Responsive Parallelism with Futures and State

Stefan K. Muller*
smuller@cs.cmu.edu
Carnegie Mellon University, USA

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

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

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

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

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

Abstract

Motivated by the increasing shift to multicore computers, recent work has developed language support for responsive parallel applications that mix compute-intensive tasks with latency-sensitive, usually interactive, tasks. These developments include calculi that allow assigning priorities to threads, type systems that can rule out priority inversions, and accompanying cost models for predicting responsiveness. These advances share one important limitation: all of this work assumes purely functional programming. This is a significant restriction, because many realistic interactive applications, from games to robots to web servers, use mutable state, e.g., for communication between threads.

In this paper, we lift the restriction concerning the use of state. We present λ_i^4, a calculus with implicit parallelism in the form of prioritized futures and mutable state in the form of references. Because both futures and references are first-class values, λ_i^4 programs can exhibit complex dependencies, including interaction between threads and with the external world (users, network, etc). To reason about the responsiveness of λ_i^4 programs, we extend traditional graph-based cost models for parallelism to account for dependencies created via mutable state, and we present a type system to outlaw priority inversions that can lead to unbounded blocking. We show that these techniques are practical by implementing them in C++ and present an empirical evaluation.

CCS Concepts: • **Software and its engineering → Parallel programming languages**; *Imperative languages*; *Concurrent programming languages.*

*These authors contributed equally to this work.

PLDI '20, June 15–20, 2020, London, UK
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386013

Keywords: responsiveness, futures, shared memory, concurrency, parallelism, type systems, Cilk

ACM Reference Format:
Stefan K. Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. 2020. Responsive Parallelism with Futures and State. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3386013

1 Introduction

Advances of the past decade have brought multicore computers to the mainstream. Many computers today, from a credit-card-size Raspberry Pi with four cores to a rack server, are built with multiple processors (cores). These developments have led to increased interest in languages and techniques for writing parallel programs with ***cooperative threading***. In cooperative threading, the user expresses parallelism at a high level and a language-supplied scheduler manages parallelism at run time. Many languages, libraries, and systems for cooperative threading have been developed, including MultiLisp [34], NESL [15], dialects of Cilk [25, 30, 45, 59], OpenMP [52], Fork/Join Java [42], dialects of Habanero [11, 19, 36], TPL [44], TBB [37], X10 [21], parallel ML [29, 32, 53, 61, 66], and parallel Haskell [39, 40].

Cooperative threading is well suited for compute-intensive jobs and may be used to maximize ***throughput*** by finishing a job as quickly as possible. But modern applications also include interactive jobs where a thread may needed to be completed as quickly as possible. For such interactive applications, the main optimization criterion is ***responsiveness*** — how long each thread takes to respond to a user. To meet the demands of such applications, the systems community has developed ***competitive threading*** techniques, which focus on hiding the latency of blocking operations by multiplexing independent sequential threads of control [12, 28, 31, 35].

Historically, collaborative and competitive threading have been researched largely separately. With the mainstream availability of parallel computers, this separation is now obsolete: many jobs today include both compute-intensive tasks and interactive tasks. In fact, applications such as games, browsers, design tools, and all sorts of interesting interactive systems involve both compute-heavy tasks (e.g., graphics, AI, statistics calculations) and interaction. Researchers have therefore started bridging the two worlds. Muller et al. [48, 49, 51] have developed programming-language techniques that allow programmers to write cooperatively threaded programs and also assign priorities to threads, as in competitive threading. By using a type system [49] and a cost model, the authors present techniques for reasoning about the responsiveness of parallel interactive program.

All of this prior work has made some progress on bridging collaborative and competitive threading, but it makes an important assumption: pure functional programming. Specifically, the work does not allow for memory effects, which are crucial for allowing threads to communicate. This restriction can be significant, because nearly all realistic interactive applications rely on mutable state and effects. As an example, consider a basic server consisting of two entities: a high-priority event loop handling queries from a user and a low-priority background thread for optimizing the server's database. Under Muller et al.'s work, the event loop and background thread can only communicate by synchronizing, but such a synchronization would lead to a priority inversion. If effects were allowed, then the threads could communicate by using a piece of shared state.

In this paper, we overcome this restriction by developing programming language support for collaborative and competitive threading in the presence of state. To this end, we consider λ_i^4, a core calculus for an implicitly parallel language with mutable state in the form of references. The parallel portion of the calculus is based on ***futures***, which represent asynchronous computations as first-class values. Futures can be created and synchronized in a very general fashion. The calculus also allows programmers to assign priorities to futures, which represent their computational urgency. Because it combines futures and state, λ_i^4 is very expressive and enables writing conventional nested-parallel programs as well as those with more complex and dynamic dependencies. For example, we can parallelize a dynamic-programming algorithm by creating an initially empty array of future references and then populating the array by creating futures, which may all be executed in parallel. Similarly, we can express rich interactive computations, e.g., a network event can be delegated to a future that sends asynchronous status updates via a piece of shared state.

The high degree of expressiveness in λ_i^4 makes it tricky to reason about the cost due to priority inversions and nondeterminism due to scheduling: because of the presence of state, the computation may depend on scheduling decisions. For these reasons, traditional graph-based cost models of parallel computations [13, 14, 62] do not apply to programs that mix futures and state. Such models typically do not take priorities into account and assume that scheduling does not change the computation graph.

We tackle these challenges by using a combination of algorithmic and formal techniques. On the algorithmic side, we extend traditional graph-based cost models to include information about priorities as well as "happens-before" edges that capture certain dependencies by reifying execution-dependent information flow through mutable state. We then prove that if a computation graph has no priority inversions, then it can be scheduled by using an extension of greedy scheduling with priorities to obtain provable bounds on the response time of any thread (Theorem 2.1). Priority inversions are not simple to reason about, so we present a type system for λ_i^4 that guarantees that any well-typed program has no priority inversions. To establish the soundness of the type system (Theorem 3.1), we model the structure of the computation by giving a dynamic semantics that, in addition to evaluating the program, creates a computation graph that captures both traditional dependencies between threads and also non-traditionally captures certain happens-before dependencies to model the impact of mutable state.

Because λ_i^4 is a formal system, it can in principle be implemented in many different languages. For this paper, we chose to implement such a system in the context of C/C++ because many real-world interactive applications with stringent performance requirements are written in C/C++. Specifically, we have developed I-Cilk, a task parallel platform that supports interactive parallel applications. I-Cilk is based on Cilk, a parallel dialect of C/C++. As with traditional cooperative threading systems, I-Cilk consists of a runtime scheduler that dynamically creates threads and maps them onto available processing cores. Unlike traditional task-parallel platforms, however, I-Cilk supports competitive threading by allowing the programmer to specify priorities of tasks. Perhaps somewhat unexpectedly, I-Cilk also includes an implementation of the λ_i^4 type system to rule out priority inversions. The type system is implemented by using inheritance, template programming, and other features of C++ to encode the restrictions necessary to prevent priority inversions. Because C++ is not a safe language, this implementation of the type system expects the programmer to obey certain conventions.

The thread scheduler of I-Cilk aims to implement the scheduling principle that Theorem 2.1 relies on. This is challenging to do efficiently because it requires maintaining global information within the scheduler that can only be achieved via frequent synchronizations. Instead, I-Cilk approximates optimal scheduling by utilizing a two-level adaptive scheduling strategy that re-evaluates the scheduling decision at a fixed scheduling quantum.

We empirically evaluate I-Cilk using three moderately-sized application benchmarks (about 1K lines each). These applications fully utilize the features of I-Cilk (including I/O and prioritization of tasks). We will dive into one application in detail to illustrate the use of future references and mutable states. To demonstrate the efficiency of I-Cilk, we compare the response times and execution times of tasks at different priority levels running on I-Cilk and on a baseline system that behaves like I-Cilk but does not account for priority. Empirically we demonstrate that indeed I-Cilk provides much better response time, illustrating the efficacy of its scheduler.

In summary, the contributions of this paper include:

- a cost model for imperative parallel programs that incorporates scheduler-dependence through mutable state (Section 2);
- a calculus λ_i^4 for imperative parallel programs, equipped with a type system that guarantees absence of priority inversions (Section 3);
- I-Cilk, a C/C++-based task parallel platform that supports interactive parallel applications with a type system and scheduler that embody the ideas of the threading model, type system, and cost model of λ_i^4 (Section 4); and
- an empirical evaluation of I-Cilk using three large case studies written with I-Cilk (Section 5).

2 A DAG Model for Responsiveness

2.1 Preliminaries

For the purpose of this paper, we will consider programs with first-class threads that implement futures. Because our models and scheduling algorithms are largely independent of the language mechanisms by which threads are created, we will simply refer to "threads" here. We assign threads a priority, written ρ, drawn from a partially ordered set R, where $\rho_1 \preceq \rho_2$ means that priority ρ_1 is lower than priority ρ_2 or $\rho_1 = \rho_2$. We write $\rho_1 \prec \rho_2$ for the strict partial-order relation that does not allow for reflexivity. Note that a total order is a partial order by definition and threads can be given priorities from a totally ordered set, e.g., integers.

