.

John Greiner and Guy E. Blelloch. 1999. A Provably Time-efficient Parallel Implementation of Full Speculation. *ACM Transactions on Programming Languages and Systems* 21, 2 (March 1999), 240–285.

Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018.* 81–93.

Robert H. Halstead. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 7 (1985), 501–538.

Mor Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action.* Cambridge University Press.

Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. 1993. Using Threads in Interactive Systems: A Case Study. *SIGOPS Oper. Syst. Rev.* 27, 5 (Dec. 1993), 94–105.

Shams Imam and Vivek Sarkar. 2015. Load Balancing Prioritized Tasks via Work-Stealing. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing.* 222–234.

Shams Mahmood Imam and Vivek Sarkar. 2014. Habanero-Java library: a Java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14.* 75–86.

Intel. 2011. Intel Threading Building Blocks. (2011). https://www.threadingbuildingblocks.org/.

Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. 2010. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (ICFP '10).* 261–272.

R. A. Knepper, S. S. Srinivasa, and M. T. Mason. 2010. Hierarchical planning architectures for mobile manipulation tasks in indoor environments. In *2010 IEEE International Conference on Robotics and Automation.* 1985–1990. https://doi.org/10.1109/ROBOT.2010.5509669

Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. 2014. Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14).* ACM, New York, NY, USA, 2–14. https://doi.org/10.1145/2594291.2594312

Steven M LaValle. 1998. *Rapidly-exploring random trees: A new tool for path planning.* Technical Report TR 98-11. Computer Science Dept., Iowa State University.

Doug Lea. 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande (JAVA '00).* 36–43.

Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09).* 227–242.

Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. 2014. Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks. In *Proceedings of the 2014 Agile Conference (AGILE '14).* IEEE Computer Society, Washington, DC, USA, 85–96. https://doi.org/10.1109/ECRTS.2014.23

Jing Li, Son Dinh, Kevin Kieselbach, Kunal Agrawal, Christopher Gill, and Chenyang Lu. 2016. Randomized Work Stealing for Large Scale Soft Real-time Systems. In *Real-Time Systems Symposium (RTSS), 2016 IEEE.* IEEE, 203–214.

Jing Li, David Ferry, Shaurya Ahuja, Kunal Agrawal, Christopher Gill, and Chenyang Lu. 2017. Mixed-criticality Federated Scheduling for Parallel Real-time Tasks. *Real-Time Syst.* 53, 5 (Sept. 2017), 760–811. https://doi.org/10.1007/s11241-017-9281-8

Sebastian Mattheis, Tobias Schuele, Andreas Raabe, Thomas Henties, and Urs Gleim. 2012. Work Stealing Strategies for Parallel Stream Processing in Soft Real-time Systems. In *Proceedings of the 25th International Conference on Architecture of Computing Systems (ARCS'12).* Springer-Verlag, Berlin, Heidelberg, 172–183.

Stefan K. Muller and Umut A. Acar. 2016. Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016.* 71–82. https://doi.org/10.1145/2935764.2935793

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, New York, NY, USA, 677–692.

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2018. Competitive Parallelism: Getting Your Priorities Right. *Proc. ACM Program. Lang.* 2, ICFP, Article 95 (July 2018), 30 pages. https://doi.org/10.1145/3236790

Girija J. Narlikar and Guy E. Blelloch. 1999. Space-Efficient Scheduling of Nested Parallelism. *ACM Transactions on Programming Languages and Systems* 21 (1999).

Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. 1994. SVR4UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '93)*. Springer-Verlag, London, UK, UK, 41–53.

Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2006. UTS: An Unbalanced Tree Search Benchmark. In *Languages and Compilers for Parallel Computing, 19th International Workshop, LCPC 2006, New Orleans, LA, USA, November 2-4, 2006. Revised Papers*. 235–250.

Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical Memory Management for Parallel Programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 392–406.

Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. 2014a. Parallel Real-Time Scheduling of DAGs. *IEEE Trans. Parallel Distrib. Syst.* 25, 12 (2014), 3242–3252. https://doi.org/10.1109/TPDS.2013.2297919

Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. 2014b. Parallel Real-Time Scheduling of DAGs. *IEEE Trans. Parallel Distrib. Syst.* 25, 12 (2014), 3242–3252. https://doi.org/10.1109/TPDS.2013.2297919

Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. 2013. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems* 49, 4 (01 Jul 2013), 404–435.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2005. *Operating system concepts (7. ed.)*. Wiley.

K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2014. MultiMLton: A multicore-aware runtime for standard ML. *Journal of Functional Programming* FirstView (6 2014), 1–62.

Daniel Spoonhower. 2009. *Scheduling Deterministic Parallel Programs*. Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA.

N. Sturtevant. 2012. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games* 4, 2 (2012), 144 – 148. http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf

Olivier Tardieu, Benjamin Herta, David Cunningham, David Grove, Prabhanjan Kambadur, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri. 2014. X10 and APGAS at Petascale. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. 53–66.

J.D. Ullman. 1975. NP-complete scheduling problems. *J. Comput. System Sci.* 10, 3 (1975), 384 – 393.

Carl A. Waldspurger and William E. Weihl. 1994. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Operating Systems Design and Implementation*. 1–11. citeseer.ist.psu.edu/waldspurger94lottery.html

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)*. 315–316.

Martin Wimmer, Francesco Versaci, Jesper Larsson Träff, Daniel Cederman, and Philippas Tsigas. 2014. Data Structures for Task-based Priority Scheduling. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. 379–380.