Threads interact with each other in two ways. First, a thread a may ***create*** a thread b, after which the two threads run in parallel. We call this operation, which returns a handle to b, "future-create" or simply `fcreate`. Second, a thread a may wait for a thread b to complete before proceeding. We call this operation "future-touch" or `ftouch`. This model subsumes the classic fork-join (spawn-sync) parallelism.

As is traditionally done, we can represent the execution of a parallel program with a ***Directed Acyclic Graph*** or a ***DAG***. A vertex of the DAG represents an operation (without loss of generality, we will assume that a single vertex represents a uniform unit of computation time, such as a processing core cycle). A directed edge from u to u', written (u, u'), indicates that the operation represented by u' depends on the operation represented by u. We write $u \sqsupseteq u'$ to mean that u is an ***ancestor*** of u', i.e., there is a (directed) path from u to u' (it may be that $u = u'$). If it is the case that $u \not\sqsupseteq u'$ and $u' \not\sqsupseteq u$, then u and u' may run in parallel.

A ***schedule*** of a DAG is an assignment of vertices to processing cores at each time step during the execution of a parallel program. Schedules must obey the dependences in the DAG: a vertex may only be assigned to a core if it is ***ready***,

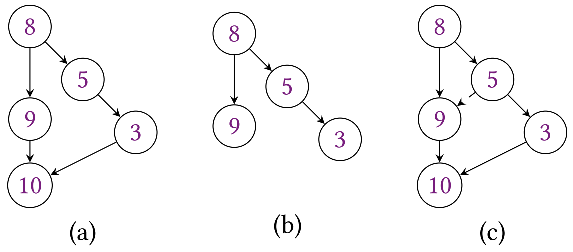


Figure 1. DAGs in which the main thread reads a valid thread handle (a) and NULL (b), and a DAG with a weak edge representing a read of a valid thread handle (c). Vertices are labeled with the line of code they represent and threads are arranged in columns.

that is, if all of its (proper) ancestors have been assigned on prior time steps. The goal of an efficient scheduler for parallel programs is to construct as short a schedule as possible. Constructing an optimal schedule is impossible when, as in many real programs, the DAG unfolds dynamically during execution and is not known ahead of time (even a relaxed ***offline*** version of the problem in which the DAG is known ahead of time is NP-hard [63]). However, prior results have shown that schedules obeying certain scheduling principles are within a constant factor of optimal length while making decisions based only on information available online (i.e., they need only know the set of ready vertices at any point in time). One such scheduling principle for DAGs with priorities is ***prompt scheduling***. At each time step, a prompt schedule assigns to a core a ready vertex u such that no currently unassigned vertex is higher-priority than u repeatedly until no cores remain or no ready vertices remain.

2.2 Weak Edges

Traditionally, cost models for parallel programs assume that scheduling does not change the DAG of a parallel computation. This assumption is reasonable for deterministic programs and provides a nice layer of abstraction over scheduling — we can assume that any schedule of a DAG corresponds to a valid execution. This fundamental assumption breaks in our setting where threads are first class values and state can be used to communicate in an unstructured fashion, leading to determinacy races.

Consider as an example the following program.

```
1  thread t = NULL;             7  void main() {
2                               8    fcreate (f);
3  void g() {}                  9    if (t != NULL) {
4  void f() {                   10     ftouch (t);
5    t = fcreate (g);           11   }
6  }                            12 }
```

The DAG for this program, in particular whether there is an edge from `g` to `main` representing the `ftouch` on line 10, depends crucially on whether `f` performs the `fcreate` and assignment to `t` before `main` reads `t`, that is, on whether the conditional on line 9 returns true or false. In fact, depending on the outcome of the condition, this program gives rise to one of two DAGs, both shown in Figure 1: one in which the conditional is true and one in which it is false. Applying the traditional separation between DAGs and schedules, given DAG (a), the scheduler could execute the vertices in the following order: 8, 9, 5, 3, 10. But under this schedule, the read on line 9 should read `NULL`, and thus line 10 should not be executed at all! Similarly, the scheduler could execute DAG (b) in the order 8, 5, 3, 9, in which case the read would read a valid thread handle.

The issue is that each DAG is valid for only certain schedules but not all. To encode this information, we extend the traditional notion of DAGs with a new type of edge we call a ***weak edge***. A weak edge from u to u' records the fact that the given DAG makes sense only for schedules where u is executed before u'. We call such a schedule ***admissible***. As an example, DAG (c) of Figure 1 includes a weak edge (shown as a dotted line) from 5 to 9. The schedule 8, 5, 9, 3, 10 is an admissible schedule of DAG (c), but 8, 9, 5, 3, 10 is not.

At first sight, the reader may feel that we can replace a weak edge with an ordinary (***strong***) edge. This is not quite correct, as strong and weak edges are treated differently in determining whether a schedule is prompt. Recall that a schedule is prompt if it assigns ready vertices in priority order. In the presence of weak edges, we define a vertex u to be ready when all of its ***strong parents***, that is, vertices u' such that there exists a strong edge (u', u), have executed.

Consider again DAG (c) from Figure 1, but now suppose we wish to construct a prompt schedule on two cores. By the above definition, a prompt schedule must execute vertex 8, followed by 5 and 9 in parallel, followed by 3, followed by 10. This is, in fact, the only prompt schedule of DAG (c), but it is not admissible because it does not execute 5 before 9. We thus conclude that there are no prompt admissible schedules of DAG (c) on two cores and DAG (b) is the only valid DAG for a two-core execution of this program (as DAG (b) has no weak edges, any prompt schedule of it is admissible). If we were to replace the weak edge (5, 9) with a strong edge, there would be a prompt schedule of DAG (c) that executes 8, followed by 5, followed by 9 and 3, followed by 10. As always, a strong edge forces vertex 9 to wait for vertex 5, but this violates the intended semantics of the program as a simple read operation should not have to block waiting for a write.

In summary, strong edges determine what schedules are valid for a given DAG, while weak edges determine whether a DAG is valid for a given schedule. That is, weak edges internalize information about schedules into the DAG, breaking what would otherwise be a circular dependency between constructing a DAG and constructing a schedule of it.

We extend the notions of ancestors and paths to distinguish between weak and strong edges. We say that a path is ***strong*** if it contains no weak edges. If $u \sqsupseteq u'$ and all paths from u to u' are strong, then we say that u is a ***strong ancestor*** of u' and write $u \sqsupseteq^s u'$. On the other hand, if there exists a weak path (i.e., a path with a weak edge) from u to u', we say u is a ***weak ancestor*** of u' and write $u \sqsupseteq^w u'$. We will continue to drop the superscript if it not important whether u is a weak or strong ancestor.

In formal notation, we represent a DAG g as a quadruple $(\mathcal{T}, E^c, E^t, E^w)$. The first component of the quadruple is a mapping from thread symbols, for which we will use the metavariables a, b and variants, to a pair of that thread's priority and the vertices it comprises. We use the notation $\vec{u}$ for a sequence of vertices $u_1 \cdot \ldots \cdot u_n$ making up a thread, and write $[]$ when $n = 0$. Such a sequence implies that g contains the edges $(u_1, u_2), \ldots (u_{n-1}, u_n)$. We will refer to such edges as ***continuation edges***. For a thread with priority ρ and vertices $\vec{u}$, we write $a \underset{\rho}{\hookrightarrow} \vec{u} \in \mathcal{T}$. We write $Prio_g(u)$ to refer to the priority of the thread containing vertex u in g.

The remaining three components are sets of edges. The set E^c contains **`fcreate`** ***edges*** (u, a) indicating that vertex u creates thread a. It is shorthand for (u, s) where s is the first vertex of a. The set E^t contains **`ftouch`** ***edges*** (a, u) indicating that vertex u touches thread a. It is shorthand for (t, u) where t is the last vertex of a. Finally, the set E^w contains weak edges.

2.3 Well-Formedness and Response Time

Our goal is to bound the ***response time*** $T(a)$ of a thread a in a DAG. If $a \underset{\rho}{\hookrightarrow} s \cdot \ldots \cdot t \in g$, for a particular schedule of g, we define $T(a)$ to be the number of time steps between when s becomes ready and when t is executed, inclusive.

Intuitively, in a well-designed program and an appropriate schedule, if thread a has priority ρ, its response time should depend only on parts of the graph that may happen in parallel with a (i.e. are not ancestors or descendants of a) and have priority not less than ρ. This is known as the ***competitor work*** $W_{\not\prec\rho}(\updownarrow a)$ of a thread a and is defined formally:

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

We must also define a metric corresponding to the critical path of a. We will call this metric the a-span, because it corresponds to the traditional notion of span in a parallel cost DAG, but we will defer its formal definition for now, because we will need other definitions first.

Bounding the response time of a in terms of only the competitor work and a-span is not possible for all DAGs: if a depends on lower-priority code along its critical path, this code must be included in the response time of a. This situation essentially corresponds to the well-known idea of a ***priority inversion***. Our response time bound guarantees efficient scheduling of any DAG that is ***well-formed***, that is, free of this type of priority inversion. Well-formedness must, at a minimum, require that no `ftouch` edges go from lower- to higher-priority threads. This requirement is formalized

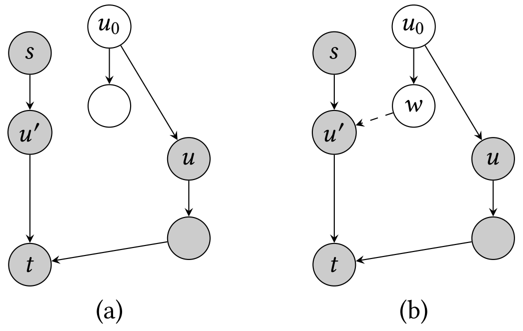


Figure 2. (a) a DAG that is not well-formed because of the strong path from u_0 to t (b) a well-formed version of the DAG with a weak path from u_0 to t.

in the first bullet point of Definition 1. There is another, more subtle, way in which priority inversions could arise. Consider the DAG in Figure 2(a), in which shaded vertices represent high-priority work. Although no `ftouch` edges violate the first requirement of the definition, it would be possible, in a prompt schedule of the DAG, for high-priority vertex t to be delayed indefinitely waiting for low-priority vertex u_0 to execute due to the chain of strong dependences through u. Note that the problem is not that u depends on a lower-priority vertex—as this is a `fcreate` edge, such a dependence is allowed. The issue is that u's thread is then `ftouch`ed by t with no other dependence relation between u_0 and t. The second bullet point of Definition 1 requires that, in such a situation, this dependence be mitigated by, e.g., the weak edge added in Figure 2(b).

We note that this second requirement actually places no additional restrictions on programs. DAGs such as the one in Figure 2(a) could not arise from real programs because in order for t to `ftouch` u's thread, it must have access to its thread handle, which will have been returned by the `fcreate` call represented by u_0. This thread handle must be propagated to t through a chain of dependences including at least one dependence through memory effects. There must therefore be a weak path from u_0 to t, as in DAG 2(b), which reflects a write (w) of the thread handle followed by a read (u').

Definition 1 formalizes the above intuitions.

Definition 1. A DAG $g = (\mathcal{T}, E^c, E^t, E^w)$ is ***well-formed*** if for all threads $a \underset{\rho}{\hookrightarrow} s \cdot \ldots \cdot t \in \mathcal{T}$,

- For all $u \in g$, if $u \sqsupseteq^s t$ and $u \not\sqsupseteq s$, then $\rho \preceq Prio_g(u)$.
- For all strong edges (u_0, u) such that $u \sqsupseteq^s t$ and $u_0 \not\sqsupseteq s$ and $Prio_g(u) \not\preceq Prio_g(u_0)$, there exists u' such that $u_0 \sqsupseteq^w u' \sqsupseteq^s t$ and $u \not\sqsupseteq u'$.

To a first approximation, we may define the a-span of a thread $s \cdot \ldots \cdot t$ as the longest path ending at t consisting of non-ancestors of s (i.e., the longest chain of vertices that might delay the completion of a). In the presence of weak edges, however, the definition is not so simple. Consider the DAG on the left of Figure 3, in which shaded nodes are high-priority. Under the above definition, the a-span includes low-priority node u_0, but in any admissible schedule, u' runs after u_0, so u_0 is not actually on the critical path. We thus transform the DAG into one, the ***strengthening***, that reflects this implicit dependence.

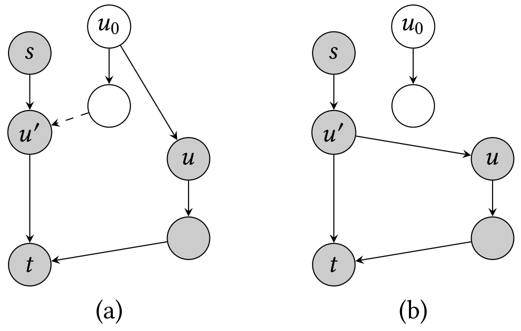


Figure 3. (a) a DAG; (b) its strengthening

Definition 2. Let g be a well-formed DAG with a thread $a \underset{\rho_a}{\hookrightarrow} s \cdot \ldots \cdot t$. We derive the *a-**strengthening***, written $\hat{g}_a$, from g as follows. For every strong edge (u_0, u) such that $u \sqsupseteq^s t$ and $Prio_g(u) \not\preceq Prio_g(u_0)$ and $u \not\sqsupseteq s$,

- Remove the edge (u_0, u).
- Let $u' \in g$ such that $u' \sqsupseteq^s t$ and $u_0 \sqsupseteq^w u'$. If $u' \not\sqsupseteq s$, then add the edge (u', u) in place of the weak edge between u_0 and u.

The strengthening of the example DAG is shown in the right side of the figure. For a thread $a \underset{\rho}{\hookrightarrow} s \cdot \ldots \cdot t \in g$, we define the a-span, written $S_a(\ddagger a)$, to be the length of the longest path in $\hat{g}_a$ ending at t consisting only of vertices that are not ancestors of s. More generally, we write $S_a(V)$ to be the length of the longest path in $\hat{g}_a$ ending at t consisting only of vertices in V. Intuitively, the a-span corresponds to the critical path of a because, in a valid and admissible schedule, it is possible that all of the vertices along this path may need to be executed sequentially while a is being executed.

Theorem 2.1 gives a bound on the response times of threads in admissible, prompt schedules of well-formed DAGs. The intuitive explanation of the bound also gives a sketch of the proof: at every time step, such a schedule is doing one of two types of work: (1) executing P vertices of competitor work or (2) executing all available vertices on the a-span. The amount of work of type (1) to be done is bounded by the competitor work divided by P. Work of type (2) can only be done during $S_a(\ddagger a)$ time steps, during which $P - 1$ of the P cores might be idle. Adding these amounts of work together gives the bound on response time.

Theorem 2.1. *Let g be a well-formed DAG and let a be a thread of priority ρ in g. For any admissible prompt schedule on P processing cores,*

$$T(a) \leq \frac{1}{P}\left[W_{\not\prec\rho}(\ddagger a) + (P-1)S_a(\ddagger a)\right]$$

Constraints	C	$::=$	$\rho \leq \rho \mid C \wedge C$
Types	τ	$::=$	$\mathtt{unit} \mid \mathtt{nat} \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau + \tau$
		$\mid$	$\tau\ \mathtt{ref} \mid \tau\ \mathtt{thread}\,[\rho] \mid \tau\ \mathtt{cmd}[\rho]$
Values	v	$::=$	$x \mid \langle\rangle \mid \overline{n} \mid \lambda x.e \mid (v, v) \mid \mathtt{inl}\ v \mid \mathtt{inr}\ v$
		$\mid$	$\mathtt{ref}[s] \mid \mathtt{tid}[a] \mid \mathtt{cmd}[\rho]\ \{m\}$
Expressions	e	$::=$	$v \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid \mathtt{ifz}\ v\ \{e; x.e\}$
		$\mid$	$v\ v \mid \mathtt{fst}\ v \mid \mathtt{snd}\ v$
		$\mid$	$\mathtt{case}\ v\ \{x.e; y.e\} \mid \mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e$
Commands	m	$::=$	$\mathtt{fcreate}[\rho;\tau]\{m\} \mid \mathtt{ftouch}\ e$
		$\mid$	$\mathtt{dcl}\ [\tau]\ s := e\ \mathtt{in}\ m$
		$\mid$	$!e \mid e := e \mid x \leftarrow e; m \mid \mathtt{ret}\ e$

Figure 4. Syntax of λ^4_i

$$\frac{\Gamma \vdash^R_\Sigma m \div \tau@\rho'}{\Gamma \vdash^R_\Sigma \mathtt{fcreate}[\rho';\tau]\{m\} \div \tau\ \mathtt{thread}\,[\rho']@\rho}\ (\textsc{Create})$$

$$\frac{\Gamma \vdash^R_\Sigma e : \tau\ \mathtt{thread}\,[\rho'] \qquad \Gamma \vdash^R \rho \leq \rho'}{\Gamma \vdash^R_\Sigma \mathtt{ftouch}\ e \div \tau@\rho}\ (\textsc{Touch})$$

$$\frac{\Gamma \vdash^R_\Sigma e : \tau \qquad \Gamma \vdash^R_{\Sigma, s\sim\tau} m \div \tau'@\rho}{\Gamma \vdash^R_\Sigma \mathtt{dcl}\ [\tau]\ s := e\ \mathtt{in}\ m \div \tau'@\rho}\ (\textsc{Dcl})$$

$$\frac{\Gamma \vdash^R_\Sigma e : \tau\ \mathtt{ref}}{\Gamma \vdash^R_\Sigma !e \div \tau@\rho}\ (\textsc{Get}) \qquad \frac{\Gamma \vdash^R_\Sigma e_1 : \tau\ \mathtt{ref} \qquad \Gamma \vdash^R_\Sigma e_2 : \tau}{\Gamma \vdash^R_\Sigma e_1 := e_2 \div \tau@\rho}\ (\textsc{Set})$$

Figure 5. Selected command typing rules.

3 Type System for Responsiveness

We describe a type system that can be used to ensure that a program results in a well-formed cost graph, by way of a core calculus λ^4_i, which extends λ^4 [49], with the key addition of mutable references (memory locations). Section 3.1 presents the calculus and type system. Section 3.2 equips λ^4_i with a ***cost semantics*** that evaluates a λ^4_i program to produce a cost graph of the form described in Section 2. We prove that, for a well-typed program, the resulting graph is well-formed, and thus the program is free of priority inversions.

3.1 The λ^4_i Core Calculus

The syntax of λ^4_i is shown in Figure 4, in A-normal form (for most expressions, any subexpressions that are not under binders are values; computations can be sequenced using let-bindings). We differentiate between ***expressions***, language constructs that do not depend on the state of memory or threads, and ***commands***, which do.

The non-standard types of λ^4_i are a type τ `ref` indicating references to memory locations holding values of type τ; a type τ `thread` $[\rho]$ representing handles to running threads of type τ at priority ρ and a type τ `cmd`$[\rho]$ representing encapsulated commands which run at priority ρ and have return type τ. Priorities are drawn from a given fixed (partially ordered) set R.

The novel values of the calculus are references `ref`$[s]$, which allow access to a memory location s; thread handles `tid`$[a]$, which reference a running thread referred to by a; and `cmd`$[\rho]$ $\{m\}$, which encapsulates the command m at priority ρ. The expression layer is otherwise standard.

Commands include operations to manipulate threads and state, including commands to create and touch threads[1] The command `dcl` $[\tau]$ $s := e$ `in` m declares a new mutable memory location s, initialized with the expression e, in the scope of m. The read command $!e$ evaluates e to a reference `ref`$[s]$ and returns the current contents of s. The assignment command $e_1 := e_2$ evaluates e_1 to a reference `ref`$[s]$ and writes the value of e_2 to s; the command also returns the new value.

Commands are sequenced with an operator $x \leftarrow e; m$, which evaluates e to an encapsulated command, executes the command, binds its return value to x and continues as m. Expressions may be embedded into the command layer using the command `ret` e which evaluates e and returns its value. These commands may be thought of as the monadic bind and return operators, respectively.

Figure 5 shows the key rules of the type system for λ^4_i, namely the rules for threads and references. Due to space constraints, we omit more standard features and present them in the full version [50].

The command typing rules in the figure define the judgment $\Gamma \vdash^R_\Sigma m \div \tau@\rho$. The signature Σ tracks type information for threads and memory locations, as well as the priorities of threads. The typing judgment is also parameterized by a partially-ordered set R of priorities and a typing context Γ. The context Γ, as usual, contains premises of the form $x : \tau$, indicating that the variable x has type τ. In addition to the return type τ of the command, the typing judgment indicates that the command may run at priority ρ. The rules Create and Touch contain notable features relating to priorities. In particular, Touch requires that e be a handle to a thread running at priority ρ' and that this priority be higher than or equal to the priority ρ of the current thread. It is this requirement that prevents priority inversion. The Create rule requires that a command run in a new thread at priority ρ' indeed be able to run at priority ρ'. Note, however, that the `fcreate` command itself may run at any priority; the language does not enforce any priority relationship between a thread and its parent. We refer the reader to the presentation of λ^4 [49] for a more thorough description of these rules.

We describe the rules for allocating and accessing references in more detail. Rule Dcl types the initialization expression e at type τ and introduces a new location s in typing m. Rule Get requires that its subexpression have reference type.

[1]while the syntax for `fcreate` and `ftouch` is drawn from the fact that our threading model is based on **f**utures, we simply use the term "threads" to refer to running asynchronous threads of control and "thread handles" to refer to the first-class values that refer to threads. This avoids some terminological confusion frequently associated with futures.

$$
\begin{array}{llll}
\textit{Frames} & f & ::= & \texttt{let}\ x = - \ \texttt{in}\ e \mid x \leftarrow -; m \mid \texttt{ftouch}\ - \\
 & & \mid & \texttt{dcl}\,[\tau]\ s := - \ \texttt{in}\ m \mid !- \mid - := e \\
 & & \mid & v := - \mid \texttt{ret}\ - \\
\textit{Stacks} & k & ::= & \epsilon \mid k; f \\
\textit{States} & K & ::= & k \rhd e \mid k \lhd v \mid k \blacktriangleright m \mid k \blacktriangleleft \texttt{ret}\ v
\end{array}
$$

Figure 6. Stack, frame and state syntax.

Rule Set requires that e_1 have type τ `ref` and that e_2 have type τ. Note that this requires memory locations to have a consistent type throughout execution. The return type of an assignment to a τ reference is τ. All of these commands may type at any priority as state operations and priorities are orthogonal.

The judgment $\Gamma \vdash^R C$ indicates that the premises contained in Γ entail the priority constraints C. The rules (which follow standard rules of logic) are found in the full version [50]

3.2 Cost Semantics and Time Bounds

In this section, we equip λ^4_i with a small-step dynamic semantics that tracks two notions of cost. First, in a straightforward sense, the number of steps taken by the semantics to execute a program gives an abstract measure of execution time. Second, we equip the dynamic semantics to construct a cost graph for the program that captures the parallelism opportunities in the execution, and also uses weak edges to record happens-before relations as described in Section 2.

We present the dynamic semantics of λ^4_i as a stack-based parallel abstract machine that serves as a rough model of the program's execution time on realistic parallel hardware. A *stack* k consists of a sequence of *stack frames* f (or is the empty stack, ϵ). Each frame is a command or expression with a hole, written $-$, to be filled with the result of the next frame. The stack thus represents the continuation of the current computation. At each step, each thread active in the machine is either executing a command or expression from the top of the stack ("popping") or returning a resulting value to the stack ("pushing"). These states are represented by $k \rhd e$ and $k \lhd v$, respectively, for expressions (and similar syntax with filled triangles for commands). The syntax of stack frames, stacks, and stack states is given in Figure 6.

A full configuration of the stack machine includes the current heap σ and set of threads μ. A heap, essentially, is a mapping from memory locations to values. For technical reasons, we also record two pieces of metadata in the heap at each location: the DAG vertex that performed the last write to that memory location (which will be used to add weak edges to the cost graph) and a signature containing threads that one might "learn about" by reading this memory location. For example, suppose thread a creates thread b and writes `tid[`b`]` into a memory location s. If thread c later reads from s, it must "learn about" the existence of thread b in order to preserve typing. We write an element of the heap as $s \mapsto (v, u, \Sigma)$. We denote the empty heap $\emptyset$, and let $\sigma[s \mapsto (v, u, \Sigma)]$ be the extension of σ with the binding $s \mapsto (v, u, \Sigma)$. If $s \in \mathit{dom}(\sigma)$, the new binding is assumed to overwrite the existing binding. A *thread pool* μ maps thread symbols a to a triple consisting of thread a's priority, its stack state and a signature Σ consisting of the threads that a "knows about," as motivated above.

Figure 7 presents a subset of the rules for the command transition judgment

$$\sigma \mid \mu \otimes a_i \xhookrightarrow[\rho_i;\Sigma_i]{} K_i \Rightarrow a_i \xhookrightarrow[\rho_i;\Sigma'_i]{} K'_i \otimes \mu'_i \mid \Sigma''_i \mid \sigma'_i \mid g'_i$$

In this judgment, K'_i is the new state of thread a_i, Σ''_i contains the memory locations allocated by the step and σ'_i contains any heap writes performed by the step. The graph g'_i contains a vertex corresponding to this step as well as any additional `fcreate`, `ftouch` or weak edges added by this step. The full semantics for the abstract machine also includes a single rule that steps some number of threads in parallel and combines the resulting states and graphs. The full set of rules can be found in the full version [50]

An `fcreate` command simply creates a new thread symbol b and adds a thread b to the thread pool to execute the command m. It returns the thread handle, and adds a `fcreate` edge to the graph. An `ftouch` command first evaluates its subexpression (D-Touch1). When the thread handle `tid[`b`]` is returned, rule D-Touch2 inspects the thread pool for the entry $b \xhookrightarrow[\rho';\Sigma']{} \epsilon \blacktriangleleft \texttt{ret}\ v$ (if b's stack is not of this form, b has not finished executing and the `ftouch` will block until it does). The command returns the value v and adds the appropriate `ftouch` edge. It also adds Σ' to the set of threads that a "knows about," because v might contain handles to threads in Σ'.

Rule D-Set3 adds a binding to the heap for the new value of the memory location, and includes as metadata the new graph vertex u and the signature Σ. Rule D-Get2 inspects the heap for the binding of s, returns its value, adds a weak edge (u', u) (recall that u' is the vertex corresponding to most recent write to s) and adds Σ' to the signature of a.

The soundness theorem for the type system states that well-typed programs have well-formed cost graphs.

Theorem 3.1. *Let m be such that $\cdot \vdash^R_\cdot m \mathrel{\dot{\sim}} \tau@\rho$. If*

$$\cdot \mid \emptyset \mid \emptyset \mid a \xhookrightarrow[\rho;\cdot]{} \epsilon \rhd m \Rightarrow^* \Sigma \mid \sigma \mid g \mid \mu$$

then g is well-formed and acyclic.

The proof of this theorem consists of showing that all steps maintain two invariants:

1. No strong edges go from lower to higher priority
2. Σ correctly reflects the "knows about" relation motivated above.

These invariants respectively imply the two well-formedness requirements of Section 2. Full proof details are available in the full version [50].

$$\frac{u\ \texttt{fresh} \qquad b\ \texttt{fresh}}{\sigma \mid \mu \otimes a \xhookrightarrow[\rho;\Sigma]{} k \blacktriangleright \texttt{fcreate}[\rho';\tau]\{m\} \Rightarrow a \xhookrightarrow[\rho;\Sigma,b\sim\tau@\rho']{} k \blacktriangleleft \texttt{ret tid}[b] \otimes b \xhookrightarrow[\rho';\Sigma]{} \epsilon \blacktriangleright m \mid \cdot \mid \sigma \mid (a \underset{\rho}{\hookrightarrow} u, \{(u,b)\}, \emptyset, \emptyset)} \text{(D-Create)}$$

$$\frac{u\ \texttt{fresh}}{\begin{array}{l}\sigma \mid \mu \otimes b \xhookrightarrow[\rho';\Sigma']{} \epsilon \blacktriangleleft \texttt{ret}\ v \otimes a \xhookrightarrow[\rho;\Sigma,b\sim\rho'@\tau']{} k; \texttt{ftouch} - \lhd \texttt{tid}[b] \\ \Rightarrow a \xhookrightarrow[\rho;\Sigma,b\sim\rho'@\tau',\Sigma']{} k \blacktriangleleft \texttt{ret}\ v \otimes \emptyset \mid \cdot \mid \sigma \mid (a \underset{\rho}{\hookrightarrow} u, \emptyset, \{(b,u)\}, \emptyset)\end{array}} \text{(D-Touch2)}$$

$$\frac{u\ \texttt{fresh} \qquad \sigma(s) = (v, u', \Sigma')}{\sigma \mid \mu \otimes a \xhookrightarrow[\rho;\Sigma,s\sim\tau']{} k; !- \lhd \texttt{ref}[s] \Rightarrow a \xhookrightarrow[\rho;\Sigma,s\sim\tau',\Sigma']{} k \blacktriangleleft \texttt{ret}\ v \otimes \emptyset \mid \cdot \mid \sigma \mid (a \underset{\rho}{\hookrightarrow} u, \emptyset, \emptyset, \{(u',u)\})} \text{(D-Get2)}$$

$$\frac{u\ \texttt{fresh}}{\sigma \mid \mu \otimes a \xhookrightarrow[\rho;\Sigma,s\sim\tau']{} k; \texttt{ref}[s] := - \lhd v \Rightarrow a \xhookrightarrow[\rho;\Sigma,s\sim\tau']{} k \blacktriangleleft \texttt{ret}\ v \otimes \emptyset \mid \cdot \mid \sigma[s \mapsto (v,u,\Sigma)] \mid (a \underset{\rho}{\hookrightarrow} u, \emptyset, \emptyset, \emptyset)} \text{(D-Set3)}$$

Figure 7. Cost semantics for threads

4 Implementation of I-Cilk

This section presents the design and implementation of I-Cilk, our prototype task-parallel platform that supports parallel interactive applications. I-Cilk is based on an open-source implementation [60] of Cilk (a parallel dialect of C/C++) called Cilk-F [59] that extends Cilk with support for futures. The implementation of I-Cilk consists of two main components, a type system to rule out priority inversions (closely following the typing rules discussed in Section 3) and a runtime scheduler that automates load balancing while prioritizing high-priority tasks over lower-priority ones.

4.1 Programming Interface

Thread creation. In I-Cilk, like in λ^4_i, a function f can invoke another function g with `fcreate`, which indicates that the execution of g is logically in parallel with the continuation of f after `fcreate`. A function invocation prefixed with `fcreate` returns a handle to the new thread, on which one can later invoke `ftouch` to ensure that the thread terminates before the control passes beyond the `ftouch` statement. Since a thread handle can be stored in a data structure or global variable and retrieved later, the use of `fcreate` and `ftouch` can generate irregular parallelism with arbitrary dependences. In I-Cilk, as is common in C-like languages, it is possible to allocate a variable of thread handle type without associating it to a thread, and later pass this variable by reference to `fcreate`, to associate it with the created thread. This is in contrast to λ^4_i, where the allocation of the handle and the creation of the thread happen simultaneously. [2]

I/O Operations. I-Cilk supports the use of I/O operations via a special type of thread, called an `io_future`, that performs an I/O operation in a latency-hiding way. Specifically, I-Cilk provides special versions of the `cilk_read` and `cilk_write` functions that behave similarly to the Linux `read` and `write` except that they return a `io_future` reference representing the I/O operation. Upon invocation, `cilk_read` and `cilk_write` create a thread to perform the I/O without occupying the processor, and then the returned `io_future` can be used to wait on the I/O by calling `ftouch` on it.

[2]I-Cilk additionally supports `spawn` and `sync` for nested parallelism. The use of `spawn` and `sync` can be subsumed by `fcreate` and `ftouch` from the type checking perspective and hence we omit the discussion here.

4.2 Type System

The type system in I-Cilk does not provide full type safety guarantees, as C++ is not type safe. Nevertheless, provided that the programmer follows a set of simple rules, the C++-based type system can ensure that a program that type checks will result in strongly well-formed DAGs when executed. The type system enables us to type check moderately large benchmarks that implement interesting functionalities involving the use of low-level system calls and concurrent data structures (discussed in Section 5.1).

Enforcing Typing Rules. We utilize templates and other C++11 language features to encode the type system. In the C++ encoding, each priority is represented as a `class`. The relationship between two priorities is captured through the class hierarchy via inheritance; if priority ρ inherits from priority ρ' or some descendant of ρ', then $\rho \succ \rho'$ (i.e., ρ has higher priority than ρ'). Such relationships can be tested at compile time using `is_base_of`, which tests whether one `class` is either the same as or the ancestor of another. Unlike in λ^4_i, priorities are thus user-defined types rather than a pre-defined set of constants.

In λ^4_i, there is a separation between the command layer and expression layer. In I-Cilk, the separation is not as clear. However, we must enforce restrictions on which functions can be invoked with `fcreate` (generating a handle that can be `ftouch`ed later) and which function can execute `ftouch`, because the priority of such functions must be retrievable at compile time in order to enforce the typing rules. We require these functions to be wrapped in a `command class` whose type relies on a template that specifies its execution priority. For ease of discussion, we will refer to such a function as a `command` function. Unlike in λ_i^4, `fcreate` is not a command — code at any priority may safely invoke a function with `fcreate`; this causes no difficulties in enforcing the typing guarantees. Also unlike in λ_i^4, code in I-Cilk does not require special syntax for invoking an expression (e.g., function that is not a `command`) within a command.

The encoding of the type system is realized by C++ macros that transform `fcreate`, `ftouch`, and declarations / invocations of command functions into the necessary C++ encodings.[3] The templated types of `command` functions allow their priority to be known at compile time, and the type system checks for priority inversion at the execution of `ftouch`. First, a function invoked with `fcreate` (which must be a `command` function) returns a thread handle whose type is templated with its priority and return type (i.e., what its corresponding thread returns when done executing, which may be void). Second, an `ftouch` can only be executed from within a `command` function, and `ftouch` on a thread handle `fptr` is translated to:

```
fptr->touch();
static_assert(is_base_of<this->Priority,
                         fptr->Priority>::value,
  "ERROR:␣priority␣inversion␣on␣future␣touch");
```

The static assert ensures that the thread invoking the `ftouch` has priority lower than or equal to that of the thread whose handle is `ftouch`ed, causing a compiler error otherwise.

Lastly, we enforce that a `command` function g, if invoked by another `command` function f, must be invoked with `fcreate` or inherits the priority of f.[4] Doing so ensures that another `command` function h joining with f (with lower priority than f but higher priority than g) does not suffer from priority inversion by waiting on g. In λ_i^4 such an issue does not arise because call is an expression whereas `fcreate` is a command, and therefore the two do not mix. This issue is an artifact of the fact that the distinction between the command and the expression is not clear in I-Cilk.

Discussion: Type Safety. Ideally we would like to guarantee that programs which type check using our API will always generate strongly well-formed DAGs when executed. However, we cannot make this guarantee in full because C++ is not a type-safe language. Nevertheless, provided that the programmer follows a few simple rules, our type system can statically prevent cases of priority inversions, and a program that type checks will result in strongly well-formed DAGs when executed.

The first rule is that the programmer should not use unsafe type casts, which circumvent the type system and allow the programmer to modify priority types in ways that the type system cannot detect.

The second rule is that the programmer should always ensure that a thread handle is already associated with a thread (via `fcreate`) before invoking `ftouch` on it. This rule is important because a strongly well-formed DAG must have a path between the vertex that invokes the `fcreate` and the vertex that invokes the `ftouch`. This is trivially satisfied in λ_i^4 because allocation and creation are inextricably linked, but in I-Cilk a thread handle allocation can be separate from its thread creation. Thus, such a requirement is not trivially satisfied, and the programmer has to manually ensure the thread has been created before an `ftouch`.

4.3 Runtime Scheduler

An execution of an I-Cilk program generates a computation DAG as described in Section 2 that dynamically unfolds on the fly, and the underlying runtime schedules the computation in a way that respects the dependences in the DAG. I-Cilk, like Cilk-F, schedules the computation using proactive work stealing [59] but in addition, prioritizes threads.

Recall from Section 2 that one can bound the response times of threads in a well-formed DAG (Theorem 2.1), provided that the schedule is admissible and ***prompt***, i.e., the schedule assigns a ready vertex u such that no currently unassigned vertex is higher-priority than u. Any schedule produced by an actual execution is admissible by construction. Promptness, however, requires the scheduler to find ready vertices of high-priority threads in the system to assign before vertices of lower-priority threads. Doing so requires maintaining centralized information, which becomes inefficient in practice due to frequent synchronizations. Thus, I-Cilk implements a scheduler that approximates promptness.

Specifically, I-Cilk uses a two-level scheduling scheme, similar to the scheme proposed by prior work A-STEAL [6, 7]. The top-level ***master*** scheduler determines how to best assign processing cores to different priority levels, and threads within each priority level are scheduled with a second-level ***work-stealing*** scheduler [8, 16], known for its decentralized scheduling protocol with low overhead and provably efficient execution time bound. I-Cilk utilizes a variant of work stealing called ***proactive work stealing*** [59] inherited from Cilk-F, the baseline scheduler I-Cilk extends.

The master scheduler evaluates the cores-to-priority-level assignments in a fixed scheduling interval, called the ***scheduling quantum***. The master assigns cores based on the desired number of cores reported by the work-stealing schedulers of each priority-level, but in a way that prioritizes high-priority threads — it always assigns cores in the order of

[3] We additionally provide macros for declaring and defining a `command` function to ease the use of `command` functions.

[4] Currently this is enforced by name mangling `command` functions which can be circumvented, but in principle this can be enforced with better compiler support.

priority. Thus, the highest priority always gets its requested cores up to the limit of what is available on the system, and the next levels get the left-over cores.

The work-stealing scheduler at each priority level maintains its ***desire***, the number of cores it wishes to get. At the end of a quantum, the scheduler for a given priority level determines its core utilization in this quantum and re-evaluates its desire based on the measured utilization and whether its desire was satisfied in this quantum. Because a work-stealing scheduler is either doing useful work (making progress on the computation), or attempting to steal (which leads to load balancing), its ***utilization*** is computed by the fraction of processing cycles that went into doing work. If its utilization exceeded a fixed threshold (e.g., 90%) and its desire was satisfied (i.e., it got its desired number of cores), it increases its desire by a multiplicative factor of the ***growth parameter*** γ. For instance, if $\gamma = 2$, double the desire. On the other hand, if the utilization exceeded the threshold but its desire was not met, it keeps the same desire. Finally, if the utilization did not meet the threshold, it reduces its desire by a factor of γ (e.g., if $\gamma = 2$, halve the desire).

Prior work [4, 5, 7] has analyzed similar two-level strategies and shown that one can bound the wasted cycles (i.e., due to low utilization) and the execution time of computations scheduled by the second-level schedulers. The prior analyses do not directly apply in our case, however, for two reasons. First, I-Cilk utilizes proactive work stealing for the second-level schedulers, which differs from the ones analyzed in prior work. Second, in prior work, the computations scheduled by the second-level schedulers are independent, whereas in our case, each second-level scheduler corresponds to a priority level, and threads in different priority levels can have dependences. Nevertheless, in Section 5, we show that our scheduler does appropriately prioritize high-priority threads over low-priority ones and provides better response time for high-priority threads compared to the baseline system that does not account for priorities.

5 Evaluation of I-Cilk

This section empirically evaluates I-Cilk. To evaluate the practicality and usability of the type system, we wrote three moderately sized application benchmarks: a proxy server (`proxy`, 1.5K LoC), a multi-user email client (`email`, 1.1K LoC), and a job server (`jserver`, 1.1K LoC).[5] The type system helps the programmer ensure that there is no priority inversion, which is not always easy to tell, as thread handles are often used to coordinate interactions among different application components. We also use the same applications to evaluate the efficiency of the scheduler by comparing I-Cilk against Cilk-F, the baseline system that utilizes proactive work stealing but does not account for the priority of threads (and thus does not incur the two-level scheduling overhead). For fair comparison, Cilk-F is also equipped with the same `io_future` library that performs I/O operations in a latency-hiding way. We use this library for the I/O operations in the benchmarks so that I/O-blocked threads do not hinder parallelism. The empirical results indicate that I-Cilk was able to prioritize high-priority threads and thus provide shorter response times.

[5]LoC exclude comments, system libraries, and runtime code.

Experimental Setup. Our experiments ran on a computer with 2 Intel Xeon Gold 6148 processors with 20 2.40-GHz cores. Each core has a 32-kB L1 data and 32-KB L1 instruction cache, and a private 1 MB L2 cache. Hyperthreading was enabled, and each core had 2 hardware threads. Both processors have a 27.5 MB shared L3 cache, and there are 768 GB of main memory. I-Cilk and all benchmarks were compiled using the Tapir compiler [57] (based on clang 5.0.0), with -O3 and -flto. Experiments ran in Linux kernel 4.15.

5.1 Application Case Studies

We evaluate the type system with three applications representative of interactive applications in the real world in that they utilize interesting features commonly used to write such applications, such as low-level file system and network libraries, and concurrent data structures implemented using primitives such as fetch-and-add and compare-and-swap. Due to space limitations, we discuss the email client in detail but only summarize the other two applications.

Proxy server. The first application, `proxy`, allows multiple clients to connect and request websites by their URL. The server fetches the website on the client's behalf, masking the client's IP address. As an optimization, the server maintains a cache of website contents using a concurrent hashtable. If a website is cached, the server can respond with it immediately. The application utilizes components with four priority levels, listed in order from highest to lowest: a) the loop that accepts client connections and the per-client event loop that handles the client requests, b) a component that fetches websites in the event of a cache miss, c) a component that logs statistics, and the lowest is d) the main function that performs server startup / shutdown. The priority specification favors response time for client requests.

Email client. The second application `email` is a multi-user, shared email client that allows users with individual mailboxes to sort messages, send messages, and print messages; a background task also runs periodically to reduce storage overhead by compressing each user's messages using Huffman codes [24][Chp. 16.3]. The application contains components with six priority levels, listed in order from highest to lowest: a) an event loop to handle user requests, b) a send component that sends email, c) a sort component that sorts emails, d) a compress component to compress emails and a print component to uncompress and send the uncompressed emails to the printer, e) a check component that periodically checks for the need to compress and fires off compression, f) the main function that performs shutdown.

One interesting feature is that the application requires the print and compress to interact with one another — if the user asks to print a particular email but it is in the midst of being compressed, the print component needs to coordinate with the compress component and wait for it to finish. Similarly, the compress component may encounter an email that it is about to compress, but it is in the midst of being printed, and thus the compress needs to wait for the print to complete.

To enable this, within each user's inbox data structure is an array indexed using the email ID where any thread attempting to print or compress the email will store its own handle. For instance, say there is an ongoing print thread for an email. The array slot corresponding to the email stores (a pointer to) the handle of the print thread. If a compress thread for the same email is created, the first thing the compress thread does is perform a compare-and-swap (CAS) on the same array slot, swapping out the handle of the print thread and inserting a pointer to its own handle into the slot. Assuming that CAS returns a non-null reference, the compress thread invokes `ftouch` on the reference to ensure that the printing is done before proceeding with the compress.

A print thread performs similar operations on the array to coordinate with an ongoing compress thread for the same email. Such an interaction is achieved by utilizing the thread handles and mutable state in an interesting way.

Job server. The `jserver` application executes jobs that arrive in the system using a smallest-work-first policy, i.e., given different types of jobs, the server knows the amount of work entailed for each type, and it prioritizes jobs with the least amount of work. We simulate user inputs using a Poisson process to generate jobs at random intervals and execute them. The priority levels correspond to the types of jobs. We simulated four different types of jobs with fixed input size n, listed in order of priority (high to low): a) parallel divide-and-conquer matrix multiplication (`matmul`, $n = 1024$), b) fibonacci (`fib`, $n = 36$), c) parallel merge sort (`sort`, $n = 1.1 \times 10^7$), and d) Smith-Waterman for sequence alignment (`sw`, $n = 1024$). This application differs from the previous two in that threads in different priority levels are independent of each other, and it is constructed so that we can easily modify the workload to simulate a server that is lightly loaded to heavily loaded.

Compilation time. Because the type system heavily utilizes templates, we measure its effect by comparing the compilation time and resulting binary sizes between code that uses priorities and code that does not.[6] As shown in Figure 1, the use of templates for enforcing the typing rules incurs acceptable overhead.

[6]The use of template can increase code size as each type instantiation of a given template gets its own code clone.

Table 1. The compilation times and resulting binary sizes of application code without and with priority. The compilation time is in seconds and the maximum out of the three compile runs. The binary size is in KB. The numbers in parentheses show overhead compared to the no priority version.

case study	*compilation time*	*binary size*
`proxy` (w/out)	1.95 (1.00×)	824.0 (1.00×)
`proxy` (with)	2.48 (1.27×)	974.7 (1.18×)
`email` (w/out)	4.66 (1.00×)	1241.16 (1.00×)
`email` (with)	5.40 (1.16×)	1454.58 (1.17×)
`jserver` (w/out)	2.10 (1.00×)	851.2 (1.00×)
`jserver` (with)	2.67 (1.27×)	987.7 (1.16×)

5.2 Empirical Evaluation

To evaluate the efficiency of our implementation, we compare the applications' running times on I-Cilk and on Cilk-F with the same latency-hiding I/O support. The main distinctions between the two systems are that a) I-Cilk prioritizes high-priority threads whereas Cilk-F does not; and b) I-Cilk utilizes the two-level scheduling scheme discussed in Section 4 whereas Cilk-F utilizes proactive work stealing only. For I-Cilk, we ran all applications with the following runtime parameters: utilization threshold of 90%, quantum length of 500 microseconds, and growth parameter of 2. These parameter values seem to work well in general.

Each of the applications represents different workload characteristics. The `proxy` server has the most I/O latency and very little computation. The `email` has a fair amount of I/O latency and slightly more computation than `proxy`. The `jserver` has little I/O latency with compute-intensive workloads. We use one socket (20 cores) to run the server and the second socket to simulate clients that generate inputs. Each application is evaluated with multiple server load configurations that range from lightly loaded to heavily loaded. For `proxy` and `email`, we ran with 90, 120, 150, and 180 connections. As we increase the number of connections, each core needs to multiplex among more connections. For `email`, the computation load also increases as the number of clients increases. For `jserver`, we simulated the job generations so that the workload results server machine utilization of 64%, 77%, 95%, and > 95% respectively.

For each application, we run the server for at least 15 seconds, during which tens of thousands of threads from various priority levels (which correspond to different application components) are created, and we measure their duration. Specifically, we measure the ***response*** time of the application, which corresponds to the time elapsed between when the user / client sends the request to when the server handles the request (which is always handled by the highest priority thread), and the ***compute*** time for each thread of different priority levels.

The standard deviation for such time measurements can be high for interactive applications, due to multiple factors. First, the timing includes the I/O latency, which is not always

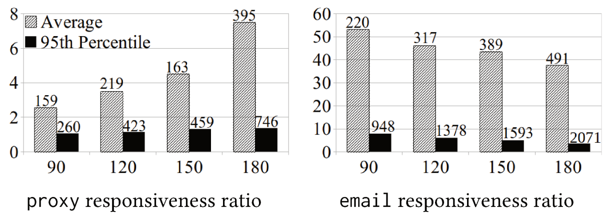


Figure 8. The relative responsiveness of proxy and email, measured as the response time running on Cilk-F normalized by I-Cilk response time, so higher means I-Cilk is more responsive. Grey bars show the responsiveness calculated using average response time and black bars show that calculated using the response time at 95^{th} percentile. The x-axis shows the number of client connections used. The numbers shown on top of the bars are the latency measured in microseconds for I-Cilk.

uniform. Second, the server is time-multiplexing among multiple client connections, and thus the measured time of a thread includes not only its computation time but also the time it took the server to get to the threads. As such, for many interactive applications, what one cares about is the latency near the tail. Thus, for all timing data, we show both the average time and the 95^{th} percentile running time (i.e., 95% of the measured time is below that value).

Figure 8 shows the response time ratio for `proxy` and `email` (the job server does not have a response time measurement as the jobs are generated in the same process as the server). We normalize the response time of Cilk-F by that of I-Cilk, and thus higher means I-Cilk is more responsive. As can be seen, I-Cilk provides much better response time, appropriately prioritizing the highest priority threads. I-Cilk appears to be much more responsive for `email` than for `proxy`. This is because `proxy` is very lightly-loaded — most of the time cores are idling, as there isn't much computation in the server execution (mostly I/O operations). In contrast, `email` has more computations to keep cores occupied, and thus high-priority threads can be delayed much longer in Cilk-F as the cores are pre-occupied by computations generated by lower-priority threads.

The high responsiveness is achieved by prioritizing the high-priority threads, sometimes at the expense of the lower priority threads. Figure 9 shows the computation times of threads from different components. For a given application and a given configuration (e.g., `proxy` with 90 clients), the bars from left to right show the normalized compute time for threads from higher to lower priority. As the figures show, I-Cilk provides better compute time than Cilk-F for the highest priority threads in the figures (which is the second highest priority for `proxy` and `email`). However, the lower priority threads can run slower. This trend can be seen across different server loads, where the compute time ratio for the higher priority threads grows larger as the load gets heavier. This is because the compute time for the higher priority threads on Cilk-F degrades as the server gets more heavily loaded, whereas I-Cilk is able to maintain similar level of quality of service. For the lower-priority threads, compute time on both systems degrades, with I-Cilk degrading more especially when the load gets heavy.

6 Related Work

Cooperative Parallelism. Many languages and systems have been developed for cooperative parallelism over the years. A large number of these, such as Id [9], Multilisp [33], NESL [15] and parallel versions of Haskell [20, 39] and ML [29, 32, 38, 53, 66], have focused on functional programming languages, in which the issues of races and deadlock do not arise or are limited; progress, however, has also been made toward handling some effects efficiently [32, 66]. These languages typically assume the fork-join model of parallelism, but there have also been advances in generalizing them to include the broader set of parallelism primitives such as futures [2].

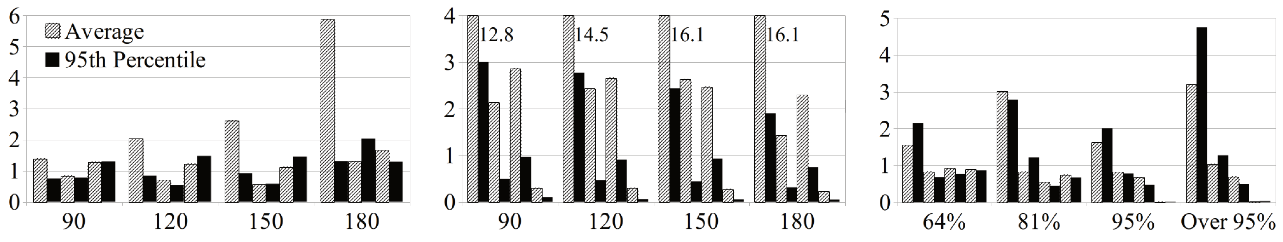
Some parallel language extensions have targeted popular imperative programming languages such as C [30] and Java [17, 21, 36, 42]. Many papers have been devoted over the years to taming races (e.g. [27, 43, 54, 64, 67]) and deadlock (e.g. [3, 22, 23, 65]). None of these languages allow the cooperative threads to be prioritized; doing so, as we do in this work, requires reasoning about *priority inversions* in addition to the problems mentioned above.

Scheduling for Responsiveness. Responsiveness has long been a concern in the systems community, as operating systems must schedule processes and threads, many of which are interactive. A thorough overview of this topic can be found in a text by Silberschatz et al. [58]. In contrast to cooperative parallel systems, OS schedulers deal with relatively small numbers of threads.

Many threading systems for which responsiveness is a concern incorporate some notion of priority. The problem of *priority inversion* has been noted in systems as early as Mesa [41]. Babaoğlu et al. [10] formalized the idea of priority inversions and discussed some techniques by which they could be prevented.

Recent work [48, 49] has introduced thread priorities into a cooperative parallel system and developed type systems for ruling out priority inversions that arise through touching a future. That work, however, targets purely functional programming, and so future handles can essentially only be passed through calls and returns, leading to a well-behaved DAG structure. In this paper, future handles can additionally be passed through mutable state, leading to much more complicated reasoning about priority inversions.

Cost Semantics. Cost semantics (e.g., [46, 55, 56]) are used to reason statically about the resource usage, broadly construed, of programs. Cost semantics for parallel programs [1, 13, 14, 62] typically represent the parallel structure of the



`proxy` compute time ratio

`email` compute time ratio. Data labels indicate the values of bars that exceed the scale of the graph.

`jserver` compute time ratio

Figure 9. The relative compute time for `proxy`, `email`, and `jserver`, measured as the compute time running on Cilk-F normalized by that running on I-Cilk, so higher means I-Cilk computes faster. For a given application and a given configuration (e.g., `proxy` with 90 clients), the bars from left to right show the compute time ratio for threads from the highest to the lowest priority. The grey bars show the compute time ratio calculated using average compute time and the black bars show that calculated using the compute time at 95^{th} percentile. The x-axis shows the number of client connections used for `proxy` and `email`, and the server utilization for `jserver`.

program as a DAG. Offline scheduling results bound the time required to execute such a DAG on P processors in terms of the work and span of the DAG. Classic offline scheduling results have shown that a "level-by-level" schedule [18] and any greedy schedule [26] are within a factor of two of optimal. Although the full details of the DAG model are usually reserved for proofs, the metrics of work and span and the scheduling results above are quite useful in practice for analyzing programs by thinking in terms of the parallel structure of the underlying algorithm (e.g., the branching factor and problem size in a divide-and-conquer algorithm). Even in cases where the input is unknown, one can reason asymptotically about work and span, much like asymptotic reasoning in sequential algorithms. Recent work has extended parallel cost semantics to reason about I/O latency [47] and responsivenesss [48, 49]. This paper further extends the state of the art by adding *weak edges* that allow DAGs to reflect information passed between threads through global state.

Much of the above prior work has drawn a distinction between the cost semantics, which uses a very abstract evaluation model to produce a cost DAG from a program, and a *provably-efficient* or *bounded implementation* [14, 48, 49], which counts the steps of an abstract machine. A proof that the abstract machine actually meets the bounds promised by the cost semantics can be quite technical and involved. In this work, we present one dynamic semantics that both counts steps and produces a cost graph. This semantics reflects execution ordering, which is important in our calculus, and simplifies the proof that the steps of the abstract machine are bounded by the cost semantics.

7 Conclusion

This paper bridges cooperative and competitive threading models by bringing together a classic threading construct, futures, with priorities and mutable state. To facilitate reasoning about efficiency and responsiveness, the paper extends the traditional graph-based cost models for parallelism to account for priorities and mutable state. The cost model applies only to computations that are free of priority inversions. To guarantee their absence, we present a formal calculus called λ_i^4 and a type system that disallows priority inversions. The cost model and the type system both rely on a novel technical device, called *weak edges*, that represent run-time happens-before dependencies that arise due to communication via mutable shared state. We show that these theoretical results are practical by presenting a reasonably faithful implementation that extends C++ with futures and priorities. This extension offers an expressive substrate for writing interactive parallel programs and is able to enforce the absence of priority inversions if the programmer avoids certain unsafe constructs of C++. Our empirical evaluation shows that the techniques work well in practice.

Acknowledgments

We thank Chris Seaton and the anonymous reviewers for their helpful feedback. This work was partially supported by the National Science Foundation under grant numbers CCF-1901381, CCF-1408940, CCF-1527692, CCF-1733873, CCF-1910568, CCF-1725647 and CCF-1439062.

References

[1] 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.

[2] Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. 2016. Dag-calculus: A Calculus for Parallel Computation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. 18–32.

[3] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K. Shyamasundar, and Katherine Yelick. 2007. Deadlock-free Scheduling of X10 Computations with Bounded Resources. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '07)*. ACM, New York, NY, USA, 229–240. https://doi.org/10.1145/1248377.1248416

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

[5] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. 2008. Adaptive scheduling with parallelism feedback. *ACM Transactions on Computing Systems* 16, 3 (2008), 7:1–7:32.

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

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

[8] 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.

[9] Arvind and K. P. Gostelow. 1978. *The Id Report: An Asychronous Language and Computing Machine*. Technical Report TR-114. Department of Information and Computer Science, University of California, Irvine.

[10] Özalp Babaoğlu, Keith Marzullo, and Fred B. Schneider. 1993. A Formalization of Priority Inversion. *Real-Time Systems* 5, 4 (1993), 285–303.

[11] Rajkishore Barik, Zoran Budimlić, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. 2009. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, Orlando, Florida, USA, 735–736.

[12] 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.

[13] Guy Blelloch and John Greiner. 1995. Parallelism in sequential functional languages. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*. ACM, 226–237.

[14] Guy E. Blelloch and John Greiner. 1996. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*. ACM, 213–225.

[15] 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.

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

[17] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09)*. 97–116.

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

[19] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. 51–61.

[20] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data parallel Haskell: a status report. In *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France, January 16, 2007*. 10–18.

[21] 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.

[22] Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. 2015. Dynamic Deadlock Verification for General Barrier Synchronisation. (2015), 150–160. https://doi.org/10.1145/2688500.2688519

[23] Tiago Cogumbreiro, Rishi Surendran, Francisco Martins, Vivek Sarkar, Vasco T. Vasconcelos, and Max Grossman. 2017. Deadlock Avoidance in Parallel Programs with Futures: Why Parallel Tasks Should Not Wait for Strangers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 103 (Oct. 2017), 26 pages. https://doi.org/10.1145/3143359

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (third ed.). The MIT Press.

[25] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. 2008. Programming with exceptions in JCilk. *Science of Computer Programming* 63, 2 (Dec. 2008), 147–171.

[26] 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.

[27] Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *ACM Symposium on Parallel Algorithms and Architectures*. 1–11.

[28] 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.

[29] 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.

[30] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 212–223.

[31] Cao Gao, Anthony Gutierrez, Ronald G. Dreslinski, Trevor Mudge, Krisztian Flautner, and Geoffery 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.

[32] 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.

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

[34] Robert H. Halstead, Jr. 1984. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming (LFP '84)*. ACM, 9–17.

[35] 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.

[36] 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.

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

[38] Suresh Jagannathan, Armand Navabi, KC Sivaramakrishnan, and Lukasz Ziarek. 2010. The Design Rationale for Multi-MLton. In *ML*

'10: Proceedings of the ACM SIGPLAN Workshop on ML. ACM.

[39] 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.

[40] 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

[41] Butler W. Lampson and David D. Redell. 1980. Experience with Processes and Monitors in Mesa. *Commun. ACM* 23, 2 (1980), 105–117.

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

[43] I-Ting Angelina Lee and Tao B. Schardl. 2015. Efficiently Detecting Races in Cilk Programs That Use Reducer Hyperobjects. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*. ACM, New York, NY, USA, 111–122. https://doi.org/10.1145/2755573.2755599

[44] 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.

[45] Charles E. Leiserson. 2010. The Cilk++ Concurrency Platform. *J. Supercomputing* 51, 3 (2010), 244–257.

[46] Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. 2008. *A Cost Semantics for Self-Adjusting Computation*. Technical Report CMU-CS-08-141. Department of Computer Science, Carnegie Mellon University.

[47] 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

[48] 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.

[49] Stefan K. Muller, Umut A. Acar, and Robert Harper. 2018. Competitive Parallelism: Getting Your Priorities Right. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '18)*.

[50] Stefan K. Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. 2020. Responsive Parallelism with Futures and State. (2020). arXiv:cs.PL/2004.02870

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

[52] OpenMP 5.0 2018. *OpenMP Application Programming Interface, Version 5.0*. Accessed in July 2018.

[53] 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.

[54] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, Beijing, China, 531–542.

[55] Mads Rosendahl. 1989. Automatic complexity analysis. In *FPCA '89: Functional Programming Languages and Computer Architecture*. ACM, 144–156.

[56] David Sands. 1990. Complexity Analysis for a Lazy Higher-Order Language. In *ESOP '90: Proceedings of the 3rd European Symposium on Programming*. Springer-Verlag, London, UK, 361–376.

[57] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, Austin, Texas, USA, 249–265. http://doi.acm.org/10.1145/3018743.3018758

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

[59] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. 2019. Proactive Work Stealing for Futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 257–271. https://doi.org/10.1145/3293883.3295735

[60] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. 2019. ProWS - Proactive Work Stealing for Futures. Available at https://github.com/wustl-pctg/ProWS. (2019). Accessed on July 2019.

[61] 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.

[62] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. 2008. Space Profiling for Parallel Functional Programs. In *International Conference on Functional Programming*.

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

[64] Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. ACM, Asilomar State Beach, CA, USA, 83–94.

[65] Caleb Voss, Tiago Cogumbreiro, and Vivek Sarkar. 2019. Transitive Joins: A Sound and Efficient Online Deadlock-avoidance Policy. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 378–390. https://doi.org/10.1145/3293883.3295724

[66] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)*.

[67] Yifan Xu, I-Ting Angelina Lee, and Kunal Agrawal. 2018. Efficient Parallel Determinacy Race Detection for Two-dimensional Dags. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 368–380. https://doi.org/10.1145/3178487.3178515